O'REILLY

JAVA Reference Library

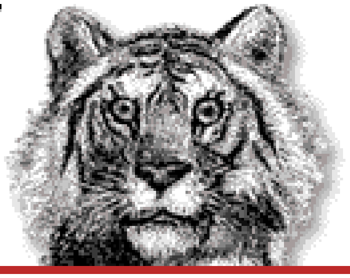
As of February 9, 1998, the entire online Java Reference Library reflects version 1.4 of the Java Deluxe CD product, which will be available at the end of February. This version includes the updated files for Exploring Java, Second Edition (published October 1997), plus minor revisions to the other files.

Combined Index
Combined Search
Web Version
Credits

Library Home | Java in a Nutshell | Java Language Reference | Java AWT Reference | Java Fundamental Classes Reference | Exploring Java

Questions, comments, and suggestions to bookquestions@ora.com.

O'REILLY



JAVA IN A NUTSHELL

Java in a Nutshell
By David Flanagan; 1-56592-262-X, 628 pages.
2nd Edition, May 1997

Table of Contents

Part I: Introducing Java

Part I is an introduction to Java and Java programming. If you know how to program in C or C++, these chapters teach you everything you need to know to start programming with Java.

If you are already familiar with Java 1.0 you may want to just skip ahead to Part II, which introduces the new features of Java 1.1.

Part II: Introducing Java 1.1

The two chapters in this part introduce the new features of Java 1.1. Chapter 4 is an overview of the new APIs, and Chapter 5 explains the new language syntax. See Part III for some examples of the new features.

Part III: Programming with the Java 1.1 API

Part III contains examples of programming with the new features of Java 1.1. You can study and learn from the examples, and you should feel free to adapt them for use in your own programs. The examples shown in these chapters may be downloaded from the Internet. See http://www.ora.com/catalog/books/javanut2/. Some of the chapters in this part also contain tables and other reference material for new features in Java 1.1.

Part III of this book is "deprecated." Most of the examples from the first edition of this book do not appear here, and Part III may disappear altogether in the next edition of the book. Unfortunately, as Java continues to grow, there is less and less room for programming examples in this book. However, all of the examples from the first edition are still available on the Web page listed above.

Part IV: Java Language Reference

Part IV contains reference material on the Java language and related topics. Chapter 13 contains a number of useful summary tables of Java syntax. Chapter 14 describes the standard Java system properties and how to use them. Chapter 15 covers the syntax of the HTML tags that allow you to include Java applets in Web pages. Chapter 16 documents the command-line syntax for the Java compiler, interpreter, and other tools shipped with the JDK.

Part V: API Quick Reference

Part V is the real heart of this book: quick-reference material for the Java API. Please read the following section, *How to Use This Quick Reference*, to learn how to get the most out of this material.

Search the text of *Java in a Nutshell*.

Library Home | Java in a Nutshell | Java Language Reference | Java AWT Reference | Java Fundamental Classes Reference | Exploring Java

Preface

Contents:

This handbook is a desktop quick reference for Java programmers; it covers version 1.1 of the Java language and API. It also includes introductory and tutorial material for C and C++ programmers who want to learn Java. It was written to sit faithfully by your keyboard for easy reference while you program. The wild success of the first edition has shown that this is exactly what Java programmers want, and I've retained the "no fluff" explanations and the to-the-point reference material in this second edition. I hope that new readers will find this book useful, and that old readers will find it even more useful than the last one!

Contents of This Book

This book is divided into five parts:

Part I: Introducing Java

> This first part of the book introduces Java and Java programming, with a particular emphasis on helping C and C++ programmers make the transition to Java. If you are already familiar with Java 1.0 programming, you can skip the three chapters in this part.

Part II: Introducing Java 1.1

This second part of the book contains two chapters that introduce the new features of the Java 1.1 API and the new language features in Java 1.1.

Part III: Programming with the Java 1.1 API

This part contains example programs that demonstrate many of the new features of Java 1.1. You may find that these examples are a good starting point for your own programs, and you should feel free to adapt them for your own use. As explained below, this example section has changed a lot since the first edition of this book.

Part IV: Java Language Reference

This part of the book contains reference material that describes the syntax of the Java language and the tools provided with the Java Development Kit (JDK), among other things.

Part V: API Quick Reference

This part is a quick reference for the Java API; it forms the bulk of the book. Please be sure to read the *How To Use This Quick Reference* material, which appears at the beginning of the part. It explains how to get the most out of the reference material.

Changes Since the First Edition

The many changes in Java 1.1 have resulted in changes to this book. The most significant change since the first edition is a direct result of the large size of Java 1.1: Java has grown too large to fit in a single book, even in quick-reference form. Thus, we need to split *Java in a Nutshell* into multiple volumes. This volume, the "original" *Java in a Nutshell* documents the most commonly used features of Java, and it is an indispensable volume for all Java programmers.

We are planning to publish a separate volume that covers the Java "Enterprise APIs," which include the database connectivity, remote method invocation, and security features of Java 1.1, as well as other forthcoming components, such as CORBA IDL support and the electronic commerce framework. And as new Java APIs are developed and released, we may consider adding new volumes to the *Java in a Nutshell* series.

While I was working on this second edition of *Java in a Nutshell*, it became clear that, even without the enterprise material, the book was becoming too long. (Too long, that is, to remain a useful quick reference, and too long to keep at an affordable price.) Something had to give. The most logical solution was to remove the example programs, which are tutorial in nature, from the book, which is a quick-reference at heart. However, we didn't want to surprise faithful readers by removing the examples altogether, so we decided to pare down the example chapters to the bare minimum. You'll notice that Part III contains examples of using the new Java 1.1 features, such as the JavaBeans API and object serialization, but it does not contain the majority of the old examples from the first edition. For now, Part III contains useful examples for experienced Java programmers who want to learn about the new features of Java 1.1. When Java 1.2 is released, though, we expect that we will have to remove the example section entirely.

Readers familiar with the first edition of *Java in a Nutshell* will notice some other changes as well. The table of contents has been rearranged to accommodate all the new material. We've used a new easier-to-read font for code listings. And we've included cross-reference material (that used to be available only in separate index chapters) directly in the quick-reference section, which should make that section substantially more useful. Be sure to read *How To Use This Quick Reference* at the beginning of the reference section to learn about these and other changes to the quick-reference format.

PREVIOUS
Contents of This Book

HOME
BOOK INDEX

NEXT
Related Books

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Related Books

O'Reilly & Associates is developing an entire series of books on Java. This series consists of introductory books, reference manuals, and advanced programming guides.

The following books on Java are currently available or due to be released soon from O'Reilly & Associates:

Exploring Java, by Patrick Niemeyer and Joshua Peck

> A comprehensive tutorial that provides a practical, hands-on approach to learning Java.

Java Language Reference, by Mark Grand

> A complete reference for the Java programming language itself.

Java AWT Reference, by John Zukowski

> A complete reference manual for the AWT-related packages in the core Java API.

Java Fundamental Classes Reference, by Mark Grand and Jonathan Knudsen

> A complete reference manual for the `java.lang`, `java.io`, `java.net`, `java.util` packages, among others, in the core Java API.

Java Virtual Machine, by Jon Meyer and Troy Downing

> A programming guide and reference manual for the Java Virtual Machine.

Java Threads, by Scott Oaks and Henry Wong

An advanced programming guide to working with threads in Java.

Java Network Programming, by Elliote Rusty Harold

A complete guide to writing sophisticated network applications.

Database Programming with JDBC and Java, by George Reese

An advanced tutorial on JDBC that presents a robust model for developing Java database programs.

Developing Java Beans, by Robert Englander

A complete guide to writing components that work with the JavaBeans API.

Look for additional advanced programming guides on such topics as distributed computing and electronic commerce from O'Reilly in the near future.

PREVIOUS Changes Since the First Edition | HOME BOOK INDEX | NEXT Java Resources

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

PREVIOUS Preface NEXT

Java Resources

Sun has online reference documentation for the Java API that you may find useful in conjunction with this quick reference handbook. Visit http://www.javasoft.com/ to view or download this API documentation and other useful documents.

There are many other sites on the Web that contain useful Java information. One of the most well-known is http://www.gamelan.com/, also known as http://java.developer.com/. For discussion (in English) about Java, try the various *comp.lang.java.** newsgroups.

PREVIOUS HOME NEXT
Related Books BOOK INDEX Java in a Nutshell Web Sites

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Java in a Nutshell Web Sites

The Web site for this book is http://www.ora.com/catalog/books/javanut2/. There you will find the examples from this book, available for download. As typos are reported, you may also find an errata list at that Web site.

My personal Web site is http://www.DavidFlanagan.COM/. This is a new site, just getting off the ground as this book goes to press, but it will eventually contain a number of Java programming resources, including commercial and shareware tools and "beans" that I have written.

PREVIOUS
Java Resources

HOME
BOOK INDEX

NEXT
Conventions Used in This Book

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Conventions Used in This Book

Italic is used for:

- Pathnames, filenames, and program names.
- New terms where they are defined.
- Internet addresses, such as domain names and URLs.

Boldface is used for:

- Particular keys on a computer keyboard.
- Names of user interface buttons and menus.

`Constant Width` is used for:

- Anything that appears literally in a Java program, including keywords, data types, constants, method names, variables, class names, and interface names.
- Command lines and options that should be typed verbatim on the screen.
- All Java code listings.
- HTML documents, tags, and attributes.
- Method parameters, and general placeholders that indicate that an item is replaced by some actual value in your own program.
- Variable expressions in command-line options.

- Java class synopses in the quick-reference section. This very narrow font allows us to fit a lot of information on the page without a lot of distracting line breaks.
- Highlighting class, method, field, and constructor names in the quick-reference section, which makes it easier to scan the class synopses.
- Method parameter names and comments in the quick-reference section.

PREVIOUS | HOME | NEXT
Java in a Nutshell Web Sites | BOOK INDEX | Request for Comments

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Request for Comments

Please help us to improve future editions of this book by reporting any errors, inaccuracies, bugs, misleading or confusing statements, and plain old typos that you find anywhere in this book. Email your bug reports and comments to us at: *bookquestions@ora.com*. (Before sending a bug report, however, you may want to check for an errata list at http://www.ora.com/catalog/books/javanut2/ to see if the bug has already been submitted.)

Please also let us know what we can do to make this book more useful to you. We take your comments seriously and will try to incorporate reasonable suggestions into future editions.

Acknowledgments

Many people helped in the creation of this book and I am grateful to them all. I am indebted to literally hundreds of readers of the first edition who wrote in with comments, suggestions, bug reports, and praise. Their many small contributions are scattered throughout the book. Also, my apologies to those who made the many good suggestions that could not be incorporated into this edition.

Paula Ferguson, a friend and colleague, edited both editions of the book. Her careful reading and always-practical suggestions made the book stronger, clearer, and more useful. She is also the one who prodded me when I started to slack off, and got me back on track when I started trying to turn *Java in a Nutshell* into *Java in a Packing Crate*.

Mike Loukides provided high-level direction and guidance for the first edition of the book. Eric Raymond and Troy Downing reviewed that first edition--they helped spot my errors and omissions, and offered good advice on making the book more useful to Java programmers.

For the second edition, John Zukowski reviewed my Java 1.1 AWT quick-reference material, and George Reese reviewed most of the remaining new material. This edition was also blessed with a "dream team" of technical reviewers from Sun. John Rose, the author of the Java Inner Classes Specification, reviewed the chapter on inner classes. Mark Reinhold, author of the character stream classes in `java.io`, reviewed my documentation of these classes. Nakul Saraiya, the designer of the new Java Reflection API, reviewed my documentation of the `java.lang.reflect` package. I am very grateful to these engineers and architects; their efforts have made this a stronger, more accurate book. Any errors that remain are of course my own.

Nicole Gipson Arigo was the production editor for this edition of the book, taking over the job from John Files, who produced the first edition. Nicole coordinated the entire production process, entered changes from edited copy, and handled the meticulous task of fixing line and page breaks in the manuscript. Madeleine Newell provided production assistance. Clairemarie Fisher O'Leary, Jane Ellin, and Sheryl Avruch performed quality control checks. Seth Maislin wrote the index. Chris Reilley created the figures, including all the detailed class hierarchy diagrams in Part V. [1] Edie Freedman designed the cover. Nancy Priest designed the interior format of the book and Lenny Muellner carefully implemented the format in *troff*, with help from Ellen Siever.

The whole production team has my thanks for once again pulling together all the pieces to create the finished product you now hold in your hands.

As always, my thanks and love to Christie.

David Flanagan
April 1997

[1] The hierarchy diagrams are loosely based on similar diagrams for Java 1.0 by Charles L. Perkins.

PREVIOUS | HOME | NEXT
Request for Comments | BOOK INDEX | Getting Started with Java

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

1. Getting Started with Java

Contents:

When it was introduced in late 1995, Java took the Internet by storm. Java 1.1, released in early 1997, nearly doubles the speed of the Java interpreter and includes many important new features. With the addition of APIs to support database access, remote objects, an object component model, internationalization, printing, encryption, digital signatures, and many other technologies, Java is now poised to take the rest of the programming world by storm.

Despite all the hype surrounding Java and the new features of Java 1.1, it's important to remember that at its core, Java is just a programming language, like many others, and its APIs are just class libraries, like those of other languages. What is interesting about Java, and thus the source of much of the hype, is that it has a number of important features that make it ideally suited for programming in the heavily networked, heterogenous world of the late 1990s. The rest of this chapter describes those interesting features of Java and demonstrates some simple Java code. Chapter 4, *What's New in Java 1.1* explores the new features that have been added to version 1.1 of the Java API.

1.1 Why Is Java Interesting?

In one of their early papers about the language, Sun described Java as follows:

> Java: A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language.

Sun acknowledges that this is quite a string of buzzwords, but the fact is that, for the most part, they aptly describe the language. In order to understand why Java is so interesting, let's take a look at the language features behind the buzzwords.

Object-Oriented

Java is an *object-oriented* programming language. As a programmer, this means that you focus on the data in your application and methods that manipulate that data, rather than thinking strictly in terms of procedures. If you're accustomed to procedure-based programming in C, you may find that you need to change how you design your programs when you use Java. Once you see how powerful this new paradigm is, however, you'll quickly adjust to it.

In an object-oriented system, a *class* is a collection of data and methods that operate on that data. Taken together, the data and methods describe the state and behavior of an *object*. Classes are arranged in a hierarchy, so that a subclass can inherit behavior from its superclass. A class hierarchy always has a root class; this is a class with very general behavior.

Java comes with an extensive set of classes, arranged in *packages*, that you can use in your programs. For example, Java provides classes that create graphical user interface components (the `java.awt` package), classes that handle input and output (the `java.io` package), and classes that support networking functionality (the `java.net` package). The `Object` class (in the `java.lang` package) serves as the root of the Java class hierarchy.

Unlike C++, Java was designed to be object-oriented from the ground up. Most things in Java are objects; the primitive numeric, character, and boolean types are the only exceptions. Strings are represented by objects in Java, as are other important language constructs like threads. A class is the basic unit of compilation and of execution in Java; all Java programs are classes.

While Java is designed to look like C++, you'll find that Java removes many of the complexities of that language. If you are a C++ programmer, you'll want to study the object-oriented constructs in Java carefully. Although the syntax is often similar to C++, the behavior is not nearly so analogous. For a complete description of the object-oriented features of Java, see Chapter 3, *Classes and Objects in Java*.

Interpreted

Java is an an interpreted language: the Java compiler generates byte-codes for the Java Virtual Machine (JVM), rather than native machine code. To actually run a Java program, you use the Java interpreter to execute the compiled byte-codes. Because Java byte-codes are platform-independent, Java programs can run on any platform that the JVM (the interpreter and run-time system) has been ported to.

In an interpreted environment, the standard "link" phase of program development pretty much vanishes. If Java has a link phase at all, it is only the process of loading new classes into the environment, which is an incremental, lightweight process that occurs at run-time. This is in contrast with the slower and more cumbersome compile-link-run cycle of languages like C and C++.

Architecture Neutral and Portable

Because Java programs are compiled to an *architecture neutral* byte-code format, a Java application can run on any system, as long as that system implements the Java Virtual Machine. This is a particularly important for applications distributed over the Internet or other heterogenous networks. But the architecture neutral approach is useful beyond the scope of network-based applications. As an application developer in today's software market, you probably want to develop versions of your application that can run on PCs, Macs, and UNIX workstations. With multiple flavors of UNIX, Windows 95, and Windows NT on the PC, and the new PowerPC Macintosh, it is becoming increasingly difficult to produce software for all of the possible platforms. If you write your application in Java, however, it can run on all platforms.

The fact that Java is interpreted and defines a standard, architecture neutral, byte-code format is one big part of being *portable*. But Java goes even further, by making sure that there are no "implementation-dependent" aspects of the language specification. For example, Java explicitly specifies the size of each of the primitive data types, as well as its arithmetic behavior. This differs from C, for example, in which an `int` type can be 16, 32, or 64 bits long depending on the platform.

While it is technically possible to write non-portable programs in Java, it is relatively easy to avoid the few platform-dependencies that are exposed by the Java API and write truly portable or "pure" Java programs. Sun's new "100% Pure Java" program helps developers ensure (and certify) that their code is portable. Programmers need only to make simple efforts to avoid non-portable pitfalls in order to live up to Sun's trademarked motto "Write Once, Run Anywhere."

Dynamic and Distributed

Java is a *dynamic* language. Any Java class can be loaded into a running Java interpreter at any time. These dynamically loaded classes can then be dynamically instantiated. Native code libraries can also be dynamically loaded. Classes in Java are represented by the `Class` class; you can dynamically obtain information about a class at run-time. This is especially true in Java 1.1, with the addition of the Reflection API, which is introduced in Chapter 12, *Reflection*.

Java is also called a *distributed* language. This means, simply, that it provides a lot of high-level support for networking. For example, the `URL` class and OArelated classes in the `java.net` package make it almost as easy to read a remote file or resource as it is to read a local file. Similarly, in Java 1.1, the Remote Method Invocation (RMI) API allows a Java program to invoke methods of remote Java objects, as if they were local objects. (Java also provides traditional lower-level networking support, including datagrams and stream-based connections through sockets.)

The distributed nature of Java really shines when combined with its dynamic class loading capabilities. Together, these features make it possible for a Java interpreter to download and run code from across the

Internet. (As we'll see below, Java implements strong security measures to be sure that this can be done safely.) This is what happens when a Web browser downloads and runs a Java applet, for example. Scenarios can be more complicated than this, however. Imagine a multi-media word processor written in Java. When this program is asked to display some type of data that it has never encountered before, it might dynamically download a class from the network that can parse the data, and then dynamically download another class (probably a Java "bean") that can display the data within a compound document. A program like this uses distributed resources on the network to dynamically grow and adapt to the needs of its user.

Simple

Java is a *simple* language. The Java designers were trying to create a language that a programmer could learn quickly, so the number of language constructs has been kept relatively small. Another design goal was to make the language look familiar to a majority of programmers, for ease of migration. If you are a C or C++ programmer, you'll find that Java uses many of the same language constructs as C and C++.

In order to keep the language both small and familiar, the Java designers removed a number of features available in C and C++. These features are mostly ones that led to poor programming practices or were rarely used. For example, Java does not support the `goto` statement; instead, it provides labelled `break` and `continue` statements and exception handling. Java does not use header files and it eliminates the C preprocessor. Because Java is object-oriented, C constructs like `struct` and `union` have been removed. Java also eliminates the operator overloading and multiple inheritance features of C++.

Perhaps the most important simplification, however, is that Java does not use pointers. Pointers are one of the most bug-prone aspects of C and C++ programming. Since Java does not have structures, and arrays and strings are objects, there's no need for pointers. Java automatically handles the referencing and dereferencing of objects for you. Java also implements automatic garbage collection, so you don't have to worry about memory management issues. All of this frees you from having to worry about dangling pointers, invalid pointer references, and memory leaks, so you can spend your time developing the functionality of your programs.

If it sounds like Java has gutted C and C++, leaving only a shell of a programming language, hold off on that judgment for a bit. As we'll see in Chapter 2, *How Java Differs from C*, Java is actually a full-featured and very elegant language.

Robust

Java has been designed for writing highly reliable or *robust* software. Java certainly doesn't eliminate the need for software quality assurance; it's still quite possible to write buggy software in Java. However, Java does eliminate certain types of programming errors, which makes it considerably easier to write reliable software.

Java is a strongly typed language, which allows for extensive compile-time checking for potential type-mismatch problems. Java is more strongly typed than C++, which inherits a number of compile-time laxities from C, especially in the area of function declarations. Java requires explicit method declarations; it does not support C-style implicit declarations. These stringent requirements ensure that the compiler can catch method invocation errors, which leads to more reliable programs.

One of the things that makes Java simple is its lack of pointers and pointer arithmetic. This feature also increases the robustness of Java programs by abolishing an entire class of pointer-related bugs. Similarly, all accesses to arrays and strings are checked at run-time to ensure that they are in bounds, eliminating the possibility of overwriting memory and corrupting data. Casts of objects from one type to another are also checked at run-time to ensure that they are legal. Finally, and very importantly, Java's automatic garbage collection prevents memory leaks and other pernicious bugs related to memory allocation and deallocation.

Exception handling is another feature in Java that makes for more robust programs. An *exception* is a signal that some sort of exceptional condition, such as a "file not found" error, has occurred. Using the `try`/`catch`/`finally` statement, you can group all of your error handling code in one place, which greatly simplifies the task of error handling and recovery.

Secure

One of the most highly touted aspects of Java is that it's a *secure* language. This is especially important because of the distributed nature of Java. Without an assurance of security, you certainly wouldn't want to download code from a random site on the Internet and let it run on your computer. Yet this is exactly what people do with Java applets every day. Java was designed with security in mind, and provides several layers of security controls that protect against malicious code, and allow users to comfortably run untrusted programs such as applets.

At the lowest level, security goes hand-in-hand with robustness. As we've already seen, Java programs cannot forge pointers to memory, or overflow arrays, or read memory outside of the bounds of an array or string. These features are one of Java's main defenses against malicious code. By totally disallowing any direct access to memory, an entire huge, messy class of security attacks is ruled out.

The second line of defense against malicious code is the byte-code verification process that the Java interpreter performs on any untrusted code it loads. These verification steps ensure that the code is well-formed--that it doesn't overflow or underflow the stack or contain illegal byte-codes, for example. If the byte-code verification step was skipped, inadvertently corrupted or maliciously crafted byte-codes might be able to take advantage of implementation weaknesses in a Java interpreter.

Another layer of security protection is commonly referred to as the "sandbox model": untrusted code is placed in a "sandbox," where it can play safely, without doing any damage to the "real world," or full Java environment. When an applet, or other untrusted code, is running in the sandbox, there are a number of restrictions on what it can do. The most obvious of these restrictions is that it has no access whatsoever to the local file system. There are a number of other restrictions in the sandbox as well. These restrictions are enforced by a `SecurityManager` class. The model works because all of the core Java classes that perform sensitive operations, such as filesystem access, first ask permission of the currently installed `SecurityManager`. If the call is being made, directly or indirectly, by untrusted code, the security manager throws an exception, and the operation is not permitted. See Chapter 6, *Applets* for a complete list of the restrictions placed on applets running in the sandbox.

Finally, in Java 1.1, there is another possible solution to the problem of security. By attaching a digital signature to Java code, the origin of that code can be established in a cryptographically secure and unforgeable way. If you have specified that you trust a person or organization, then code that bears the digital signature of that trusted entity is trusted, even when loaded over the network, and may be run without the restrictions of the sandbox model.

Of course, security isn't a black-and-white thing. Just as a program can never be guaranteed to be 100% bug-free, no language or environment can be guaranteed 100% secure. With that said, however, Java does seem to offer a practical level of security for most applications. It anticipates and defends against most of the techniques that have historically been used to trick software into misbehaving, and it has been intensely scrutinized by security experts and hackers alike. Some security holes were found in early versions of Java, but these flaws were fixed almost as soon as they were found, and it seems reasonable to expect that any future holes will be fixed just as quickly.

High-Performance

Java is an interpreted language, so it is never going to be as fast as a compiled language like C. Java 1.0 was said to be about 20 times slower than C. Java 1.1 is nearly twice as fast as Java 1.0, however, so it might be reasonable to say that compiled C code runs ten times as fast as interpreted Java byte-codes. But before you throw up your arms in disgust, be aware that this speed is more than adequate to run interactive, GUI and network-based applications, where the application is often idle, waiting for the user to do something, or waiting for data from the network. Furthermore, the speed-critical sections of the Java run-time environment, that do things like string concatenation and comparison, are implemented with efficient native code.

As a further performance boost, many Java interpreters now include "just in time" compilers that can translate Java byte-codes into machine code for a particular CPU at run-time. The Java byte-code format was designed with these "just in time" compilers in mind, so the process of generating machine code is fairly efficient and it produces reasonably good code. In fact, Sun claims that the performance of byte-codes converted to machine code is nearly as good as native C or C++. If you are willing to sacrifice code portability to gain speed, you can also write portions of your program in C or C++ and use Java native methods to interface with this native code.

When you are considering performance, it's important to remember where Java falls in the spectrum of available programming languages. At one end of the spectrum, there are high-level, fully-interpreted scripting languages such as Tcl and the UNIX shells. These languages are great for prototyping and they are highly portable, but they are also very slow. At the other end of the spectrum, you have low-level compiled languages like C and C++. These languages offer high performance, but they suffer in terms of reliability and portability. Java falls in the middle of the spectrum. The performance of Java's interpreted byte-codes is much better than the high-level scripting languages (even Perl), but it still offers the simplicity and portability of those languages.

Multithreaded

In a GUI-based network application such as a Web browser, it's easy to imagine multiple things going on at the same time. A user could be listening to an audio clip while she is scrolling a page, and in the background the browser is downloading an image. Java is a *multithreaded* language; it provides support for multiple threads of execution (sometimes called lightweight processes) that can handle different tasks. An important benefit of multithreading is that it improves the interactive performance of graphical applications for the user.

If you have tried working with threads in C or C++, you know that it can be quite difficult. Java makes programming with threads much easier, by providing built-in language support for threads. The `java.lang` package provides a `Thread` class that supports methods to start and stop threads and set thread priorities, among other things. The Java language syntax also supports threads directly with the `synchronized` keyword. This keyword makes it extremely easy to mark sections of code or entire methods that should only be run by a single thread at a time.

While threads are "wizard-level" stuff in C and C++, their use is commonplace in Java. Because Java makes threads so easy to use, the Java class libraries require their use in a number of places. For example, any applet that performs animation does so with a thread. Similarly, Java does not support asynchronous, non-blocking I/O with notification through signals or interrupts--you must instead create a thread that blocks on every I/O channel you are interested in.

1.2 A Simple Example

By now you should have a pretty good idea of why Java is such an interesting language. So we'll stop talking about abstract concepts and look at some concrete Java code. Before we look at an interesting applet, however, we are going to pay tribute to that ubiquitous favorite, "Hello World."

Hello World

Example 1.1 shows the simplest possible Java program: "Hello World."

Example 1.1: Hello World

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

This program, like every Java program, consists of a public class definition. The class contains a method named `main()`, which is the main entry point for all Java applications--that is, the point at which the interpreter starts executing the program. The body of `main()` consists of only a single line, which prints out the message:

```
Hello World!
```

This program must be saved in a file with the same name as the public class plus a *.java* extension. To compile it, you would use *javac*: [1]

> [1] Assuming you're using Sun's Java Development Kit (JDK). If you're using a Java development environment from some other vendor, follow your vendor's instructions.

```
% javac HelloWorld.java
```

This command produces the *HelloWorld.class* file in the current directory. To run the program, you use the Java interpreter, *java*:

```
% java HelloWorld
```

Note that when you invoke the interpreter, you do not supply the *.class* extension for the file you want to run.

A Scribble Applet

Example 1.2 shows a less trivial Java program. This program is an applet, rather than a standalone Java application like the "Hello World" program above. Because this example is an applet, it has a different structure than a standalone application; notably, it does not have a `main()` method. Like all applets, this one runs inside an applet viewer or Web browser, and lets the user draw (or scribble) with the mouse, as illustrated in Figure 1.1.

Figure 1.1: A Java applet running in a Web browser

[Graphic: Figure 1-1]

One of the major changes between Java 1.0 and Java 1.1 is in the way that Java programs are notified of "events", such as mouse motion. Example 1.2 uses the Java 1.0 event model rather than the preferred Java 1.1 event model. This is because the current generation of Web browsers (as this is written) still use Java 1.0. In order for this applet to be widely usable, it is coded with the old, "deprecated" event model. [2]

[2] If you are interested in updating this program to use Java 1.1, see Chapter 7, *Events* for

information on how to use the new 1.1 event model. In addition, you need to change the call to `bounds()` in the `action()` method to a call to `getBounds()`, if you want to avoid a compilation warning about using a deprecated method.

Example 1.2: A Java Applet

```
import java.applet.*;
import java.awt.*;
public class Scribble extends Applet {
  private int last_x, last_y;                // Store the last mouse position.
  private Color current_color = Color.black; // Store the current color.
  private Button clear_button;               // The clear button.
  private Choice color_choices;              // The color dropdown list.
  // This method is called to initialize the applet.
  // Applets don't have a main() method.
  public void init() {
    // Set the background color.
    this.setBackground(Color.white);
    // Create a button and add it to the applet.  Set the button's colors.
    clear_button = new Button("Clear");
    clear_button.setForeground(Color.black);
    clear_button.setBackground(Color.lightGray);
    this.add(clear_button);
    // Create a menu of colors and add it to the applet.
    // Also set the menu's colors and add a label.
    color_choices = new Choice();
    color_choices.addItem("black");
    color_choices.addItem("red");
    color_choices.addItem("yellow");
    color_choices.addItem("green");
    color_choices.setForeground(Color.black);
    color_choices.setBackground(Color.lightGray);
    this.add(new Label("Color: "));
    this.add(color_choices);
  }
  // This method is called when the user clicks the mouse to start a scribble.
  public boolean mouseDown(Event e, int x, int y)
  {
    last_x = x; last_y = y;
    return true;
  }
  // This method is called when the user drags the mouse.
  public boolean mouseDrag(Event e, int x, int y)
  {
    Graphics g = this.getGraphics();
    g.setColor(current_color);
    g.drawLine(last_x, last_y, x, y);
    last_x = x;
    last_y = y;
    return true;
  }
  // This method is called when the user clicks the button or chooses a color.
  public boolean action(Event event, Object arg) {
    // If the Clear button was clicked on, handle it.
    if (event.target == clear_button) {
      Graphics g = this.getGraphics();
      Rectangle r = this.bounds();
      g.setColor(this.getBackground());
      g.fillRect(r.x, r.y, r.width, r.height);
      return true;
    }
    // Otherwise if a color was chosen, handle that.
    else if (event.target == color_choices) {
      if (arg.equals("black")) current_color = Color.black;
      else if (arg.equals("red")) current_color = Color.red;
      else if (arg.equals("yellow")) current_color = Color.yellow;
      else if (arg.equals("green")) current_color = Color.green;
      return true;
    }
    // Otherwise, let the superclass handle it.
    else return super.action(event, arg);
  }
}
```

Don't expect to be able to understand the entire applet at this point. It is here to give you the flavor of the language. In Chapter 2, *How Java Differs from C* and Chapter 3, *Classes and Objects in Java* we'll explain the language constructs you need to understand the example. Then, in Chapter 6, *Applets* and Chapter 7, *Events* we'll explain the applet and event-handling concepts used in this example.

The first thing you should notice when browsing through the code is that it looks reassuringly like C and C++. The `if` and `return` statements are familiar. Assignment of values to variables uses the expected syntax. Procedures (called "methods" in Java) are recognizable as such.

The second thing to notice is the object-oriented nature of the code. As you can see at the top of the example, the program consists of the definition of a public class. The name of the class we are defining is `Scribble`; it is an extension, or subclass, of the `Applet` class. (The full name of the `Applet` class is `java.applet.Applet`. One of the `import` statements at the top of the example allows us to refer to `Applet` by this shorter name.)

Classes are said to "encapsulate" data and methods. As you can see, our `Scribble` class contains both variable and method declarations. The methods are actually defined inside of the class. The methods of a class are often invoked through an instance of the class. Thus you see lines like:

```
color_choices.addItem("black");
```

This line of code invokes the `addItem()` method of the object referred to by the `color_choices` variable. If you're a C programmer, but not a C++ programmer, this syntax may take a little getting used to. We'll see lots more of it in Chapters 2 and 3. Note that `this` is a keyword, not a variable name. It refers to the current object; in this example, it refers to the `Scribble` object.

The `init()` method of an applet is called by the Web browser or applet viewer when it is starting the applet up. In our example, this method creates a **Clear** button and a menu of color choices, and then adds these GUI components to the applet.

The `mouseDown()` and `mouseDrag()` methods are called when the user clicks and drags the mouse. These are the methods that are responsible for drawing lines as the user scribbles. The `action()` method is invoked when the user clicks on the **Clear** button or selects a color from the menu of colors. The body of the method determines which of these two "events" has occurred and handles the event appropriately. Recall that these methods are part of the Java 1.0 event model. Chapter 7, *Events* explains this model and also explains the Java 1.1 event model that replaces it.

To compile this example, you'd save it in a file named *Scribble.java* and use *javac*:

```
% javac Scribble.java
```

This example is an applet, not a standalone program like our "Hello World" example. It does not have a `main()` method, and therefore cannot be run directly by the Java interpreter. Instead, we must reference it in an HTML file and run the applet in an applet viewer or Web browser. It is the applet viewer or Web browser that loads the applet class into its running Java interpreter and invokes the various methods of the applet at the appropriate times. To include the applet in a Web page, we'd use an HTML fragment like the following:

```
<APPLET code="Scribble.class" width=500 height=300>
</APPLET>
```

Example 1.3 shows a complete HTML file that we might use to display the applet. Chapter 15, *Java-Related HTML Tags* explains the HTML syntax for applets in full detail.

Example 1.3: An HTML File that Contains an Applet

```
<HTML>
<HEAD>
<TITLE>The Scribble Applet</TITLE>
</HEAD>
<BODY>
Please scribble away in the applet below.
<P>
<APPLET code="Scribble.class" width=500 height=300>
Your browser does not support Java, or Java is not enabled. Sorry!
</APPLET>
</BODY>
</HTML>
```

Suppose we save this example HTML file as *Scribble.html*. Then to run this applet, you could use Sun's *appletviewer* command like this:

```
% appletviewer Scribble.html
```

You could also display the applet by viewing the *Scribble.html* file in your Web browser, if your browser supports Java applets. Figure 1.1 showed the `Scribble` applet running in Netscape Navigator.

PREVIOUS
Why Is Java Interesting?

HOME
BOOK INDEX

NEXT
How Java Differs from C

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

2. How Java Differs from C

Contents:

Java is a lot like C, which makes it relatively easy for C programmers to learn. But there are a number of important differences between C and Java, such as the lack of a preprocessor, the use of 16-bit Unicode characters, and the exception handling mechanism. This chapter explains those differences, so that programmers who already know C can start programming in Java right away!

This chapter also points out similarities and differences between Java and C++. C++ programmers should beware, though: While Java borrows a lot of terminology and even syntax from C++, the analogies between Java and C++ are not nearly as strong as those between Java and C. C++ programmers should be careful not to be lulled into a false sense of familiarity with Java just because the languages share a number of keywords.

One of the main areas in which Java differs from C, of course, is that Java is an object-oriented language and has mechanisms to define classes and create objects that are instances of those classes. Java's object-oriented features are a topic for a chapter of their own, and they'll be explained in detail in Chapter 3, *Classes and Objects in Java*.

2.1 Program Structure and Environment

A program in Java consists of one or more class definitions, each of which has been compiled into its own *.class* file of Java Virtual Machine object code. One of these classes must define a method `main()`, which is where the program starts running. [1]

> [1] *Method* is an object-oriented term for a procedure or function. You'll see it used throughout this book.

To invoke a Java program, you run the Java interpreter, *java*, and specify the name of the class that contains the `main()` method. You should omit the *.class* extension when doing this. Note that a Java applet is not an application--it is a Java class that is loaded and run by an already running Java application such as a Web browser or applet viewer.

The `main()` method that the Java interpreter invokes to start a Java program must have the following prototype:

```
public static void main(String args[])
```

The Java interpreter runs until the `main()` method returns, or until the interpreter reaches the end of `main()`. If no threads have been created by the program, the interpreter exits. Otherwise, the interpreter continues running until the last thread terminates.

Command-Line Arguments

The single argument to `main()` is an array of strings, conventionally named `args` or `argv`. The length of this array (which would be passed as the `argc` argument in C) is available as `argv.length`, as is the case with any Java array. The elements of the array are the arguments, if any, that appeared on the interpreter command line after the class name. Note that the first element of the array is *not* the name of the class, as a C programmer might expect it to be. Example 2.1 shows how you could write a UNIX-style *echo* command (a program that simply prints out its arguments) in Java.

Example 2.1: An Echo Program in Java

```
public class echo {
    public static void main(String argv[]) {
        for(int i=0; i < argv.length; i++)
```

```
            System.out.print(argv[i] + " ");
        System.out.print("\n");
        System.exit(0);
    }
}
```

Program Exit Value

Note that `main()` must be declared to return `void`. Thus you cannot return a value from your Java program with a `return` statement in `main()`. If you need to return a value, call `System.exit()` with the desired integer value, as we've done in Example 2.1. Note that the handling and interpretation of this exit value are, of course, operating-system dependent. `System.exit()` causes the Java interpreter to exit immediately, whether or not other threads are running.

Environment

The Java API does not allow a Java program to read operating system environment variables because they are platform-dependent. However, Java defines a similar, platform-independent mechanism, known as the system properties list, for associating textual values with names.

A Java program can look up the value of a named property with the `System.getProperty()` method:

```
String homedir = System.getProperty("user.home");
String debug = System.getProperty("myapp.debug");
```

The Java interpreter automatically defines a number of standard system properties when it starts up. You can insert additional property definitions into the list by specifying the `-D` option to the interpreter:

```
% java -Dmyapp.debug=true myapp
```

See Chapter 14, *System Properties* for more information on system properties.

PREVIOUS
A Simple Example

HOME
BOOK INDEX

NEXT
The Name Space: Packages, Classes, and Members

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

2.2 The Name Space: Packages, Classes, and Members

As a language that is designed to support dynamic loading of modules over the entire Internet, Java takes special care to avoid name space conflicts. Global variables are simply not part of the language. Neither are "global" functions or procedures, for that matter.

No Global Variables

In Java, every field and method is declared within a class and forms part of that class. Also, every class is part of a *package* (in Java 1.1, classes can also be declared within other classes). The fields and methods (and classes in 1.1) of a class are known as the *members* of a class. Every Java field or method may be referred to by its fully qualified name, which consists of the package name, the class name, and the member name (i.e., the field or the method name), all separated by periods. Package names are themselves usually composed of multiple period-separated components. Thus, the fully qualified name for a method might be:

```
david.games.tetris.SoundEffects.play()
```

Java Filenames and Directory Structure

A file of Java source code has the extension *.java*. It consists of an optional `package` statement followed by any number of `import` statements followed by one or more class or interface definitions. (The `package` and `import` statements will be introduced shortly.) If more than one class or interface is defined in a Java source file, only one of them may be declared `public` (i.e., made available outside of the package), and the source file must have the same name as that public class or interface, plus the *.java* extension.

Each class or interface definition in a *.java* file is compiled into a separate file. These files of compiled Java byte-codes are known as "class files," and must have the same name as the class or interface they define, with the extension *.class* appended. For example, the class `SoundEffects` would be stored in the file *SoundEffects.class*.

Class files are stored in a directory that has the same components as the package name. If the fully qualified name of a class is `david.games.tetris.SoundEffects`, for example, the full path of the class file must be *david/games/tetris/SoundEffects.class*. This filename is interpreted relative to the Java "class path," described below. [2]

> [2] We'll use UNIX-style directory specifications in this book. If you are a Windows programmer, simply change all the forward slashes in filenames to backward slashes. Similarly, in path specifications, change colons to semicolons.

Packages of the Java API

The Java 1.1 API consists of the classes and interfaces defined in the twenty-three packages listed in Table 2.1.

Table 2.1: The Packages of the Java API

Package name	**Contents**
java.applet	Applet classes
java.awt	Graphics, window, and GUI classes
java.awt.datatransfer	Data transfer (e.g., cut-and-paste) classes
java.awt.event	Event processing classes and interfaces
java.awt.image	Image processing classes
java.awt.peer	GUI interfaces for platform independence
java.beans	JavaBeans component model API
java.io	Various types of input and output classes
java.lang	Core language classes
java.lang.reflect	Reflection API classes
java.math	Arbitrary precision arithmetic
java.net	Networking classes
java.rmi	Remote Method Invocation classes
java.rmi.dgc	RMI-related classes
java.rmi.registry	RMI-related classes
java.rmi.server	RMI-related classes

java.security	Security classes
java.security.acl	Security-related classes
java.security.interfaces	Security-related classes
java.sql	JDBC SQL API for database access
java.text	Internationalization classes
java.util	Various useful data types
java.util.zip	Compression and decompression classes

The Java Class Path

The Java interpreter knows where its standard system classes are installed, and loads them from that location as needed. By default, it looks up user-defined classes in or relative to the current directory. You can set the `CLASSPATH` environment variable to tell the interpreter where to look for user-defined classes. The interpreter always appends the location of its system classes to the end of the path specified by this environment variable. The entries in a class path specification should be directories or ZIP files that contain the classes. The directories in a class path specification should be colon-separated on a UNIX system, and semicolon-separated on a Windows system. For example, on a UNIX system, you might use:

```
setenv CLASSPATH .:/home/david/classes:/usr/local/javatools/classes.zip
```

On a Windows system you could use:

```
setenv CLASSPATH .;C:\david\classes;D:\local\javatools\classes.zip
```

This tells Java to search in and beneath the specified directories for non-system classes. Note that the current directory (`.`) is included in these paths.

You can also specify a class path to the Java interpreter with the `-classpath` command-line argument. Setting this option overides any path specified in the `CLASSPATH` environment variable. Note that the interpreter does not append the location of the system classes to the end of this path, so you must be sure to specify those system classes yourself. Finally, note that the Java compiler also recognizes and honors class paths specified with the `CLASSPATH` environment variable and the `-classpath` command-line argument.

Globally Unique Package Names

The Java designers have proposed an Internet-wide unique package naming scheme that is based on the Internet domain name of the organization at which the package is developed.

Figure 2.1 shows some fully qualified names, which include package, class, and field components.

Figure 2.1: Fully qualified names in Java

[Graphic: Figure 2-1]

Some organizations are following this naming scheme, and producing classes with names like `com.sybase.jdbc.SybDriver`. Another trend that is developing, however, is for companies to simply use their company name as the first component of their package names, and produce classes like `netscape.javascript.JSObject`.

The top-level package names `java` and `sun` are reserved for use by Sun, of course. Developers should not define new classes within these packages.

The package Statement

The `package` statement must appear as the first statement (i.e., the first text other than comments and whitespace) in a file of Java source code, if it appears at all. It specifies which package the code in the file is part of. Java code that is part of a particular package has access to all classes (`public` and non-`public`) in the package, and to all non-`private` methods and fields in all those classes. When Java code is part of a named package, the compiled class file must be placed at the appropriate position in the `CLASSPATH` directory hierarchy before it can be accessed by the Java interpreter or other utilities.

If the `package` statement is omitted from a file, the code in that file is part of an unnamed default package. This is convenient for small test programs, or during development, because it means that the code can be interpreted from the current directory.

The import Statement

The `import` statement makes Java classes available to the current class under an abbreviated name. Public Java classes are always available by their fully qualified names, assuming that the appropriate class file can be found (and is readable) relative to the `CLASSPATH` environment variable. `import` doesn't actually make the class available or "read it in"; it simply saves you typing and makes your code more legible.

Any number of `import` statements may appear in a Java program. They must appear, however, after the optional `package` statement at the top of the file, and before the first class or interface definition in the file.

There are two forms of the `import` statement:

```
import package.class ;
import package.* ;
```

The first form allows the specified class in the specified package to be known by its class name alone. Thus, this `import` statement allows you to type `Hashtable` instead of `java.util.Hashtable`:

```
import java.util.Hashtable;
```

The second form of the `import` statement makes all classes in a package available by their class name. For example, the following `import` statement is implicit (you need not specify it yourself) in every Java program:

```
import java.lang.*;
```

It makes the core classes of the language available by their unqualified class names. If two packages imported with this form of the statement contain classes with the same name, it is an error to use either of those ambiguous classes without using its fully qualified name.

Access to Packages, Classes, and Class Members

Java has the following rules about access to packages, classes, and class members. (Class members are the variables, methods, and, in Java 1.1, nested classes defined within a class). Note that the `public`, `private`, and `protected` keywords used in these rules will be explained in more detail in the next chapter.

- A package is accessible if the appropriate files and directories are accessible (e.g., if local files have appropriate read permissions, or if they can be downloaded via the network).

- All classes and interfaces in a package are accessible to all other classes and interfaces in the same package. It is not possible to define classes in Java that are visible only within a single file of source code.
- A class declared `public` in one package is accessible within another package, assuming that the package itself is accessible. A non-`public` class is not accessible outside of its package.
- Members of a class are accessible from a different class within the same package, as long as they are not declared `private`. `private` members are accessible only within their own class.
- Af member of a class `A` is accessible from a class `B` in a different package if `A` is `public` and the member is `public`, or if `A` is `public`, the member is `protected`, and `B` is a subclass of `A`.
- All members of a class are always accessible from within that class.

Local Variables

The name space rules we've been describing apply to packages, classes, and the members within classes. Java also supports local variables, declared within method definitions. These local variables behave just like local variables in C--they do not have globally unique hierarchical names, nor do they have access modifiers like `public` and `private`. Local variables are quite different from class fields.

PREVIOUS
Program Structure and Environment

HOME
BOOK INDEX

NEXT
Comments

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

2.3 Comments

Java supports three types of comments:

- A standard C-style comment that begins with `/*` and continues until the next `*/`. As in most implementations of C, this style of comment cannot be nested.
- A C++-style comment that begins with `//` and continues until the end of the line.
- A special "doc comment" that begins with `/**` and continues until the next `*/`. These comments may not be nested. Doc comments are specially processed by the *javadoc* program to produce simple online documentation from the Java source code. See Chapter 13, *Java Syntax* for more information on the doc comment syntax, and Chapter 16, *JDK Tools* for more information on the *javadoc* program.

Since C-style comments do not nest, it is a good idea to use C++-style `//` comments for most of your short comments within method bodies. This allows you to use `/* */` comments to comment out large blocks of code when you need to do that during development. This is especially important because, as you will see, Java does not support a preprocessor that allows you to use `#if 0` to comment out a block.

2.4 No Preprocessor

Java does not include any kind of preprocessor like the C *cpp* preprocessor. It may seem hard to imagine programming without `#define`, `#include`, and `#ifdef`, but in fact, Java really does not require these constructs.

Defining Constants

Any variable declared `final` in Java is a constant--its value must be specified with an initializer when it is declared, and that value may never be changed. The Java equivalent of a C `#define`'ed constant is a `static final` variable declared within a class definition. If the compiler can compute the value of such a `static final` variable at compile-time, it uses the computed value to pre-compute other compile-time constants that refer to the value. The variable `java.lang.Math.PI` is an example of such a constant. It is declared like this:

```
public final class Math {
    ...
    public static final double PI = 3.14159.....;
    ...
}
```

Note two things about this example. First, the C convention of using CAPITAL letters for constants is also a Java convention. Second, note the advantage Java constants have over C preprocessor constants: Java constants have globally unique hierarchic names, while constants defined with the C preprocessor always run the risk of a name collision. Also, Java constants are strongly typed and allow better type-checking by the compiler than C preprocessor constants.

Defining Macros

The C preprocessor allows you to define macros--a construct that looks like a function invocation but that is actually replaced directly with C code, saving the overhead of a function call. Java has no equivalent to this sort of macro, but compiler technology has advanced to a point where macros are rarely necessary any more. A good Java compiler should automatically be able to "inline" short Java methods where appropriate.

Including Files

Java does not have a `#include` directive, but it does not need one. Java defines a mapping of fully qualified class names (like `java.lang.Math`) to a directory and file structure (like *java/lang/Math.class*). This means that when the Java compiler needs to read in a specified class file, it knows exactly where to find it and does not need a special directive to tell it where to look.

Furthermore, Java does not make the distinction between *declaring* a variable or procedure and *defining* it that C does. This means that there is no need for C-style header files or function prototypes--a single Java object file serves as the interface definition and implementation for a class.

Java does have an `import` statement, which is superficially similar to the C preprocessor `#include` directive. What this statement does, however, is tell the compiler that the current file is using the specified classes, or classes from the specified package, and allows us to refer to those classes with abbreviated names. For example, since the compiler implicitly `import`s all the classes of the `java.lang` package, we can refer to the constant `java.lang.Math.PI` by the shorter name `Math.PI`.

Conditional Compilation

Java does not have any form of the C `#ifdef` or `#if` directives to perform conditional compilation. In theory, conditional compilation is not necessary in Java since it is a platform-independent language, and thus there are no platform dependencies that require the technique. In practice, however, conditional compilation is still often useful in Java--to provide slightly different user interfaces on different platforms, for example, or to support optional inclusion of debugging code in programs.

While Java does not define explicit constructs for conditional compilation, a good Java compiler (such as Sun's *javac*) performs conditional compilation implicitly--that is, it does not compile code if it can prove that the code will never be executed. Generally, this means that code within an `if` statement testing an expression that is always `false` is not included. Thus, placing code within an `if (false)` block is equivalent to surrounding it with `#if 0` and `#endif` in C.

Conditional compilation also works with constants, which, as we saw above, are `static final` variables. A class might define the constant like this:

```
private static final boolean DEBUG = false;
```

With such a constant defined, any code within an `if (DEBUG)` block is not actually compiled into the class file. To activate debugging for the class, it is only necessary to change the value of the constant to `true` and recompile the class.

PREVIOUS
Comments

HOME
BOOK INDEX

NEXT
Unicode and Character Escapes

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

2.5 Unicode and Character Escapes

Java characters, strings, and identifiers (e.g., variable, method, and class names) are composed of 16-bit Unicode characters. This makes Java programs relatively easy to internationalize for non-English-speaking users. It also makes the language easier to work with for non-English-speaking programmers--a Thai programmer could use the Thai alphabet for class and method names in her Java code.

If two-byte characters seem confusing or intimidating to you, fear not. The Unicode character set is compatible with ASCII and the first 256 characters (0x0000 to 0x00FF) are identical to the ISO8859-1 (Latin-1) characters 0x00 to 0xFF. Furthermore, the Java language design and the Java `String` API make the character representation entirely transparent to you. If you are using only Latin-1 characters, there is no way that you can even distinguish a Java 16-bit character from the 8-bit characters you are familiar with. For more information on Unicode, see Chapter 11, *Internationalization*.

Most platforms cannot display all 38,885 currently defined Unicode characters, so Java programs may be written (and Java output may appear) with special Unicode escape sequences. Anywhere within a Java program (not only within character and string literals), a Unicode character may be represented with the Unicode escape sequence `\u`*`xxxx`*, where *`xxxx`* is a sequence of four hexadecimal digits.

Java also supports all of the standard C character escape sequences, such as `\n`, `\t`, and `\`*`xxx`* (where `\`*`xxx`*is three octal digits). Note, however, that Java does not support line continuation with `\` at the end of a line. Long strings must either be specified on a single long line, or they must be created from shorter strings using the string concatenation (`+`) operator. (Note that the concatenation of two constant strings is done at compile-time rather than at run-time, so using the `+` operator in this way is not inefficient.)

There are two important differences between Unicode escapes and C-style escape characters. First, as we've noted, Unicode escapes can appear anywhere within a Java program, while the other escape characters can appear only in character and string constants.

The second, and more subtle, difference is that Unicode `\u` escape sequences are processed before the other escape characters, and thus the two types of escape sequences can have very different semantics. A

Unicode escape is simply an alternative way to represent a character that may not be displayable on certain (non-Unicode) systems. Some of the character escapes, however, represent special characters in a way that prevents the usual interpretation of those characters by the compiler. The following examples make this difference clear. Note that `\u0022` and `\u005c` are the Unicode escapes for the double-quote character and the backslash character.

```
// \" represents a " character, and prevents the normal
// interpretation of that character by the compiler.
// This is a string consisting of a double-quote character.
String quote = "\"";
// We can't represent the same string with a single Unicode escape.
// \u0022 has exactly the same meaning to the compiler as ".
// The string below turns into """: an empty string followed
// by an unterminated string, which yields a compilation error.
String quote = "\u0022";
// Here we represent both characters of an \" escape as
// Unicode escapes. This turns into "\"", and is the same
// string as in our first example.
String quote = "\u005c\u0022";
```

2.6 Primitive Data Types

Java adds `byte` and `boolean` primitive types to the standard set of C types. In addition, it strictly defines the size and signedness of its types. In C, an `int` may be 16, 32, or 64 bits, and a `char` may act signed or unsigned depending on the platform. Not so in Java. In C, an uninitialized local variable usually has garbage as its value. In Java, all variables have guaranteed default values, though the compiler may warn you in places where you rely, accidentally or not, on these default values. Table 2.2 lists Java's primitive data types. The subsections below provide details about these types.

Table 2.2: Java Primitive Data Types

				Min Value
Type	**Contains**	**Default**	**Size**	**Max Value**
boolean	`true` or `false`	`false`	1 bit	N.A.
				N.A.
char	Unicode character	`\u0000`	16 bits	`\u0000`
				`\uFFFF`
byte	signed integer	0	8 bits	-128
				127
short	signed integer	0	16 bits	-32768
				32767
int	signed integer	0	32 bits	-2147483648
				2147483647
long	signed integer	0	64 bits	-9223372036854775808
				9223372036854775807
float	IEEE 754	0.0	32 bits	+/-3.40282347E+38
	floating-point			+/-1.40239846E-45

double	IEEE 754	0.0	64 bits	+/-1.79769313486231570E+308
	floating-point			+/-4.94065645841246544E-324

The boolean Type

`boolean` values are not integers, may not be treated as integers, and may never be cast to or from any other type. To perform C-style conversions between a `boolean` value `b` and an `int i`, use the following code:

```
b = (i != 0);   // integer-to-boolean: non-0 -> true; 0 -> false;
i = (b)?1:0;    // boolean-to-integer: true -> 1; false -> 0;
```

The char Type

`char` values represent characters. Character literals may appear in a Java program between single quotes. For example:

```
char c = 'A';
```

All of the standard C character escapes, as well as Unicode escapes, are also supported in character literals. For example:

```
char newline = '\n', apostrophe = '\", delete = '\377', aleph='\u05D0';
```

Values of type `char` do not have a sign. If a `char` is cast to a `byte` or a `short`, a negative value may result.

The `char` type in Java holds a two-byte Unicode character. While this may seem intimidating to those not familiar with Unicode and the techniques of program internationalization, it is in fact totally transparent. Java does not provide a way to compute the size of a variable, nor does it allow any sort of pointer arithmetic. What this means is that if you are only using ASCII or Latin-1 characters, there is no way to distinguish a Java `char` from a C `char`.

Integral Types

The integral types in Java are `byte`, `short`, `char`, `int`, and `long`. Literals for these types are written just as they are in C. All integral types, other than `char`, are signed. There is no `unsigned` keyword as there is in C. It is not legal to write `long int` or `short int` as it is in C. A `long` constant may be distinguished from other integral constants by appending the character `l` or `L` to it.

Integer division by zero or modulo zero causes an `ArithmeticException` to be thrown. [3]

[3] Exceptions signal errors in Java. Exception handling is described later in this chapter.

Floating-Point Types

The floating-point types in Java are `float` and `double`. Literals for these types are written just as they are in C. Literals may be specified to be of type `float` by appending an `f` or `F` to the value; they may be specified to be of type `double` by appending a `d` or `D`.

`float` and `double` types have special values that may be the result of certain floating-point operations: positive infinity, negative infinity, negative zero and not-a-number. The `java.lang.Float` and `java.lang.Double` classes define some of these values as constants: `POSITIVE_INFINITY`, `NEGATIVE_INFINITY`, and `NaN`.

`NaN` is unordered--comparing it to any other number, including itself, yields `false`. Use `Float.isNaN()` or `Double.isNaN()` to test for `NaN`. Negative zero compares equal to regular zero (positive zero), but the two zeros may be distinguished by division: one divided by negative zero yields negative infinity; one divided by positive zero yields positive infinity.

Floating-point arithmetic never causes exceptions, even in the case of division by zero.

String Literals

Strings in Java are not a primitive type, but are instances of the `String` class. However, because they are so commonly used, string literals may appear between quotes in Java programs, just as they do in C. When the compiler encounters such a string literal, it automatically creates the necessary `String` object.

2.7 Reference Data Types

The non-primitive data types in Java are objects and arrays. These non-primitive types are often called "reference types" because they are handled "by reference"--in other words, the address of the object or array is stored in a variable, passed to methods, and so on. By comparison, primitive types are handled "by value"--the actual primitive values are stored in variables and passed to methods.

In C, you can manipulate a value by reference by taking its address with the & operator, and you can "dereference" an address with the `*` and `->` operators. These operators do not exist in Java: primitive types are always passed by value; arrays and objects are always passed by reference.

Because objects are passed by reference, two different variables may refer to the same object:

```
Button p, q;
p = new Button();           // p refers to a Button object.
q = p;                      // q refers to the same Button.
p.setLabel("Ok");           // A change to the object through p...
String s = q.getLabel();    // ...is also visible through q.
                            // s now contains "Ok."
```

This is not true of primitive types, however:

```
int i = 3;                  // i contains the value 3.
int j = i;                  // j contains a copy of the value in i.
i = 2;                      // Changing i doesn't change j.
                            // Now, i == 2 and j == 3.
```

Terminology: Pass by Reference

The statement that Java manipulates objects "by reference" causes confusion for some programmers, because there are several different meanings of "by reference" in common use. Regardless of what we call it, it is important to understand what Java does. Java works with references to objects. A Java variable holds only a reference to an object, not the object itself. When an object is passed to a method, only a reference to the object is actually passed, not the entire object. It is in this sense that Java manipulates objects "by reference."

Some people use the term "pass by reference" to mean that a reference to a variable is passed to a method. Java does not do this. For example, it is *not* possible to write a working `swap()` function like the following in Java:

```
public void swap(Object a, Object b) {
  Object temp = a;
  a = b;
  b = temp;
}
```

The method parameters `a` and `b` contain references to objects, not addresses of variables. Thus, while this `swap()` function does compile and run, it has no effect except on its own local variables and arguments.

To solve this terminology problem, perhaps we should say that Java manipulates objects "by reference," but it passes object references to methods "by value."

Copying Objects

Because reference types are not passed by value, assigning one object to another in Java does not copy the value of the object. It merely assigns a reference to the object. Consider the following code:

```
Button a = new Button("Okay");
Button b = new Button("Cancel");
a = b;
```

After these lines are executed, the variable `a` contains a reference to the object that `b` refers to. The object that `a` used to refer to is lost.

To copy the data of one object into another object, use the `clone()` method:

```
Vector b = new Vector;
c = b.clone();
```

After these lines run, the variable `c` refers to an object that is a duplicate of the object referred to by `b`. Note that not all types support the `clone()` method. Only classes that implement the `Cloneable` interface may be cloned. For more information on cloning objects, look up `java.lang.Cloneable` and `java.lang.Object.clone()` in Chapter 25, *The java.lang Package*.

Arrays are also reference types, and assigning an array simply copies a reference to the array. To actually copy the values stored in an array, you must assign each of the values individually or use the `System.arraycopy()` method.

Checking Objects for Equality

Another implication of passing objects by reference is that the `==` operator tests whether two variables refer to the same object, not whether two objects contain the same values. To actually test whether two separate objects are the same, you must use a specially written method for that object type (just as you might use `strcmp()` to compare C strings for equality). In Java, a number of classes define an `equals()` method that you can use to perform this test.

Java Has No Pointers

The referencing and dereferencing of objects is handled for you automatically by Java. Java does not allow you to manipulate pointers or memory addresses of any kind:

- It does not allow you to cast object or array references into integers or vice-versa.
- It does not allow you to do pointer arithmetic.
- It does not allow you to compute the size in bytes of any primitive type or object.

There are two reasons for these restrictions:

- Pointers are a notorious source of bugs. Eliminating them simplifies the language and eliminates many potential bugs.
- Pointers and pointer arithmetic could be used to sidestep Java's run-time checks and security mechanisms. Removing pointers allows Java to provide the security guarantees that it does.

To a C programmer, the lack of pointers and pointer arithmetic may seem an odious restriction in Java. But once you get used to the Java object-oriented programming model, it no longer seems like a serious restriction at all. The lack of pointers does mean that you probably can't do things like write UNIX device drivers in Java (at least not without using native methods written in C). But big deal--most of us never have to do this kind of low-level programming anyway.

null

The default value for variables of all reference types is `null`. `null` is a reserved value that indicates "an absence of reference"--i.e., that a variable does not refer to any object or array.

In Java, `null` is a reserved keyword, unlike `NULL` in C, where it is just a constant defined to be 0. `null` is an exception to the strong typing rules of Java--it may be assigned to any variable of reference type (i.e., any variable which has a class, interface, or array as its type).

`null` can't be cast to any primitive type, including integral types and `boolean`. It shouldn't be considered equal to zero (although it may be implemented this way).

Reference Type Summary

The distinction between primitive types passed by value, and objects and arrays passed by reference is a crucial one in Java. Be sure you understand the following:

- All objects and arrays are handled by reference in Java. (Those object references are passed-by-value to methods, however.)
- The = and == operators assign and test references to objects. Use `clone()` and `equals()` to actually copy or test the objects themselves.
- The necessary referencing and dereferencing of objects and arrays is handled automatically by Java.
- A reference type can never be cast to a primitive type.
- A primitive type can never be cast to a reference type.
- There is no pointer arithmetic in Java.
- There is no `sizeof` operator in Java.
- `null` is a special value that means "no object" or indicates an absence of reference.

PREVIOUS HOME NEXT
Primitive Data Types BOOK INDEX Objects

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

2.8 Objects

Now that you know objects are passed by reference, we should discuss how they are created, used, and destroyed. The following subsections provide a very brief overview of objects. Chapter 3, *Classes and Objects in Java* explains classes and objects in much greater detail.

Creating Objects

Declaring a variable to hold an object does not create the object itself; the variable only holds the reference to the object. To actually create an object, you must use the `new` keyword. This is followed by the object's class (i.e., its type) and an optional argument list in parentheses. These arguments are passed to the constructor method for the class, which serves to initialize internal fields in the new object. For example:

```
java.awt.Button b = new java.awt.Button();
ComplexNumber c = new ComplexNumber(1.0, 1.414);
```

There are actually two other ways to create an object. First, you can create a `String` object simply by enclosing characters in double quotes:

```
String s = "This is a test";
```

Because strings are used so frequently, the Java compiler provides this technique as a shortcut. Another way to create objects is by calling the `newInstance()` method of a `Class` object. This technique is generally used only when dynamically loading classes, so we won't discuss it here. In Java 1.1, objects can also be created by "de-serializing" them--i.e., recreating an object that had its state saved through "serialization."

The memory for newly created objects is dynamically allocated. Creating an object with `new` in Java is like calling `malloc()` in C to allocate memory for an instance of a `struct`. It is also, of course, a lot like using the `new` operator in C++. (Below, we'll see where this analogy to `malloc()` in C and `new` in

C++ breaks down.)

Accessing Objects

As you've probably noticed in various example code fragments by now, the way you access the fields of an object is with a dot:

```
ComplexNumber c = new ComplexNumber();
c.x = 1.0;
c.y = -1.414;
```

This syntax is reminiscent of accessing the fields of a `struct` in C. Recall, though, that Java objects are always accessed by reference, and that Java performs any necessary dereferencing for you. Thus, the dot in Java is more like `->` in C. Java hides the fact that there is a reference here in an attempt to make your programming easier. The other difference between C and Java when accessing objects is that in Java you refer to an object's methods with the same syntax used for fields:

```
ComplexNumber c = new ComplexNumber(1.0, -1.414);
double magnitude = c.magnitude();
```

Garbage Collection

Objects in Java are created with the `new` keyword, but there is no corresponding `old` or `delete` keyword or `free()` method to get rid of them when they are no longer needed. If creating an object with `new` is like calling `malloc()` in C or using `new` in C++, then it would seem that Java is full of memory leaks, because we never call `free()` or use the `delete` operator.

In fact, this isn't the case. Java uses a technique called *garbage collection* to automatically detect objects that are no longer being used (an object is no longer in use when there are no more references to it) and to free them. This means that in our programs, we never need to worry about freeing memory or destroying objects--the garbage collector takes care of that.

If you are a C or C++ programmer, it may take some getting used to to just let allocated objects go without worrying about reclaiming their memory. Once you get used to it, however, you'll begin to appreciate what a nice feature this is. We'll discuss garbage collection in more detail in the next chapter.

2.9 Arrays

Most of what we learned in the previous sections about reference types and objects applies equally well to arrays in Java:

- Arrays are manipulated by reference.
- They are dynamically created with `new`.
- They are automatically garbage collected when no longer referred to.

The following subsections explain these and other details.

Creating and Destroying Arrays

There are two ways to create arrays in Java. The first uses `new`, and specifies how large the array should be:

```
byte octet_buffer[] = new byte[1024];
Button buttons[] = new Button[10];
```

Since creating an array does not create the objects that are stored in the array, there is no constructor to call, and the argument list is omitted with this form of the `new` keyword. The elements of an array created in this way are initialized to the default value for their type. The elements of an array of `int` are initialized to `0`, for example, and the elements of an array of objects are initialized to `null`. This last point is important to remember: creating an array of objects only allocates storage for object references, not objects themselves. The objects that will be referred to by the elements of the array must be created separately.

The second way to create an array is with an initializer, which looks just like it does in C:

```
int lookup_table[] = {1, 2, 4, 8, 16, 32, 64, 128};
```

This syntax dynamically creates an array and initializes its elements to the specified values. The elements specified in an array initializer may be arbitrary expressions. This is different than in C, where they must be constant expressions.

In Java 1.1, arrays may also be created and initialized "anonymously" by combining the `new` syntax with the initializer syntax. It looks like this:

```
Menu m = createMenu("File", new String[] { "Open...", "Save", "Quit" });
```

Arrays are automatically garbage collected, just like objects are.

Multidimensional Arrays

Java also supports multidimensional arrays. These are implemented as arrays-of-arrays, as they are in C. You specify a variable as a multidimensional array type simply by appending the appropriate number of `[]` pairs after it. You allocate a multidimensional array with `new` by specifying the appropriate number of elements (between square brackets) for each dimension. For example:

```
byte TwoDimArray[][] = new byte[256][16];
```

This statement does three things:

- Declares a variable named `TwoDimArray`. This variable has type `byte[][]` (array-of-array-of-`byte`).

- Dynamically allocates an array with 256 elements. The type of this newly allocated array is `byte[][]`, so it can be assigned to the variable we declared. Each element of this new array is of type `byte[]`--a single-dimensional array of `byte`.

- Dynamically allocates 256 arrays of bytes, each of which holds 16 bytes, and stores a reference to these 256 `byte[]` arrays into the 256 elements of the `byte[][]` array allocated in the second step. The 16 bytes in each of these 256 arrays are initialized to their default value of `0`.

When allocating a multidimensional array, you do not have to specify the number of elements that are contained in each dimension. For example:

```
int threeD[][][] = new int[10][][];
```

This expression allocates an array that contains ten elements, each of type `int[][]`. It is a single-dimensional allocation, although when the array elements are properly initialized to meaningful values, the array will be multidimensional. The rule for this sort of array allocation is that the first *n* dimensions (where *n* is at least one) must have the number of elements specified, and these dimensions may be followed by *m* additional dimensions with no dimension size specified. The following is legal:

```
String lots_of_strings[][][][] = new String[5][3][][];
```

This is not:

```
double temperature_data[][][] = new double[100][][10];  // illegal
```

Multidimensional arrays can also be allocated and initialized with nested initializers. For example, you might declare the following multidimensional array of strings for use by the `getParameterInfo()` method of an applet:

```
String param_info[][] = {
    {"foreground", "Color",  "foreground color"},
    {"background", "Color",  "background color"},
    {"message",    "String", "the banner to display"}
};
```

Note that since Java implements multidimensional arrays as arrays-of-arrays, multidimensional arrays need not be "rectangular." For example, this is how you could create and initialize a "triangular array":

```
short triangle[][] = new short[10][];       // A single-dimensional array.
for(int i = 0; i < triangle.length; i++) {  // For each element of that array
    triangle[i] = new short[i+1];           // Allocate a new array.
    for(int j=0; j < i+1; j++)              // For each element of new array
        triangle[i][j] = (short) i + j;     // Initialize it to a value.
}
```

You can also declare and initialize non-rectangular arrays with nested initializers:

```
static int[][] twodim = {{1, 2}, {3, 4, 5}, {5, 6, 7, 8}};
```

To simulate multiple dimensions within a single-dimensional array, you'd use code just as you would in C:

```
final int rows = 600;
final int columns = 800;
byte pixels[] = new byte[rows*columns];
// access element [i,j] like this:
byte b = pixels[i + j*columns];
```

Accessing Array Elements

Array access in Java is just like array access in C--you access an element of an array by putting an integer-valued expression between square brackets after the name of the array.

```
int a[] = new int[100];
a[0] = 0;
for(int i = 1; i < a.length; i++) a[i] = i + a[i-1];
```

Notice how we computed the number of elements of the array in this example--by accessing the `length` field of the array. This is the only field that arrays support. Note that it is a constant (`final`) field--any attempt to store a value into the `length` field of an array will fail.

In all Java array references, the index is checked to make sure it is not too small (less than zero) or too big (greater than or equal to the array length). If the index is out of bounds, an `ArrayIndexOutOfBoundsException` is thrown. [4] This is another way that Java works to prevent bugs (and security problems).

[4] The discussion of exceptions and exception handling is still to come.

Are Arrays Objects?

It is useful to consider arrays to be a separate kind of reference type from objects. In some ways, though, arrays behave just like objects. As we saw, arrays use the object syntax `.length` to refer to their length. Arrays may also be assigned to variables of type `Object`, and the methods of the `Object` class may be invoked for arrays. (`Object` is the root class in Java, which means that all objects can be assigned to a variable of type `Object` and all objects can invoke the methods of `Object`.)

The evidence suggests that arrays are, in fact, objects. Java defines enough special syntax for arrays, however, that it is still most useful to consider them a different kind of reference type than objects.

Declaring Array Variables and Arguments

In C, you declare an array variable or array function argument by placing square brackets next to the variable name:

```
void reverse(char strbuf[], int buffer_size)  {
    char buffer[500];
        ...
}
```

In Java, you would have to declare `buffer` as an array variable, and then allocate the array itself with `new`, but otherwise you could use the same syntax, with the array brackets after the variable or argument name.

However, Java also allows you to put the array brackets after the type name instead. So you could rewrite this code fragment to look something like this:

```
void reverse(char[] strbuf, int buffer_size) {
    char[] buffer = new char[500];
        ...
}
```

In a lot of ways, this new array syntax is easier to read and easier to understand. (It doesn't work in C, by the way, because pointers make C's type declaration syntax a real mess.) The only problem with this new syntax is that if you get in the habit of using it, it will make it harder for you when you (hopefully only occasionally!) have to switch back and program in C.

Java even allows you to mix the declaration styles, which is something you may find occasionally useful (or frequently confusing!) for certain data structures or algorithms. For example:

```
// row and column are arrays of byte.
// matrix is an array of an array of bytes.
byte[] row, column, matrix[];
// This method takes an array of bytes and an
// array of arrays of bytes.
public void dot_product(byte[] column, byte[] matrix[]) { ... }
```

A final point to note about array declarations is that (as we've seen throughout this section) the size of an array is not part of its type as it is in C. Thus, you can declare a variable to be of type `String[]`, for example, and assign any array of `String` objects to it, regardless of the length of the array:

```
String[] strings;           // this variable can refer to any String array
strings = new String[10];   // one that contains 10 Strings
strings = new String[20];   // or one that contains 20.
```

PREVIOUS | HOME | NEXT
Objects | BOOK INDEX | Strings

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

2.10 Strings

Strings in Java are *not* null-terminated arrays of characters as they are in C. Instead, they are instances of the `java.lang.String` class. Java strings are unusual, in that the compiler treats them almost as if they were primitive types--for example, it automatically creates a `String` object when it encounters a double-quoted constant in the program. And, the language defines an operator that operates on `String` objects--the `+` operator for string concatenation.

An important feature of `String` objects is that they are immutable--i.e., there are no methods defined that allow you to change the contents of a `String`. If you need to modify the contents of a `String`, you have to create a `StringBuffer` object from the `String` object, modify the contents of the `StringBuffer`, and then create a new `String` from the contents of the `StringBuffer`.

Note that it is moot to ask whether Java strings are terminated with a NUL character (`\u0000`) or not. Java performs run-time bounds checking on all array and string accesses, so there is no way to examine the value of any internal terminator character that appears after the last character of the string.

Both the `String` and `StringBuffer` classes are documented in Chapter 25, *The java.lang Package*, and you'll find a complete set of methods for string handling and manipulation there. Some of the more important `String` methods are: `length()`, `charAt()`, `equals()`, `compareTo()`, `indexOf()`, `lastIndexOf()`, and `substring()`.

2.11 Operators

Java supports almost all of the standard C operators. These standard operators have the same precedence and associativity in Java as they do in C. They are listed in Table 2.3 and also in quick reference form in Chapter 13, *Java Syntax*.

Table 2.3: Java Operators

Prec.	Operator	Operand Type(s)	Assoc.	Operation Performed
1	++	arithmetic	R	pre-or-post increment
				(unary)
	--	arithmetic	R	pre-or-post decrement (unary)
	+, -	arithmetic	R	unary plus, unary minus
	~	integral	R	bitwise complement (unary)
	!	boolean	R	logical complement (unary)
	(*type*)	any	R	cast
2	*, /, %	arithmetic	L	multiplication, division, remainder
3	+, -	arithmetic	L	addition, subtraction
	+	string	L	string concatenation
4	<<	integral	L	left shift
	>>	integral	L	right shift with sign extension
	>>>	integral	L	right shift with zero extension
5	<, <=	arithmetic	L	less than, less than or equal
	>, >=	arithmetic	L	greater than, greater than or equal
	instanceof	object, type	L	type comparison
6	==	primitive	L	equal (have identical

				values)
	!=	primitive	L	not equal (have different values)
	==	object	L	equal (refer to same object)
	!=	object	L	not equal (refer to different objects)
7	&	integral	L	bitwise AND
	&	boolean	L	boolean AND
8	^	integral	L	bitwise XOR
	^	boolean	L	boolean XOR
9	|	integral	L	bitwise OR
	|	boolean	L	boolean OR
10	&&	boolean	L	conditional AND
11	||	boolean	L	conditional OR
12	?:	boolean, any, any	R	conditional (ternary) operator
13	=	variable, any	R	assignment
	*=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, |=	variable, any	R	assignment with operation

Note the following Java operator differences from C. Java does not support the comma operator for combining two expressions into one (although the `for` statement simulates this operator in a useful way). Since Java does not allow you to manipulate pointers directly, it does not support the reference and dereference operators `*`, `->`, and `&`, nor the `sizeof` operator. Further, Java doesn't consider `[]` (array access) and `.` (field access) to be operators, as C does.

Java also adds some new operators:

The `+` operator applied to `String` values concatenates them. [5] If only one operand of `+` is a `String`, the other one is converted to a string. The conversion is done automatically for primitive types, and by calling the `toString()` method of non-primitive types. This `String` `+` operator has the same precedence as the arithmetic `+` operator. The `+=` operator works as you would expect for `String` values.

> [5] To C++ programmers, this looks like operator overloading. In fact, Java does not support operator overloading--the language designers decided (after much debate) that overloaded operators were a neat idea, but that code that relied on them became hard to read and understand.

The `instanceof` operator returns `true` if the object `o` on its left-hand side is an instance of the class

C or implements the interface I specified on its right-hand side. It also returns true if o is an instance of a subclass of C or is an instance of a subclass of some class that implements I. instanceof returns false if o is not an instance of C or does not implement I. It also returns false if the value on its left is null. If instanceof returns true, it means that o is *assignable to* variables of type C or I. The instanceof operator has the same precedence as the <, <=, >, and >= operators.

Because all integral types in Java are signed values, the Java >> operator is defined to do a right shift with sign extension. The >>> operator treats the value to be shifted as an unsigned number and shifts the bits right with zero extension. The >>>= operator works as you would expect.

& and |

> When & and | are applied to integral types in Java, they perform the expected bitwise AND and OR operations. Java makes a strong distinction between integral types and the boolean type, however. Thus, if these operators are applied to boolean types, they perform logical AND and logical OR operations. These logical AND and logical OR operators always evaluate both of their operands, even when the result of the operation is determined after evaluating only the left operand. This is useful when the operands are expressions with side effects (such as method calls) and you always want the side effects to occur. However, when you do not want the right operand evaluated if it is not necessary, you can use the && and || operators, which perform "short-circuited" logical AND and logical OR operations just as in C. The &= and |= operators perform a bitwise or logical operation depending on the type of the operands, as you would expect.

2.12 Statements

Many of Java's control statements are similar or identical to C statements. This section lists and, where necessary, explains Java's statements. Note that the topic of exceptions and the try/catch/finally statement is substantial enough that it is covered later in a section of its own.

The if/else, while, and do/while Statements

The if, else, do, and while statements are exactly the same in Java as they are in C. The only substantial difference arises because the Java boolean type cannot be cast to other types. In Java, the values 0 and null are not the same as false, and non-zero and non-null values are not the same as true.

The conditional expression that is expected by the if, the while, and the do/while statements must be of boolean type in Java. Specifying an integer type or a reference type won't do. Thus, the following C code is not legal in Java:

```
int i = 10;
while(i--) {
    Object o = get_object();
    if (o) {
        do { ... } while(j);
    }
}
```

In Java, you must make the condition you are testing for clear by explictly testing your value against 0 or null. Use code like the following:

```
int i = 10;
while(i-- > 0) {
    Object o = get_object();
    if (o != null) {
        do { ... } while(j != 0);
    }
}
```

The switch Statement

The `switch` statement is the same in Java as it is in C. You may use `byte`, `char`, `short`, `int`, or `long` types as the values of the `case` labels, and you may also specify a `default` label just as you do in C.

The for Loop

The `for` statement is perhaps the most useful looping construct available in Java. There are only two differences between the Java `for` loop and the C `for` loop. The first difference is that although Java does not support the C comma operator (which allows multiple expressions to be joined into a single expression), the Java `for` loop simulates it by allowing multiple comma-separated expressions to appear in the initialization and increment sections, but not the test section, of the loop. For example:

```
int i;
String s;
for(i=0, s = "testing";             // Initialize variables.
    (i < 10) && (s.length() >= 1);  // Test for continuation.
    i++, s = s.substring(1))        // Increment variables.
{
    System.out.println(s);          // Loop body.
}
```

As you can see, this "difference" between the Java and C `for` loops is really a similarity.

The second difference is the addition of the C++ ability to declare local loop variables in the initialization section of the loop:

```
for(int i = 0; i < my_array.length; i++)
    System.out.println("a[" + i + "] = " + my_array[i]);
```

Variables declared in this way have the `for` loop as their scope. In other words, they are only valid within the body of the `for` loop and within the initialization, test, and increment expressions of the loop. Although variables declared in `for` loops have their own scope, the Java compiler won't let you declare a loop variable that has the same name as an already existing variable or parameter.

Note that because variable declaration syntax also uses the comma, the Java syntax allows you to either specify multiple comma-separated initialization expressions or to declare and initialize multiple comma-separated variables of the same type. You may not mix variable declarations with other, non-declaration expressions. For example, the following `for` loop declares and initializes two variables that are valid only within the `for` loop.

```
for(int i=0, j=10; i < j; i++, j--) System.out.println("k = " + i*j);
```

Labelled break and continue Statements

The `break` and `continue` statements, used alone, behave the same in Java as they do in C. However, in Java, they may optionally be followed by a label that specifies an enclosing loop (for `continue`) or any enclosing statement (for `break`). The labelled forms of these statements allow you to "break" and "continue" any specified statement or loop within a method definition, not only the nearest enclosing statements or loop.

The `break` statement, without a label, transfers control out of ("breaks out of" or terminates) the nearest enclosing `for`, `while`, `do` or `switch` statement, exactly as in C. If the `break` keyword is followed by an identifier that is the label of an arbitrary enclosing statement, execution transfers out of that enclosing statement. After the `break` statement is executed, any required `finally` clauses are executed, and control resumes at the statement following the terminated statement. (The `finally` clause and the `try` statement it is associated with are exception handling constructs and are explained in the next section.) For example:

```
test: if (check(i)) {
    try {
        for(int j=0; j < 10; j++) {
            if (j > i) break;           // Terminate just this loop.
            if (a[i][j] == null)
                break test;             // Do the finally clause and
        }                               // terminate the if statement.
    }
    finally { cleanup(a, i, j); }
}
```

Without a label, the `continue` statement works exactly as in C: It stops the iteration in progress and causes execution to resume after the last statement in the `while`, `do`, or `for` loop, just before the loop iteration is to begin again. If the `continue` keyword is followed by an identifier that is the label of an enclosing loop, execution skips to the end of that loop instead. If there are any `finally` clauses between the `continue` statement and the end of the appropriate loop, these clauses are executed before control is transferred to the end of the loop.

The following code fragment illustrates how the `continue` statement works in its labelled and unlabelled forms.

```
big_loop: while(!done) {
    if (test(a,b) == 0) continue;       // Control goes to point 2.
    try {
        for(int i=0; i < 10; i++) {
            if (a[i] == null)
                continue;               // Control goes to point 1.
            else if (b[i] == null)
                continue big_loop;      // Control goes to point 2,
                                        // after executing the finally block.
            doit(a[i],b[i]);
           // Point 1.  Increment and start loop again with the test.
        }
    }
    finally { cleanup(a,b); }
    // Point 2.  Start loop again with the (!done) test.
}
```

Note the non-intuitive feature of the labelled `continue` statement: The loop label must appear at the top of the loop, but `continue` causes execution to transfer to the very bottom of the loop.

No goto Statement

`goto` is a reserved word in Java, but the `goto` statement is not currently part of the language. Labelled `break` and `continue` statements replace some important and legitimate uses of `goto`, and the `try`/`catch`/`finally` statement replaces the others.

The synchronized Statement

Since Java is a multithreaded system, care must often be taken to prevent multiple threads from modifying objects simultaneously in a way that might leave the object's state corrupted. Sections of code that must not be executed simultaneously are known as "critical sections." Java provides the `synchronized` statement to protect these critical sections. The syntax is:

```
synchronized (expression) statement
```

`expression` is an expression that must resolve to an object or an array. The *`statement`* is the code of the critical section, which is usually a block of statements (within `{` and `}`). The *`synchronized`* statement attempts to acquire an exclusive lock for the object or array specified by `expression`. It does not execute the critical section of code until it can obtain this lock, and in this way, ensures that no other threads can be executing the section at the same time.

Note that you do not have to use the `synchronized` statement unless your program creates multiple threads that share data. If only one thread ever accesses a data structure, there is no need to protect it with `synchronized`. When you do have to use it, it might be in code like the following:

```
public static void SortIntArray(int[] a) {
    // Sort the array a. This is synchronized so that some other
    // thread can't change elements of the array while we're sorting it.
    // At least not other threads that protect their changes to the
    // array with synchronized.
    synchronized (a) {
        // Do the array sort here.
    }
}
```

The `synchronized` keyword is more often used as a method modifier in Java. When applied to a method, it indicates that the entire method is a critical section. For a `synchronized` class method (a static method), Java obtains an exclusive lock on the class before executing the method. For a `synchronized` instance method, Java obtains an exclusive lock on the class instance. (Class methods and instance methods are discussed in the next chapter.)

The package and import Statements

The `package` statement, as we saw earlier in the chapter, specifies the package that the classes in a file of Java source code are part of. If it appears, it must be the first statement of a Java file. The `import` statement, which we also saw earlier, allows us to refer to classes by abbreviated names. `import` statements must appear after the `package` statement, if any, and before any other statements in a Java file. For example:

```
package games.tetris;
import java.applet.*;
import java.awt.*;
```

PREVIOUS | HOME | NEXT
Operators | BOOK INDEX | Exceptions and Exception Handling

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

2.13 Exceptions and Exception Handling

Exception handing is a significant new feature of Java. [6] There are a number of new terms associated with exception handling. First, an *exception* is a signal that indicates that some sort of exceptional condition (such as an error) has occurred. To *throw* an exception is to signal an exceptional condition. To *catch* an exception is to handle it--to take whatever actions are necessary to recover from it.

[6] It is similar to, but not quite the same as, exception handling in C++.

Exceptions propagate up through the lexical block structure of a Java method, and then up the method call stack. If an exception is not caught by the block of code that throws it, it propagates to the next higher enclosing block of code. If it is not caught there, it propagates up again. If it is not caught anywhere in the method, it propagates to the invoking method, where it again propagates through the block structure. If an exception is never caught, it propagates all the way to the `main()` method from which the program started, and causes the Java interpreter to print an error message and a stack trace and exit.

As we'll see in the subsections below, exceptions make error handling (and "exceptional condition" handling) more regular and logical by allowing you to group all your exception handling code into one place. Instead of worrying about all of the things that can go wrong with each line of your code, you can concentrate on the algorithm at hand and place all your error handling code (that is, your exception catching code) in a single place.

Exception Objects

An exception in Java is an object that is an instance of some subclass of `java.lang.Throwable`. `Throwable` has two standard subclasses: `java.lang.Error` and `java.lang.Exception`. [7] Exceptions that are subclasses of `Error` generally indicate linkage problems related to dynamic loading, or virtual machine problems such as running out of memory. They should almost always be considered unrecoverable, and should not be caught. While the distinction is not always clear, exceptions that are subclasses of `Exception` indicate conditions that may be caught and recovered from. They include such exceptions as `java.io.EOFException`, which signals the end of a file, and `java.lang.ArrayAccessOutOfBounds`, which indicates that a program has tried to read past the end of an array.

[7] We'll use the term "exception" to refer to any subclass of `Throwable`, whether it is actually an `Exception` or an `Error`.

Since exceptions are objects, they can contain data and define methods. The `Throwable` object, at the top of the exception class hierarchy, includes a `String` message in its definition and this field is inherited by all exception classes. This field is used to store a human-readable error message that describes the exceptional condition. It is set when the exception object is created by passing an argument to the constructor method. The message can be read from the exception with the `Throwable.getMessage()` method. Most exceptions contain only this single message, but a few add other data. The `java.io.InterruptedIOException`, for example, adds the following field:

```
public int bytesTransferred;
```

This field specifies how much of the I/O was complete before the exceptional condition occurred.

Exception Handling

The `try`/`catch`/`finally` statement is Java's exception handling mechanism. `try` establishes a block of code that is to have its exceptions and abnormal exits (through `break`, `continue`, `return`, or exception propagation) handled. The `try` block is followed by zero or more `catch` clauses that catch and handle specified types of exceptions. The `catch` clauses are optionally followed by a `finally` block that contains "clean-up" code. The statements of a `finally` block are guaranteed to be executed, regardless of how the code in the `try` block exits. A detailed example of the `try`/`catch`/`finally` syntax is shown in Example 2.2.

Example 2.2: The try/catch/finally Statement

```
try {
        // Normally this code runs from the top of the block to the bottom
        // without problems.  But it sometimes may raise exceptions or
        // exit the block via a break, continue, or return statement.
}
catch (SomeException e1) {
        // Handle an exception object e1 of type SomeException
        // or of a subclass of that type.
}
catch (AnotherException e2) {
        // Handle an exception object e2 of type AnotherException
        // or of a subclass of that type.
}
finally {
        // Always execute this code, after we leave the try clause,
        // regardless of whether we leave it:
        //    1) Normally, after reaching the bottom of the block.
        //    2) With an exception that is handled by a catch.
        //    3) With an exception that is not handled.
        //    4) Because of a break, continue, or return statement.
}
```

try

The try clause simply establishes a block of code that is to have its exceptions and abnormal exits (through break, continue, return, or exception propagation) handled. The try clause by itself doesn't do anything interesting; it is the catch and finally clauses that do the exception handling and clean-up operations.

catch

A try block may be followed by zero or more catch clauses that specify code to handle various types of exceptions. catch clauses have an unusual syntax: each is declared with an argument, much like a method argument. This argument must be of type Throwable or a subclass. When an exception occurs, the first catch clause that has an argument of the appropriate type is invoked. The type of the argument must match the type of the exception object, or it must be a superclass of the exception. This catch argument is valid only within the catch block, and refers to the actual exception object that was thrown.

The code within a catch block should take whatever action is necessary to cope with the exceptional condition. If the exception was a java.io.FileNotFoundException exception, for example, you might handle it by asking the user to check his or her spelling and try again. Note that it is not required to have a catch clause for every possible exception--in some cases the correct response is to allow the exception to propagate up and be caught by the invoking method. In other cases, such as a programming error signaled by NullPointerException, the correct response is to not catch the exception at all, but to allow it to propagate and to have the Java interpreter exit with a stack trace and an error message.

finally

The finally clause is generally used to clean up (close files, release resources, etc.) after the try clause. What is useful about the finally clause is that the code in a finally block is guaranteed to be executed, if any portion of the try block is executed, regardless of how the code in the try block completes. In the normal case, control reaches the end of the try block and then proceeds to the finally block, which performs any necessary cleanup.

If control leaves the try block because of a return, continue, or break statement, the contents of the finally block are executed before control transfers to its new destination.

If an exception occurs in the try block and there is a local catch block to handle the exception, control transfers first to the catch block, and then to the finally block. If there is not a local catch block to handle the exception, control transfers first to the finally block, and then propagates up to the nearest catch clause that can handle the exception.

Note that if a finally block itself transfers control with a return, continue, or break statement, or by raising an exception, the pending control transfer is abandoned, and this new transfer is processed.

Also note that `try` and `finally` can be used together without exceptions or any `catch` clauses. In this case, the `finally` block is simply cleanup code that is guaranteed to be executed regardless of any `break`, `continue`, or `return` statements within the `try` clause.

Declaring Exceptions

Java requires that any method that can cause a "normal exception" to occur must either catch the exception or specify the type of the exception with a `throws` clause in the method declaration. [8] Such a `throws` clause might look like these:

[8] C++ programmers should note that Java uses `throws` where C++ uses `throw`.

```
public void open_file() throws IOException {
    // Statements here that might generate an uncaught java.io.IOException
}
public void myfunc(int arg) throws MyException1, MyException2 {
    ...
}
```

Note that the exception class specified in a `throws` clause may be a superclass of the exception type that is actually thrown. Thus if a method throws exceptions `a`, `b`, and `c`, all of which are subclasses of `d`, the `throws` clause may specify all of `a`, `b`, and `c`, or it may simply specify `d`.

We said above that the `throws` clause must be used to declare any "normal exceptions." This oxymoronic phrase refers to any subclass of `Throwable` that is not a subclass of `Error` or a subclass of `RuntimeException`. Java does not require these types of exceptions to be declared because practically any method can conceivably generate them, and it would quickly become tedious to properly declare them all. For example, every method running on a buggy Java interpreter can throw an `InternalError` exception (a subclass of `Error`) and it doesn't make sense to have to declare this in a `throws` clause for every method. Similarly, as far as the Java compiler is concerned, any method that accesses an array can generate an `ArrayIndexOutOfBoundsException` exception (a subclass of `RuntimeException`).

The standard exceptions that you often have to declare are `java.io.IOException` and a number of its more specific subclasses. `java.lang.InterruptedException` and several other less commonly used exceptions must also be declared. How do you know when you have to declare a `throws` clause? One way is to pay close attention to the documentation for the methods you call--if any "normal exceptions" can be thrown, either catch them or declare them. Another way to know what exceptions you've got to declare is to declare none and wait for the compilation errors--the compiler will tell you what to put in your `throws` clause!

Defining and Generating Exceptions

You can signal your own exceptions with the `throw` statement. The `throw` keyword must be followed by an object that is Throwable or a subclass. Often, exception objects are allocated in the same statement that they are thrown in:

```
throw new MyException("my exceptional condition occurred.");
```

When an exception is thrown, normal program execution stops and the interpreter looks for a catch clause that can handle the exception. Execution propagates up through enclosing statements and through invoking functions until such a handler is found. Any finally blocks that are passed during this propagation are executed.

Using exceptions is a good way to signal and handle errors in your own code. By grouping all your error handling and recover code together within the try/catch/finally structure, you will end up with cleaner code that is easier to understand. Sometimes, when you are throwing an exception, you can use one of the exception classes already defined by Java API. Often, though, you will want to define and throw your own exception types.

Example 2.3 shows how you can define your own exception types, throw them, and handle them. It also helps clarify how exceptions propagate. It is a long example, but worth studying in some detail. You'll know you understand exception handling if you can answer the following: What happens when this program is invoked with no argument; with a string argument; and with integer arguments 0, 1, 2, and 99?

Example 2.3: Defining, Throwing, and Handling Exceptions

```
// Here we define some exception types of our own.
// Exception classes generally have constructors but no data or
// other methods.  All these do is call their superclass constructors.
class MyException extends Exception {
  public MyException() { super(); }
  public MyException(String s) { super(s); }
}
class MyOtherException extends Exception {
  public MyOtherException() { super(); }
  public MyOtherException(String s) { super(s); }
}
class MySubException extends MyException {
  public MySubException() { super(); }
  public MySubException(String s) { super(s); }
}
public class throwtest {
  // This is the main() method.  Note that it uses two
  // catch clauses to handle two standard Java exceptions.
  public static void main(String argv[]) {
    int i;

    // First, convert our argument to an integer.
    // Make sure we have an argument and that it is convertible.
    try {  i = Integer.parseInt(argv[0]);  }
    catch (ArrayIndexOutOfBoundsException e) { // argv is empty
      System.out.println("Must specify an argument");
      return;
    }
    catch (NumberFormatException e) { // argv[0] isn't an integer
      System.out.println("Must specify an integer argument");
      return;
    }

    // Now, pass that integer to method a().
    a(i);
  }

  // This method invokes b(), which is declared to throw
  // one type of exception.  We handle that one exception.
  public static void a(int i) {
    try {
      b(i);
    }
    catch (MyException e) {                                      // Point 1
      // Here we handle MyException and its subclass MySubException.
      if (e instanceof MySubException)
        System.out.print("MySubException: ");
      else
        System.out.print("MyException: ");
      System.out.println(e.getMessage());
      System.out.println("Handled at point 1");
    }
  }

  // This method invokes c(), and handles one of the two exception
  // types that that method can throw.  The other exception type is
  // not handled, and is propagated up and declared in this method's
  // throws clause.  This method also has a finally clause to finish
  // up the work of its try clause.  Note that the finally clause is
  // executed after a local catch clause, but before a containing
  // catch clause or one in an invoking procedure.
  public static void b(int i) throws MyException {
    int result;
    try {
      System.out.print("i = " + i);
      result = c(i);
      System.out.print(" c(i) = " + result);
    }
    catch (MyOtherException e) {                                 // Point 2
```

```
      // Handle MyOtherException exceptions:
      System.out.println("MyOtherException: " + e.getMessage());
      System.out.println("Handled at point 2");
    }
    finally {
      // Terminate the output we printed above with a newline.
      System.out.print("\n");
    }
  }

  // This method computes a value or throws an exception.
  // The throws clause only lists two exceptions, because
  // one of the exceptions thrown is a subclass of another.
  public static int c(int i) throws MyException, MyOtherException {
    switch (i) {
    case 0:      // processing resumes at point 1 above
      throw new MyException("input too low");
    case 1:      // processing resumes at point 1 above
      throw new MySubException("input still too low");
    case 99:     // processing resumes at point 2 above
      throw new MyOtherException("input too high");
    default:
      return i*i;
    }
  }
}
```

PREVIOUS
Statements

HOME
BOOK INDEX

NEXT
Miscellaneous Differences

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

2.14 Miscellaneous Differences

A number of miscellaneous differences between Java and C are described in the sections that follow. Miscellaneous differences that were mentioned elsewhere, such as the lack of the `goto` statement and the `sizeof` operator, are not repeated here.

Local Variable Declarations

A feature that Java has borrowed from C++ is the ability to declare and initialize local variables anywhere in a method body or other block of code. Declarations and their initializers no longer have to be the first statements in any block--you can declare them where it is convenient and fits well with the structure of your code.

Don't let this freedom make you sloppy, however! For someone reading your program, it is nice to have variable declarations grouped together in one place. As a rule of thumb, put your declarations at the top of the block, unless you have some good organizational reason for putting them elsewhere.

Forward References

For compiler efficiency, C requires that variables and functions must be defined, or at least declared, before they can be used or called. That is, forward references are not allowed in C. Java does not make this restriction, and by lifting it, it also does away with the whole concept of a variable or function declaration that is separate from the definition.

Java allows very flexible forward references. A method may refer to a variable or another method of its class, regardless of where in the current class the variable or method is defined. Similarly, it may refer to any class, regardless of where in the current file (or outside of the file) that class is defined. The only place that forward references are not allowed is in variable initialization. A variable initializer (for local variables, class variables, or instance variables) may not refer to other variables that have not yet been declared and initialized.

Method Overloading

A technique that Java borrows from C++ is called *method overloading*. Overloaded methods are methods that have the same name, but have different signatures. In other words, they take different types of arguments, a different number of arguments, or the same type of arguments in different positions in the argument list. You cannot overload a method by changing only its return type. Two methods with the same name may have different return types, but only if the method arguments also differ. Similarly, two overloaded methods may throw different exceptions, but only if their arguments differ as well.

Method overloading is commonly used in Java to define a number of related functions with the same name, but different arguments. Overloaded methods usually perform the same basic operation, but allow the programmer to specify arguments in different ways depending on what is convenient in a given situation. Method overloading is discussed in more detail in the next chapter.

The void Keyword

The `void` keyword is used in Java, as in C, to indicate that a function returns no value. (As we will see in the next section, constructor methods are an exception to this rule.)

Java differs from C (and is similar to C++) in that methods that take no arguments are declared with empty parentheses, not with the `void` keyword. Java does not have any `void *` type, nor does it use a `(void)` cast in order to ignore the result returned by a call to a non-`void` method.

Modifiers

Java defines a number of modifier keywords that may be applied to variable and/or method declarations to provide additional information or place restrictions on the variable or method:

`final`

> The `final` keyword is a modifier that may be applied to classes, methods, and variables. It has a similar, but not identical, meaning in each case. A `final` class may never be subclassed. A `final` method may never be overridden. A `final` variable may never have its value set. In Java 1.1, this modifier may also be applied to local variables and method parameters. This modifier is discussed in more detail in the next chapter.

`native`

> `native` is a modifier that may be applied to method declarations. It indicates that the method is implemented elsewhere in C, or in some other platform-dependent fashion. A `native` method should have a semicolon in place of its body.

synchronized

> We saw the synchronized keyword in a previous section where it was a statement that marked a critical section of code. The same keyword can also be used as a modifier for class or instance methods. It indicates that the method modifies the internal state of the class or the internal state of an instance of the class in a way that is not thread-safe. Before running a synchronized class method, Java obtains a lock on the class, to ensure that no other threads can modif the class concurrently. Before running a synchronized instance method, Java obtains a lock on the instance that invoked the method, ensuring that no other thread can modify the object at the same time.

transient

> The transient keyword is a modifier that may be applied to instance fields in a class. This modifier is legal but unused in Java 1.0. In Java 1.1, it indicates a field that is not part of an object's persistent state and thus does not need to be serialized with the object.

volatile

> The volatile keyword is a modifier that may be applied to fields. It specifies that the field is used by synchronized threads and that the compiler should not attempt to perform optimizations with it. For example, it should read the variable's value from memory every time and not attempt to save a copy of it on the stack.

No Structures or Unions

Java does not support C struct or union types. Note, however, that a class is essentially the same thing as a struct, but with more features. And you can simulate the important features of a union by subclassing.

No Enumerated Types

Java does not support the C enum keyword for defining types that consist of one of a specified number of named values. This is somewhat surprising for a strongly-typed language like Java. Enumerated types can be partially simulated with the use of static final constant values.

No Method Types

C allows you to store the address of a function in a variable and to pass function addresses to other functions. You cannot do this in Java: methods are not data, and cannot be manipulated by Java programs. Note, however, that objects are data, and that objects can define methods. [9] So, when you need to pass a method to another method, you declare a class that defines the desired method and pass an instance of that class. See, for example, the `FilenameFilter` interface in the `java.io` package.

[9] An interesting way to think about objects in Java is as a kind of method that defines multiple entry points.

No Bitfields

Java does not support the C ability to define variables that occupy particular bits within `struct` and `union` types. This feature of C is usually only used to interface directly to hardware devices, which is never necessary with Java's platform-independent programming model.

No typedef

Java does not support the C `typedef` keyword to define aliases for type names. Java has a much simpler type naming scheme than C does, however, and so there is no need for something like `typedef`.

No Variable-Length Argument Lists

Java does not allow you to define methods that take a variable number of arguments, as C does. This is because Java is a strongly typed language and there is no way to do appropriate type checking for a method with variable arguments. Method overloading allows you to simulate C "varargs" functions for simple cases, but there is no general replacement for this C feature.

3. Classes and Objects in Java

Contents:

Java is an *object-oriented* language. "Object-oriented" is a term that has become so commonly used as to have practically no concrete meaning. This chapter explains just what "object-oriented" means for Java. It covers:

- Classes and objects in Java
- Creating objects
- Garbage collection to free up unused objects
- The difference between class (or static) variables and instance variables, and the difference between class (or static) methods and instance methods
- Extending a class to create a subclass
- Overriding class methods and dynamic method lookup
- Abstract classes

- Interface types and their implementation by classes

If you are a C++ programmer, or have other object-oriented programming experience, many of the concepts in this list should be familiar to you. If you do not have object-oriented experience, don't fear: This chapter assumes no knowledge of object-oriented concepts.

We saw in the last chapter that close analogies can be drawn between Java and C. Unfortunately for C++ programmers, the same is not true for Java and C++. Java uses object-oriented programming concepts that are familiar to C++ programmers, and it even borrows from C++ syntax in a number of places, but the analogies between Java and C++ are not nearly as strong as those between Java and C. [1] C++ programmers may have an easier time with this chapter than C programmers will, but they should still read it carefully and try not to form preconceptions about Java based on their knowledge of C++.

[1] As we'll see, Java supports garbage collection and dynamic method lookup. This actually makes it a closer relative, beneath its layer of C-like syntax, to languages like Smalltalk than to C++.

3.1 Introduction to Classes and Objects

A *class* is a collection of data and methods that operate on that data. [2] The data and methods, taken together, usually serve to define the contents and capabilities of some kind of *object*.

[2] A *method* is the object-oriented term for a procedure or a function. You'll see it used a lot in this book. Treat it as a synonym for "procedure."

For example, a circle can be described by the x, y position of its center and by its radius. There are a number of things we can do with circles: compute their circumference, compute their area, check whether points are inside them, and so on. Each circle is different (i.e., has a different center or radius), but as a *class*, circles have certain intrinsic properties that we can capture in a definition. Example 3.1 shows how we could partially define the class of circles in Java. Notice that the class definition contains data and methods (procedures) within the same pair of curly brackets. [3]

[3] C++ programmers should note that methods go inside the class definition in Java, not outside with the `::` operator as they usually do in C++.

Example 3.1: The Class of Circles, Partially Captured in Java Code

```
public class Circle {
    public double x, y;   // The coordinates of the center
    public double r;      // The radius
    // Methods that return the circumference and area of the circle
    public double circumference() { return 2 * 3.14159 * r; }
    public double area() { return 3.14159 * r*r; }
}
```

Objects Are Instances of a Class

Now that we've defined (at least partially) the class `Circle`, we want to do something with it. We can't do anything with the class of circles itself--we need a particular circle to work with. We need an *instance* of the class, a single circle object.

By defining the `Circle` class in Java, we have created a new data type. We can declare variables of that type:

```
Circle c;
```

But this variable `c` is simply a name that *refers to* a circle object; it is not an object itself. In Java, all objects must be created dynamically. This is almost always done with the `new` keyword:

```
Circle c;
c = new Circle();
```

Now we have created an instance of our `Circle` class--a circle object--and have assigned it to the variable `c`, which is of type `Circle`.

Accessing Object Data

Now that we've created an object, we can use its data fields. The syntax should be familiar to C programmers:

```
Circle c = new Circle();
c.x = 2.0;  // Initialize our circle to have center (2, 2) and radius 1.0.
c.y = 2.0;
c.r = 1.0;
```

Using Object Methods

This is where things get interesting! To access the methods of an object, we use the same syntax as accessing the data of an object:

```
Circle c = new Circle();
double a;
c.r = 2.5;
a = c.area();
```

Take a look at that last line. We did not say:

```
a = area(c);
```

We said:

```
a = c.area();
```

This is why it is called "object-oriented" programming; the object is the focus here, not the function call. This is probably the single most important feature of the object-oriented paradigm.

Note that we don't have to pass an argument to `c.area()`. The object we are operating on, `c`, is implicit in the syntax. Take a look at Example 3.1 again: you'll notice the same thing in the definition of the `area()` method--it doesn't take an argument. It is implicit in the language that a method operates on an instance of the class within which it is defined. Thus our `area()` method can use the `r` field of the class freely--it is understood that it is referring to the radius of whatever `Circle` instance invokes the method.

How It Works

What's going on here? How can a method that takes no arguments know what data to operate on? In fact, the `area()` method does have an argument! `area()` is implemented with an implicit argument that is not shown in the method declaration. The implicit argument is named `this`, and refers to "this object"--the `Circle` object through which the method is invoked. `this` is often called the "this pointer." [4]

[4] "this pointer" is C++ terminology. Since Java does not support pointers, I prefer the term "this reference."

The implicit `this` argument is not shown in method signatures because it is usually not needed--whenever a Java method accesses the fields in its class, it is implied that it is accessing fields in the object referred to by the `this` argument. The same is true, as we'll see, when a method in a class invokes other methods in the class--it is implicit that the methods are being invoked for the `this` object.

We can use the `this` keyword explicitly when we want to make explicit that a method is accessing its own variables and/or methods. For example, we could rewrite the `area()` method like this:

```
public double area() { return 3.14159 * this.r * this.r; }
```

In a method this simple, it is not necessary to be explicit. In more complicated cases, however, you may find that it increases the clarity of your code to use an explicit `this` where it is not strictly required.

An instance where the `this` keyword *is* required is when a method argument or a local variable in a method has the same name as one of the fields of the class. In this case, you must use `this` to access the field. If you used the field name alone, you would end up accessing the argument or local variable with the same name. We'll see examples of this in the next section.

PREVIOUS HOME NEXT
Miscellaneous Differences BOOK INDEX Object Creation

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

3.2 Object Creation

Take another look at how we've been creating our circle object:

```
Circle c = new Circle();
```

What are those parentheses doing there? They make it look like we're calling a function! In fact, that is exactly what we're doing. Every class in Java has at least one *constructor* method, which has the same name as the class. The purpose of a constructor is to perform any necessary initialization for the new object. Since we didn't define one for our `Circle` class, Java gave us a default constructor that takes no arguments and performs no special initialization.

The way it works is this: The `new` keyword creates a new dynamic instance of the class--i.e., it allocates the new object. The constructor method is then called, passing the new object implicitly (a `this` reference, as we saw above), and passing the arguments specified between parentheses explicitly.

Defining a Constructor

There is some obvious initialization we could do for our circle objects, so let's define a constructor. Example 3.2 shows a constructor that lets us specify the initial values for the center and radius of our new `Circle` object. The example also shows a use of the `this` keyword, as described in the previous section.

Example 3.2: A Constructor for the Circle Class

```
public class Circle {
    public double x, y, r;  // The center and the radius of the circle
    // The constructor method.
    public Circle(double x, double y, double r)
    {
        this.x = x;
        this.y = y;
        this.r = r;
    }
    public double circumference() { return 2 * 3.14159 * r; }
    public double area() { return 3.14159 * r*r; }
}
```

With the old, default constructor, we had to write code like this:

```
Circle c = new Circle();
c.x = 1.414;
c.y = -1.0;
c.r = .25;
```

With this new constructor the initialization becomes part of the object creation step:

```
Circle c = new Circle(1.414, -1.0, .25);
```

There are two important notes about naming and declaring constructors:

- The constructor name is always the same as the class name.
- The return type is implicitly an instance of the class. No return type is specified in constructor declarations, nor is the `void` keyword used. The `this` object is implicitly returned; a constructor should not use a `return` statement to return a value.

Multiple Constructors

Sometimes you'll want to be able to initialize an object in a number of different ways, depending on what is most convenient in a particular circumstance. For example, we might want to be able to initialize the radius of a circle without initializing the center, or we might want to initialize a circle to have the same center and radius as another circle, or we might want to initialize all the fields to default values. Doing this is no problem: A class can have any number of constructor methods. Example 3.3 shows how.

Example 3.3: Multiple Circle Constructors

```
public class Circle {
    public double x, y, r;
    public Circle(double x, double y, double r) {
        this.x = x; this.y = y; this.r = r;
    }
    public Circle(double r) { x = 0.0; y = 0.0; this.r = r; }
    public Circle(Circle c) { x = c.x; y = c.y; r = c.r; }
    public Circle() { x = 0.0; y = 0.0; r = 1.0; }
    public double circumference() { return 2 * 3.14159 * r; }
    public double area() { return 3.14159 * r*r; }
}
```

Method Overloading

The surprising thing in this example (not so surprising if you're a C++ programmer) is that all the constructor methods have the same name! So how can the compiler tell them apart? The way that you and I tell them apart is that the four methods take different arguments and are useful in different circumstances. The compiler tells them apart in the same way. In Java, a method is distinguished by its name, and by the number, type, and position of its arguments. This is not limited to constructor methods--any two methods are not the same unless they have the same name, and the same number of arguments of the same type passed at the same position in the argument list. When you call a method and there is more than one method with the same name, the compiler automatically picks the one that matches the data types of the arguments you are passing.

Defining methods with the same name and different argument types is called *method overloading*. It can be a convenient technique, as long as you only give methods the same name when they perform similar functions on slightly different forms of input data. Overloaded methods may have different return types, but only if they have different arguments. Don't confuse method overloading with *method overriding*, which we'll discuss later.

this Again

There is a specialized use of the `this` keyword that arises when a class has multiple constructors--it can be used from a constructor to invoke one of the other constructors of the same class. So we could rewrite the additional constructors from Example 3.3 in terms of the first one like this:

```
public Circle(double x, double y, double r) {
    this.x = x; this.y = y; this.r = r;
}
public Circle(double r) { this(0.0, 0.0, r); }
public Circle(Circle c) { this(c.x, c.y, c.r); }
public Circle() { this(0.0, 0.0, 1.0); }
```

Here, the `this()` call refers to whatever constructor of the class takes the specified type of arguments. This would be a more impressive example, of course, if the first constructor that we were invoking did a more significant amount of initialization, as it might, for example, if we were writing a more complicated class.

There is a very important restriction on this `this` syntax: it may only appear as the first statement in a constructor. It may, of course, be followed by any additional initialization that a particular version of the constructor needs to do. The reason for this restriction involves the automatic invocation of superclass constructor methods, which we'll explore later in this chapter.

PREVIOUS | HOME | NEXT
Introduction to Classes and Objects | BOOK INDEX | Class Variables

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

3.3 Class Variables

In our `Circle` class definition, we declared three "instance" variables: `x`, `y`, and `r`. Each instance of the class--each circle--has its own copy of these three variables. These variables are like the fields of a `struct` in C--each instance of the `struct` has a copy of the fields. Sometimes, though, we want a variable of which there is only one copy--something like a global variable in C.

The problem is that Java doesn't allow global variables. (Actually, those in the know consider this is feature!) Every variable in Java must be declared inside a class. So Java uses the `static` keyword to indicate that a particular variable is a *class variable* rather than an *instance variable*. That is, that there is only one copy of the variable, associated with the class, rather than many copies of the variable associated with each instance of the class. The one copy of the variable exists regardless of the number of instances of the class that are created--it exists and can be used even if the class is never actually instantiated.

This kind of variable, declared with the `static` keyword, is often called a *static variable*. I prefer (and recommend) the name "class variable" because it is easily distinguished from its opposite, "instance variable." We'll use both terms in this book.

An Example

As an example (a somewhat contrived one), suppose that while developing the `Circle` class we wanted to do some testing on it and determine how much it gets used. One way to do this would be to count the number of `Circle` objects that are instantiated. To do this we obviously need a variable associated with the class, rather than with any particular instance. Example 3.4 shows how we can do it--we declare a `static` variable and increment it each time we create a `Circle`.

Example 3.4: Static Variable Example

```
public class Circle {
    static int num_circles = 0; // class variable: how many circles created
    public double x, y, r;      // instance vars: the center and the radius
    public Circle(double x, double y, double r) {
        this.x = x; this.y = y; this.r = r;
        num_circles++;
    }
    public Circle(double r) { this(0.0, 0.0, r); }
    public Circle(Circle c) { this(c.x, c.y, c.r); }
    public Circle() { this(0.0, 0.0, 1.0); }
    public double circumference() { return 2 * 3.14159 * r; }
    public double area() { return 3.14159 * r*r; }
}
```

Accessing Class Variables

Now that we are keeping track of the number of `Circle` objects created, how can we access this information? Because `static` variables are associated with the class rather than with an instance, we access them through the class rather than through the instance. Thus, we might write: [5]

> [5] Recall that `System.out.println()` prints a line of text, and that the string concatenation operator, `+`, converts non-string types to strings as necessary.

```
System.out.println("Number of circles created: " + Circle.num_circles);
```

Notice that in our definition of the constructor method in Example 3.4, we just used `num_circles` instead of `Circle.num_circles`. We're allowed to do this within the class definition of `Circle` itself. Anywhere else, though, we must use the class name as well.

Global Variables?

Earlier we said that Java does not support global variables. In a sense, though, `Circle.num_circles` behaves just like one. What is different from a global variable in C is that there is no possibility of name conflicts. If we use some other class with a class variable named `num_circles`, there won't be a "collision" between these two "global" variables, because they must both be referred to by their class names. Since each class variable must be part of a class and must be referred to with its class name, each has a unique name. Furthermore, each class has a unique name because, as we saw in Chapter 2, *How Java Differs from C*, it is part of a package with a unique name.

Constants: Another Class Variable Example

Let's try a less forced example of why you might want to use a class variable with the `Circle` class. When computing the area and circumference of circles, we use the value pi. Since we use the value frequently, we don't want to keep typing out 3.14159, so we'll define it as a class variable that has a convenient name:

```
public class Circle {
    public static final double PI = 3.14159265358979323846;
    public double x, y, r;
    // ... etc....
}
```

Besides the static keyword that we've already seen, we use the final keyword, which means that this variable can never have its value changed. This prevents you from doing something stupid like:

```
Circle.PI = 4;
```

which would tend to give you some pretty square-looking circles.

The Java compiler is smart about variables declared both static and final--it knows that they have constant values. So when you write code like this:

```
double circumference = 2 * Circle.PI * radius;
```

the compiler precomputes the value 2 * Circle.PI , instead of leaving it for the interpreter.

Java does not have a preprocessor with a C-style #define directive. static final variables are Java's substitute for C's #define'd constants. Note that the C convention of capitalizing constants has been carried over into Java.

PREVIOUS Object Creation | HOME BOOK INDEX | NEXT Class Methods

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

3.4 Class Methods

Let's define a new method in our `Circle` class. This one tests whether a specified point falls within the defined circle:

```
public class Circle {
    double x, y, r;
    // is point (a,b) inside this circle?
    public boolean isInside(double a, double b)
    {
        double dx = a - x;
        double dy = b - y;
        double distance = Math.sqrt(dx*dx + dy*dy);
        if (distance < r) return true;
        else return false;
    }
        .
        .  // Constructor and other methods omitted.
        .
}
```

What's this `Math.sqrt()` thing? It looks like a method call and, given its name and its context, we can guess that it is computing a square root. But the method calls we've discussed are done through an object. `Math` isn't the name of an object that we've declared, and there aren't any global objects in Java, so this must be a kind of method call that we haven't seen before.

static Methods

What's going on here is that `Math` is the name of a class. `sqrt()` is the name of a *class method* (or static method) defined in `Math`. It differs from the *instance methods*, such as `area()` in `Circle`, that we've seen so far.

Class methods are like class variables in a number of ways:

- Class methods are declared with the `static` keyword.

- Class methods are often referred to as "static methods."

- Class methods are invoked through the class rather than through an instance. (Although within the class they may be invoked by method name alone.)

- Class methods are the closest Java comes to "global" methods. Because they must be referred to by the class name, there is no danger of name conflicts.

No this

Class methods differ from instance methods in one important way: they are not passed an implicit `this` reference. Thus, these `this`-less methods are not associated with any instance of the class and may not refer to any instance variables or invoke instance methods.

Since class methods are not passed a `this` reference, and are not invoked through an object, they are the closest thing that Java offers to the "normal" C procedures that you may be accustomed to, and may therefore seem familiar and comforting. If you're sick and tired of this object-oriented business, it is perfectly possible to write complete Java programs using only class methods, although this does defeat an important purpose of using the language!

But don't think that class methods are somehow cheating--there are perfectly good reasons to declare a method `static`. And indeed, there are classes like `Math` that declare all their methods (and variables) `static`. Since `Math` is a collection of functions that operate on floating-point numbers, which are a primitive type, there are no objects involved, and no need for instance methods. `System` is another class that defines only class methods--it provides a varied collection of system functions for which there is no appropriate object framework.

A Class Method for Circles

Example 3.5 shows two (overloaded) definitions of a method for our `Circle` class. One is an instance method and one is a class method.

Example 3.5: A Class Method and an Instance Method

```
public class Circle {
    public double x, y, r;
    // An instance method.  Returns the bigger of two circles.
    public Circle bigger(Circle c) {
        if (c.r > r) return c; else return this;
    }
    // A class method.  Returns the bigger of two circles.
    public static Circle bigger(Circle a, Circle b) {
        if (a.r > b.r) return a; else return b;
    }
```

```
        .
        .  // Other methods omitted here.
        .
}
```

You would invoke the instance method like this:

```
Circle a = new Circle(2.0);
Circle b = new Circle(3.0);
Circle c = a.bigger(b);         // or, b.bigger(a);
```

And you would invoke the class method like this:

```
Circle a = new Circle(2.0);
Circle b = new Circle(3.0);
Circle c = Circle.bigger(a,b);
```

Neither of these is the "correct" way to implement this method. One or the other will seem more natural, depending on circumstances.

A Mystery Explained

Now that we understand class variables, instance variables, class methods, and instance methods, we are in a position to explore that mysterious method call we saw in our very first Java "Hello World" example:

```
System.out.println("Hello world!");
```

One hypothesis is that `println()` is a class method in a class named `out`, which is in a package named `System`. Syntactically, this is perfectly reasonable (except perhaps that class names always seem to be capitalized by convention, and `out` isn't capitalized). But if you look at the API documentation, you'll find that `System` is not a package name; it is the name of a class (which is in the `java.lang` package, by the way). Can you figure it out?

Here's the story: `System` is a class. It has a class variable named `out`. `out` refers to an object of type `PrintStream`. The object `System.out` has an instance method named `println()`. Mystery solved!

Static Initializers

Both class and instance variables can have initializers attached to their declarations. For example:

```
static int num_circles = 0;
float r = 1.0;
```

Class variables are initialized when the class is first loaded. Instance variables are initialized when an object is created.

Sometimes we need more complex initialization than is possible with these simple variable initializers. For instance variables, there are constructor methods, which are run when a new instance of the class is created. Java also allows you to write an initialization method for class variables. Such a method is called a *static initializer*.

The syntax of static initializers gets kind of bizarre. Consider that a static initializer is invoked automatically by the system when the class is loaded. Thus there are no meaningful arguments that can be passed to it (unlike the arguments we can pass to a constructor method when creating a new instance). There is also no value to return. So a static initializer has no arguments and no return value. Furthermore, it is not really necessary to give it a name, since the system calls the method automatically for us. What part of a method declaration is left? Just the `static` keyword and the curly brackets!

Example 3.6 shows a class declaration with a static initializer. Notice that the class contains a regular static variable initializer of the kind we've seen before, and also a static initializer--an arbitrary block of code between `{` and `}`.

Example 3.6: A Static Initializer

```
// We can draw the outline of a circle using trigonometric functions.
// Trigonometry is slow though, so we pre-compute a bunch of values.
public class Circle {
    // Here are our static lookup tables, and their own simple initializers.
    static private double sines[] = new double[1000];
    static private double cosines[] = new double[1000];
    // Here's a static initializer "method" that fills them in.
    // Notice the lack of any method declaration!
    static {
        double x, delta_x;
        int i;
        delta_x = (Circle.PI/2)/(1000-1);
        for(i = 0, x = 0.0; i < 1000; i++, x += delta_x) {
            sines[i] = Math.sin(x);
            cosines[i] = Math.cos(x);
        }
    }
        .
        .  // The rest of the class omitted.
        .
}
```

The syntax gets even a little stranger than this. Java allows any number of static initializer blocks of code to appear within a class definition. What the compiler actually does is to internally produce a single class initialization routine that combines all the static variable initializers and all of the static initializer blocks of code, in the order that they appear in the class declaration. This single initialization procedure is run automatically, one time only, when the class is first loaded.

One common use of static initializers is for classes that implement `native` methods--i.e., methods written in C. The static initializer for such a class should call `System.load()` or `System.loadLibrary()` to read in the native library that implements these native methods.

Instance Initializers

In Java 1.1, a class definition may also include *instance initializers*. These look like static initializers, but without the `static` keyword. An instance initializer is like a constructor: it runs when an instance of the class is created. We'll see more about instance initializers in Chapter 5, *Inner Classes and Other New Language Features*, *Inner Classes and Other New Language Features*.

5. Inner Classes and Other New Language Features

Contents:

The largest enhancement to the Java language in Java 1.1 is something called "inner classes." With this addition to the language, classes can be defined as members of other classes, just as fields and methods can be defined within classes. Classes can also be defined within a block of Java code, just as local variables can be defined within a block of code.

From one point of view, the addition of inner classes regularizes the syntax of Java. From another point of view, though, inner classes create quite a few special cases, and a confusing array of new rules. In practice, however, if you avoid the obscure and pathological cases, inner classes prove to be an elegant and extremely useful addition to the language. Their use is particularly common in conjunction with the new event model defined by the AWT in Java 1.1.

5.1 An Overview of Inner Classes

Java 1.0 allowed classes and interfaces to be defined in exactly one context: at the "top level," as members of packages. Java 1.1 adds one new type of top-level classes and interfaces, and adds three new types of "inner classes," as outlined below. Later sections of this chapter describe each of these new types of classes and interfaces in more detail and present examples of their use.

Nested top-level classes and interfaces

A *nested top-level* class or interface is defined as a static member of an enclosing top-level class or interface. The definition of a nested top-level class uses the `static` modifier, just as the definition of a static method or static field does. Nested interfaces are implicitly static (though they may be declared `static` to make this explicit) and so are always top-level. A nested top-level class or interface behaves just like a "normal" class or interface that is a member of a package. The difference is that the name of a nested top-level class or interface includes the name of the class in which it is defined. Thus, a `LinkedList` class could define a nested top-level interface `Linkable`. This interface would be referred to as `LinkedList.Linkable`. Nested top-level classes and interfaces are typically used as a convenient way to group related classes.

Member classes

A *member class* is also defined as a member of an enclosing class, but unlike a nested top-level class, it is not defined with the `static` modifier. This means that it is an inner class, rather than a top-level class. Nested interfaces are always implicitly `static`, so they are always top-level; there is no such thing as a "member interface," or any kind of "inner interface." In many ways, a member class is analogous to the other members--the instance fields and methods--of a class. Member classes are of interest because the code within a member class can implicitly refer to any of the fields and methods, including `private` fields and methods, of its enclosing class. [1] Every instance of a member class is associated with an enclosing instance of the class that defines it. Because of the requirement for this enclosing instance, several new pieces of syntax have been introduced into the Java language.

[1] Unfortunately, in Java 1.1 and 1.1.1 there are compiler bugs that prevent access to the `private` fields and methods of enclosing classes from working correctly. It is not yet clear when these bugs will be fixed. So while access to `private` members of enclosing classes is part of the inner class specification, it is a feature that is currently best avoided. If a field or method must be visible to nested classes, you should give it package visibility rather than `private` visibility.

Local classes

A *local class* is an inner class defined within a block of Java code; it is visible only within that block. Interfaces can not be defined locally. Because a local class is defined within a block of code, it is analogous, in some ways, to a local variable. Local classes are not member classes, but can still use the fields and methods of enclosing classes. More important, however, the code within a local class definition can use any `final` local variables or parameters that are accessible in the scope of the block that defines the class. Local classes are useful primarily as "adapter classes" and are commonly used with the new event-handling model required by the Java 1.1 AWT and by JavaBeans. For example, a block of Java 1.1 code that creates a `java.awt.Button` object could use a local class to define a simple implementation of the `java.awt.event.ActionListener` interface. Then it could instantiate this simple implementation and pass the resulting object to the button's `addActionListener()` method, thereby connecting the button to the "callback" code that is executed when the button is pressed.

Anonymous classes

An *anonymous class* is an extension to the local class concept described above. Instead of declaring a local class with one Java statement, and then instantiating and using it in another statement, an anonymous class combines the two steps in a single Java expression. An anonymous class, as you might guess, does not have a name. And because it is instantiated in the same expression that defines it, it can only be instantiated once. Except for these differences, anonymous classes are quite similar to local classes in behavior and use. Interfaces cannot be defined anonymously, of course. When writing a simple adapter class, the choice between a named local class and an unnamed anonymous class typically comes down to a matter of style and code clarity, rather than any difference in functionality.

Table 5.1 summarizes the types of classes and interfaces that can be defined in Java 1.1; the remaining sections of the chapter document each type in more detail.

Table 5.1: Inner Class Summary

Class Type		**Description**
Top-level classes and interfaces	Package member class or interface	An ordinary class or interface that is a direct member of a package. The basic Java class understood by the VM. All nested and inner classes are converted to this type.
	Nested top-level class or interface	A conveniently nested top-level class or interface. Must be declared `static` within another top-level class or interface. (Nested interfaces are implicitly `static`.) May use the `static` members of its containing type.

Inner classes	Member class	A class defined as a member (non-`static`) of another. Each instance has an enclosing instance, and can use its members. New syntax for `this`, `new`, and `super`. Cannot have `static` members. Cannot have same name as a containing class.
	Local class	A class defined in a block of code. Can use members of enclosing classes and `final` local variables and parameters. New `this` syntax. Same restrictions as member classes.
	Anonymous class	Unnamed class defined within an expression. Has features of a local class. Allows a one-shot class to be defined exactly where needed. Same restrictions as local class, plus has no name or constructor. Only one instance of the class is created.

4.12 New JDK Utilities

JDK 1.1 includes a number of new tools. In the discussion of applets above, we've already seen *jar* for creating JAR archives and *javakey* for adding digital signatures to JAR archives. In fact, *javakey* can do much more than that--it is a very flexible tool for managing a database of entities, generating keys and certificates, and generating digital signatures.

serialver is a new tool used in conjunction with object serialization. When an object is deserialized, it is important to verify that the version of the class file for that object matches the version that was used to serialize it. This is done by computing a unique identifier for the class and encoding it in a private variable of the class. When an incompatible change is made to the class, a new unique identifier is computed, and the new value is stored in the private variable. It is the *serialver* tool that is used to compute this unique identifier.

native2ascii is a tool for programmers who program in a locale that uses a non-ASCII file encoding. The *javac* compiler can only compile files encoded in ASCII, with all Unicode characters converted to the `\u`*xxxx* format. What *native2ascii* does is to convert its input file to Unicode, and then output that Unicode version as an ASCII file that uses the `\u` escape for all non-ASCII Unicode characters. After you process a locally-encoded file with *native2ascii*, *javac* can compile it.

In addition to the tools described here, JDK 1.1 also includes two new programs, *rmic* and *rmiregistry*, that are used in conjunction with Remote Method Invocation. They will be documented in *Java Enterprise in a Nutshell*.

4.11 Applet Changes

There are several new features in Java 1.1 that affect applets. The first is the introduction of JAR files. "JAR" stands for Java ARchive, and a JAR file is just that: an archive of files used by a Java applet. An applet often requires multiple class files, as well as images, sounds, and other resources, to be loaded over the network. Prior to Java 1.1, each of these files was loaded through a separate HTTP request, which is fairly inefficient. With Java 1.1, all (or many) of the files an applet needs can be combined into a single JAR file, which an applet viewer or Web browser can download with a single HTTP request. Chapter 6, *Applets*, demonstrates the use of JAR files.

JAR files are stored in the ZIP file format. A JAR archive can be created with the *jar* tool shipped with the JDK. Once you have created a JAR file, you refer to it in a `<APPLET>` tag with the `ARCHIVE` attribute. This `ARCHIVE` attribute may actually be set to a comma-separated list of archive files to be downloaded. Note that specifying an `ARCHIVE` attribute simply tells the applet viewer or browser the name of a JAR file or files to load; it does not tell the browser the name of the applet that is to be run. Thus, you still must specify the `CODE` attribute (or the new `OBJECT` attribute, as we'll see below). For example, you might use an `<APPLET>` tag like the following to tell the browser to download the *animation.jar* file and start the applet contained in the file *Animator.class*:

```
<APPLET CODE="Animator.class" ARCHIVE="animation.jar" WIDTH=500 HEIGHT=200>
</APPLET>
```

There is another advantage to the use of JAR files. Every JAR file contains a "manifest" file, which you either specify explicitly when you create the archive, or which is created for you by the *jar* tool. The manifest is stored in a file named `META-INF/MANIFEST.MF` and contains meta-information about the files in the archive. By default, the *jar* tool creates a manifest file that contains MD5 and SHA message digests for each file in the archive. This information can be used by the applet viewer or Web browser to verify that the files in the archive have not been corrupted since the JAR file was created.

The main reason to include message digests in the manifest file, however, is so that a JAR file can have digital signatures added to it. An archive can be signed with the *javakey* tool. What a digital signature allows you to do is verify that the files in a JAR file have not been modified since the digital signature was added to the archive. If you trust the person or entity who signed the file, then you ought to trust the applet contained in the JAR file. (The *javakey* tool allows you to specify whether or not you trust any given entity.) Chapter 6, *Applets* also describes how you might use digital signatures and *javakey*.

In JDK 1.1, the *appletviewer* tool understands digitally signed JAR files. When it loads an applet that has been signed by a trusted entity, it runs that applet without subjecting it to the usual security restrictions--the applet can read and write files, and do anything that a standalone Java application can do. Common Web browsers are likely to follow suit and give special privileges to trusted applets. One refinement we may see in the future is the ability to specify varying levels of trust, and to assign different sets of privileges to applets at those varying trust levels.

Besides the introduction of JAR files and trusted applets, Java 1.1 also supports "serialized applets." In an `<APPLET>` tag, you can specify the `OBJECT` attribute instead of the `CODE` attribute. If you do this, the value of the `OBJECT` attribute should be the name of a file that contains a serialized representation of the applet to be run. Graphical application-builder tools may prefer to output applets as pre-initialized object trees, rather than generating custom Java code to perform the initializations. See Chapter 9, *Object Serialization* for more information on serialized applets.

4.10 Enterprise APIs: JDBC, RMI, and Security

Java 1.1 provides a number of important new features that are loosely grouped under the name "Enterprise APIs." These include JDBC (Java DataBase Connectivity), RMI (Remote Method Invocation), and Java Security. With release 1.1, Java has grown too big for all of it to be documented, even in quick-reference format, in a single volume. Therefore, the JDBC, RMI, and Security packages will be documented, along with other, forthcoming Enterprise APIs, in a separate volume, *Java Enterprise in a Nutshell*. Note, however, that while this volume does not cover the Java Security API, it *does* cover applet security, signed applets, and the *javakey* program that is used to create digital signatures, generate key pairs, and manage a database of entities and their keys.

4.9 Java Beans

JavaBeans is a "software component model" for Java that has generated quite a lot of interest from many quarters. The JavaBeans API specification defines "beans" as follows: "A Java Bean is a reusable software component that can be manipulated visually in a builder tool." The `java.beans` package defines classes and interfaces designed to work with beans at three distinct levels, described below.

Much of the JavaBeans API is intended for use only by those few people who are writing interface builder tools that manipulate beans. The main thing that a builder tool needs to be able to do with beans is "introspect" on them--i.e., to determine what properties are exposed by a bean, what methods it exports, and what events it can generate. This is information that a builder tool must be able to display to the programmer who is using the tool. The JavaBeans API defines a set of naming conventions for the methods that a bean defines. If a bean follows these conventions, a builder tool can use the new Reflection API to determine what properties, methods, and events the bean supports. The `Introspector` class uses reflection to obtain information about a bean and presents it to the builder tool in the form of a `BeanInfo` object, which itself contains various `FeatureDescriptor` objects describing the properties, methods, and events of the bean.

At the second level, the JavaBeans API contains classes and interfaces intended for use by programmers who are creating beans for others to use. Chapter 10, *Java Beans* describes how to use the classes in `java.beans` in this manner. One of the surprising features of the JavaBeans API is that there is no `Bean` class that all beans must extend. A bean can be of any class; however, as we've seen, beans should follow certain naming conventions. The `java.beans` classes that a bean creator uses are generally auxiliary classes, used not by the bean, but by the builder tool that manipulates the bean. These auxiliary classes are shipped with a bean, and provide additional information or methods that a builder tool may use with the bean. These classes are not included in finished software built with the bean.

For example, one of the auxiliary classes a bean may define is a custom `BeanInfo` class to provide information to the builder tool that is not available through the Reflection API. This information might include a human-readable description of the bean's properties, methods, and events, for example. Or, if a bean does not follow the standard naming conventions, this custom `BeanInfo` class must also provide more basic information about the bean's properties, methods, and events.

Besides a `BeanInfo` class, complex beans may also provide a `Customizer` class and one or more `PropertyEditor` classes. A `Customizer` class is a kind of configuration tool or "wizard" for a bean. It is instantiated by the builder tool in order to guide the user through bean customization. A `PropertyEditor` class is used to allow the user to edit the value of bean properties of a particular class. Builder tools have built-in property editors for common types such as strings, colors, and fonts, but a bean that has properties of some unusual or custom type may want to provide a `PropertyEditor` subclass to allow the user to easily specify values for those properties.

The third level at which the JavaBeans API can be used is by programmers who are assembling an application using beans. Some programmers may do this through a builder tool, while others may do it "by hand", the old-fashioned way. Programmers using beans do not typically have to use the `java.beans` package. At this level, it is more a matter of reading the documentation for the particular beans being used and following those instructions. Nevertheless, a programmer using beans does need to be familiar with the event model used by beans, which is the same as the Java 1.1 event model for AWT. Also, programmers using beans "by hand" should be familiar with the naming conventions for bean properties, methods, and events, in order to more easily understand how a given bean can be used. In Java 1.1, all AWT components are beans and follow these naming conventions.

4.8 Reflection

Reflection in Java 1.1 refers to the ability of Java classes to reflect upon themselves, or to "look inside themselves." The `java.lang.Class` class has been greatly enhanced in Java 1.1. It now includes methods that return the fields, methods, and constructors defined by a class. These items are returned as objects of type `Field`, `Method`, and `Constructor`, respectively. These new classes are part of the new `java.lang.reflect` package, and they each provide methods to obtain complete information about the field, method, or constructor they represent. For example, the `Method` object has methods to query the name, the parameter types, and the return type of the method it represents. Chapter 12, *Reflection* provides some examples of using the Reflection API.

Besides allowing a program to inspect the members of a class, the `java.lang.reflect` package also allows a program to manipulate these fields and methods. The `Field` class defines methods that get and set the value of the represented field for any given object of the appropriate type. Similarly, the `Method` object defines an `invoke()` method that allows the represented method to be invoked, and the `Constructor` class defines a `newInstance()` method that creates a new object and invokes the represented constructor on it. `java.lang.reflect` also defines an `Array` class. It does not represent a specific array, but defines static methods that read and write array elements and dynamically create new arrays.

With the addition of reflection, the `Class` class has been expanded to represent not just Java classes, but any Java type, including primitive types and array types. There is a special `Class` object that represents each of the eight Java primitive types, and another special `Class` object that represents the `void` type. These special `Class` objects are available as constants in the wrapper objects for the primitive types. `Integer.TYPE` is a `Class` object that represents the `int` type, for example, and `Void.TYPE` is a `Class` object that represents the `void` type.

Finally, new Java language syntax makes it easier to obtain a `Class` object that represents a Java class. If you follow the name of a class, interface, or other type with `.class`, Java evaluates that expression and returns the corresponding `Class` object. So, for example, the following two expressions are equivalent:

```
String.class
Class.forName("java.lang.String")
```

Note that this syntax also works with primitive type names: you can write `short.class`, for example, which returns the same value as `Short.TYPE`.

PREVIOUS | HOME | NEXT
Object Serialization | BOOK INDEX | Java Beans

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

4.7 Object Serialization

Object serialization is one of the major new features of Java 1.1. It refers to the ability to write the complete state of an object (including any objects it refers to) to an output stream, and then recreate that object at some later time by reading its serialized state from an input stream. You can serialize an object simply by passing it to the `writeObject()` method of an `ObjectOutputStream`. Similarly, you can create an object from a serialized object stream by calling the `readObject()` method of an `ObjectInputStream`. Both of these new object stream types are part of the `java.io` package.

Typically, object serialization is as simple as calling `writeObject()` and `readObject()`. There are a few additional twists, however, that are worth mentioning here. First, only objects that subclass the `Serializable` (or `Externalizable`) interface can be serialized. The `Serializable` interface does not define any methods, but merely acts as a marker that indicates whether serialization is allowed on a given object. Second, fields of a class declared `transient` are not serialized as part of an object's state. The `transient` modifier was legal in Java 1.0, but had no defined behavior. Third, some objects may need to implement custom serialization or de-serialization behavior. They can do this by implementing special `readObject()` and `writeObject()` methods. Chapter 9, *Object Serialization* describes all of these aspects of object serialization in more detail.

Despite the fact that only a few classes and interfaces are part of the Object Serialization API, serialization is a very important technology and is used in several places in Java 1.1. It is used as the basis for transferring objects via cut-and-paste. It is used to transfer objects between a client and a server for remote method invocation. It is used by the JavaBeans API--beans are often provided as pre-initialized, serialized objects, rather than merely as class files. Java 1.1 also adds the capability for applets to be loaded into an applet viewer or browser as serialized objects. One common use we are likely to see for object serialization is as a way to save user preferences and other application states--a serialized object is an instant file format that works for any application. Another use that should be popular with GUI builder tools is saving the complete `Component` hierarchy of an application's GUI as a serialized object, and then later loading in that object in order to automatically recreate the GUI.

PREVIOUS
Internationalization

HOME
BOOK INDEX

NEXT
Reflection

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

4.6 Internationalization

Internationalization [1] is the process of enabling a program to run internationally. That is, an internationalized program has the flexibility to run correctly in any country. Once a program has been internationalized, enabling it to run in a particular country and/or language is merely a matter of "localizing" it for that country and language, or *locale*.

You might think that the main task of localization is the matter of translating a program's user-visible text into a local language. While this is an important task, it is not by any means the only one. Other concerns include displaying dates and times in the customary format for the locale, displaying number and currency values in the customary format for the locale, and sorting strings in the customary order for the locale.

Underlying all these localization issues is the even more fundamental issue of character encodings. Almost every useful program must perform input and output of text, and before we can even think about textual I/O, we must be able to work with the local character encoding standard. This hurdle to internationalization lurks slightly below the surface, and is not very "programmer-visible." Nevertheless, it is one of the most important and difficult issues in internationalization.

As described in Chapter 11, *Internationalization*, Java 1.1 provides facilities that address all of these internationalization issues. If you write programs that correctly make use of these facilities, the task of localizing your program for a new country really does boil down to the relatively simple matter of hiring a translator to convert your program's messages. With the expansion of the global economy, and particularly of the global Internet, writing internationalized programs is going to become more and more important, and you should begin to take advantage of Java's internationalization capabilities right away.

There are several distinct pieces to the problem of internationalization, and Java's solution to this problem also comes in distinct pieces. The first issue in internationalization is the matter of knowing

[1] This word is sometimes abbreviated I18N, because there are 18 letters between the first I and the last N.

what locale a program is running in. A locale is typically defined as a political, geographical, or cultural region that has a distinct language or distinct conventions for things such as date and time formats. The notion of a locale is encapsulated in Java 1.1 by the `Locale` class, which is part of the `java.util` package. Every Java program has a default locale, which is inherited from the operating system (where it may be set by the user). A program can simply rely on this default, or it can change the default. Additionally, all Java methods that rely on the default locale also have variants that allow you to explicitly specify a locale. Typically, though, using the default locale is exactly what you want to do.

Once a program knows what locale it is running in, the most fundamental internationalization issue, as noted above, is the ability to read and write localized text. Since Java uses the Unicode encoding for its characters and strings, any character of any commonly-used modern written language is representable in a Java program, which puts Java at a huge advantage over older languages such as C and C++. Thus, working with localized text is merely a matter of converting from the local character encoding to Unicode when reading text, such as a file or input from the user, and converting from Unicode to the local encoding when writing text. Java's solution to this problem is in the `java.io` package, in the form of a new suite of character-based input and output streams (known as "readers" and "writers") that complement the existing byte-based input and output streams.

The `FileReader` class, for example, is a character-based input stream used to read characters (which are not the same as bytes in all languages) from a file. The `FileReader` class assumes that the specified file is encoded using the default character encoding for the default locale, so it converts characters from the local encoding to Unicode characters as it reads them. In most cases, this assumption is a good one, so all you need to do to internationalize the character set handling of your program is to switch from a `FileInputStream` to a `FileReader` object, and make similar switches for text output as well. On the other hand, if you need to read a file that is encoded using some character set other than the default character set of the default locale, you can use a `FileInputStream` to read the bytes of the file, and then use an `InputStreamReader` to convert the stream of bytes to a stream of characters. When you create an `InputStreamReader`, you specify the name of the encoding in use, and it performs the appropriate conversion automatically.

As you can see, internationalizing the character set of your programs is a simple matter of switching from byte I/O streams to character I/O streams. Internationalizing other aspects of your program requires a little more effort. The classes in the `java.text` package are designed to allow you to internationalize your handling of numbers, dates, times, string comparisons, and so on. `NumberFormat` is used to convert numbers, monetary amounts, and percentages to an appropriate textual format for a locale. Similarly, the `DateFormat` class, along with the `Calendar` and `TimeZone` classes from the `java.util` package, are used to display dates and times in a locale-specific way. The `Collator` class is used to compare strings according to the alphabetization rules of a given locale, and the `BreakIterator` class is used to locate word, line, and sentence boundaries.

The final major problem of internationalization is making your program flexible enough to display messages (or any type of user-visible text, such as the labels on GUI buttons) to the user in an appropriate language for the current locale. Typically, this means that the program cannot use hard-coded messages and must instead read in a set of messages at run-time, based on the locale setting. Java provides an easy way to do this. You define your messages as key/value pairs in a `ResourceBundle` subclass. Then, you create a subclass of `ResourceBundle` for each language or locale your application supports, naming each class following a convention that includes the locale name. At run-time, you can use the `ResourceBundle.getBundle()` method to load the appropriate `ResourceBundle` class for the current locale. The `ResourceBundle` contains the messages your application uses, each associated with a key, that serves as the message name. Using this technique, your application can look up a locale-dependent message translation based on a locale-independent message name. Note that this technique is useful for things other than textual messages. It can be used to internationalize icons, or any other locale-dependent object used by your application.

There is one final twist to this problem of internationalizing messages. Often, we want to output messages such as, "Error at line 5 of file hello.java.", where parts of the message are static, and parts are dynamically generated at run-time (such as the line number and filename above). This is further complicated by the fact that when we translate such messages the values we substitute in at run-time may appear in a different order. For example, in some different English speaking locale, we might want to display the line above as: "hello.java: error at line 5". The `MessageFormat` class of the `java.text` package allows you to substitute dynamic values into static messages in a very flexible way and helps tremendously with this situation, particularly when used in conjunction with resource bundles.

4.5 Other AWT Improvements

In addition to the major change in the AWT event model, there have been quite a few other improvements to the AWT. These improvements are summarized in the sections below.

Printing

Printing in Java 1.1 is implemented through the new `PrintJob` class and `PrintGraphics` interface. The `PrintJob` class represents a print request. When a `PrintJob` object is created, the user is prompted with a platform-dependent print dialog, which allows her to specify options such as which printer to use.

The `getGraphics()` method of a `PrintJob` object returns a `Graphics` object that can be used for printing. This object is an instance of a subclass of `Graphics` that knows how to print in a platform-dependent way. The object also implements the `PrintGraphics` interface. To print a component, you simply pass this `Graphics` object to the component's `print()` method. If the component does not define this method, the default implementation simply invokes the `paint()` method, which usually does the right thing. When you want to print a component and all of its subcomponents, you can simply pass the `Graphics` object to the `printAll()` method of the component.

Printing multiple pages is more complex, of course. The application is responsible for pagination of the output, and in order to draw the output on the page the application may also need to query the `PrintJob` object to determine the page size (in pixels) and page resolution (in pixels per inch).

For security reasons, applets are not allowed to initiate print jobs; if they were, you could expect to see applets on the Net that automatically printed hardcopy advertisements to your printer! Note, however, that this does not mean that applets cannot print themselves when the browser or applet viewer initiates the print request object and invokes the `printAll()` method of the applet.

Chapter 8, *New AWT Features* contains an example that uses the printing capabilities of Java 1.1.

Cut-and-Paste

Data transfer via the cut-and-paste metaphor is supported in Java 1.1 by the classes and interfaces in the `java.awt.datatransfer` package. One half of this package provides generic data-transfer functionality, and the other half provides the classes and interfaces needed for clipboard-based cut-and-paste. In future versions of the JDK, we can expect to see support for the drag-and-drop data transfer metaphor added to this package.

For the purposes of data transfer, the `DataFlavor` class represents the notion of a data type or data format. A `DataFlavor` consists of a human-readable name for the flavor and one of two possible machine-readable format definitions. The first of the machine-readable descriptions is a `String` that specifies a MIME type for the transferred data. The second is a `Class` object that represents a Java class. When a `DataFlavor` is specified with a `Class` object, it is an instance of this class that is passed when data transfer actually occurs.

Any value that can be transferred through the Java 1.1 data transfer mechanism must be represented by a class that implements the `Transferable` interface. This interface defines methods to query the data flavors that the class supports, and it defines a method that the data transfer mechanism calls to convert the transferable value into the appropriate form for a given `DataFlavor`.

While the `DataFlavor` class and the `Transferable` interface define the fundamental data transfer mechanism, they, by themselves, are not enough to initiate or perform data transfer. For this purpose, `java.awt.datatransfer` also defines the `Clipboard` class and the `ClipboardOwner` interface. Together, they support a cut-and-paste metaphor for data transfer. Because strings are often transferred between applications, `java.awt.datatransfer` provides the `StringSelection` class. This class implements both the `Transferable` and the `ClipboardOwner` interfaces and makes it very easy to transfer textual data through cut-and-paste.

Inter-application data transfer is performed through the system clipboard. It is also possible to perform intra-application transfers through a private clipboard that an application creates for itself. Note that untrusted applets are not allowed to access the system clipboard--there could be sensitive information contained on it that untrusted code should not have access to. This means that applets cannot participate in inter-application cut-and-paste. Chapter 8, *New AWT Features* provides an example that demonstrates intra-application cut-and-paste data transfer.

Popup Menus and Menu Shortcuts

Java 1.1 adds support for popup menus to the AWT. The `PopupMenu` class is a subclass of `Menu`; menu items are added to it just as they are added to regular pulldown menus. A popup menu can be attached to an arbitrary AWT component, using the new `add()` method of the `Component` class. And, finally, a popup menu can be "popped up" by calling its `show()` method. (The menu pops itself down automatically.)

An application typically displays a popup menu when the user clicks a certain mouse button over the component that the menu is attached to. However, different platforms traditionally use different mouse buttons to display popup menus. You can use the new `isPopupTrigger()` method of `MouseEvent` to determine whether a given mouse click has the appropriate modifiers set to trigger the popup menu for the current platform.

Java 1.1 also adds support for menu shortcut keys. The new `MenuShortcut` class represents a menu shortcut key. An instance of this class may optionally be specified whenever you create a `MenuItem` object. Again, different platforms use different modifier keys to invoke menu shortcuts, so when you create a `MenuShortcut` object, you specify only the key in question (plus, optionally, the **Shift** key). The system translates this into a platform-dependent shortcut using **Ctrl**, **Alt**, or some other modifier key.

The example in Chapter 8, *New AWT Features* demonstrates both a popup menu and menu shortcuts.

Keyboard Focus Traversal

The ability to operate a GUI without using the mouse is an important feature of any windowing toolkit. The addition of menu shortcuts in Java 1.1 is an important step in this direction. Java 1.1 also adds rudimentary facilities for keyboard focus traversal (i.e., moving keyboard focus among the individual components in a window) using the **Tab** and **Shift-Tab** keys.

Under the new focus traversal scheme, components within a container are traversed in the order in which they were added to the container. (Note, however, that it is possible to override this order by specifying an explicit position within the container's component list for a new component as it is added to the container with the `add()` method.) Beyond adding components to their container in the order desired for traversal, nothing else is required of the programmer in order to make keyboard focus traversal work.

If you are creating a custom component that can accept keyboard focus, you should override the `isFocusTraversable()` method to return `true`. The component should call the `requestFocus()` method it inherits from `Component` when the user clicks on it or otherwise activates it. Finally, when a component receives focus, (i.e., when its `processFocusEvent()` method is invoked), it should provide some sort of visual indication, such as a highlighted border, that it has the focus.

Miscellaneous Improvements

The `SystemColor` class represents a color used by the desktop system. On some platforms, these colors may be dynamically updated while the system is running. The `SystemColor` class also implements quite a few constants that represent system colors for various GUI components. Thus, if you want your GUIs to match the desktop color scheme, you might create them using colors such as `SystemColor.menu` (the background color for menus) and `SystemColor.menuText` (foreground color for menus), for example.

The treatment of fonts has been changed and improved somewhat in Java 1.1. The use of the font names "TimesRoman," "Helvetica," and "Courier" is now discouraged. Instead, you should use "serif," "sansserif," and "monospaced"--these names convey the essential style of the font face, without specifying the exact font to be used. The font names "Dialog" and "DialogInput" are still supported in Java 1.1. An important reason for switching to generic font names is that Java can now display any Unicode character for which there is an appropriate font installed on the host system. The names "serif" and "sansserif" have meaning even when applied to non-Latin character sets, such as Japanese Kanji characters; the names "timesroman" and "helvetica" clearly do not. Another result of this fuller Unicode support is that the use of the "ZapfDingbats" font is also discouraged. Instead, regardless of what font you are using, you can simply encode these graphical symbols using Unicode characters between `\u2700` and `\u27ff`. (See Chapter 11, *Internationalization* for an example.) This improved support for Unicode makes it much easier to write internationalized programs in Java.

In Java 1.0, mouse cursors could only be specified for a `Frame`. In Java 1.1, every component can have a its own cursor, represented by the new `Cursor` object. There are new methods of `Component` for setting and querying the cursor. This change does not add any new predefined cursor images, nor does it add the ability to create custom cursors; it merely allows you to specify a cursor for any arbitrary component, and to do so in a more logical fashion.

The `ScrollPane` class is new in Java 1.1. It is a `Container` that makes it very easy to scroll a large component or GUI within a smaller visible area. Doing this in Java 1.0 required a custom container, and suffered from some serious performance problems. Chapter 8, *New AWT Features* shows the use of a `ScrollPane` object.

Another new feature is the ability to create "lightweight components." These are components and containers that do not have a native window of their own. In Java 1.0, custom components and containers had to be subclassed from `Canvas` or `Panel`. In Java 1.1, however, you can subclass `Component` and `Container` directly. Doing so creates a simpler component or container, without the overhead of a native window. It also allows you to create partially transparent components that appear non-rectangular.

Java 1.1 also includes several miscellaneous changes to clipping and image manipulation:

- The `Graphics` class defines a method to set an arbitrary clipping rectangle, even to one that is larger than the current clipping region. There is also a new method to query the current clipping region.

- `Graphics` also defines two new `drawImage()` methods that are more flexible than the existing drawImage() methods. These new methods allow arbitrary image cropping, scaling, and flipping.

- There are two new classes, ReplicateScaleFilter and AreaAveragingScaleFilter, that can be used to scale an image as it is loaded, and a new convenience method, Image.getScaledInstance(), to obtain a new Image object that contains a scaled version of some other Image.

- New methods have been added to the MemoryImageSource class that allow images generated from memory to be dynamically and efficiently updated, allowing a kind of image animation.

- New methods have been added to the PixelGrabber class to make it more efficient and flexible to use.

PREVIOUS

NEXT

4.4 Deprecated Features

Although you can use the old AWT event model in Java 1.1, it has been officially "deprecated," and its use in new software is discouraged. When you compile code that uses the 1.0 event model, you'll be made aware of this by the "deprecation warning" that the *javac* compiler issues. This warning notifies you that your code relies on methods or classes in the Java API that have been superseded by newer, preferred alternatives. If you compile using the `-deprecation` flag, *javac* provides a detailed warning about each use of a deprecated feature. You can simply ignore these warnings, but when time permits, the better approach is to update your code so that it no longer relies on deprecated features of the Java API. While it is not strictly true to say that deprecated features are "unsupported," they will almost certainly receive far less support in practice than the features that replace them.

The reason that the compiler is able to issue deprecation warnings at all is the addition of a new `@deprecated` tag to the documentation-comment syntax of Java 1.1. As you may be aware, comments that begin with the `/**` character sequence are treated specially in Java, and are used by the *javadoc* tool to automatically generate online documentation for packages, classes, methods, and fields. Prior to Java 1.1, the compiler ignored the contents of documentation comments. In Java 1.1, however, it scans these comments for the `@deprecated` tag. If it is found, the compiler marks the class, interface, constructor, method, or field following the comment as deprecated, and issues a warning when the deprecated feature is used.

The old AWT event-handling model is not the only Java 1.0 feature that has been deprecated in Java 1.1; merely the one you are most likely to encounter first. A number of common AWT component methods have been renamed, to follow a more regular naming scheme that fits the JavaBeans naming conventions. These methods can be invoked by the old name or the new, but if you use the old name, you'll be rewarded with a deprecation warning. Fortunately, in simple cases like this, it is trivial to write a script or program to mechanically convert from the old name to the new. Other areas of the Java API have been deprecated as well. You'll notice that a few of the input and output stream classes in the `java.io` package have been deprecated and superseded by "Reader" and "Writer" stream classes, for example.

PREVIOUS | HOME | NEXT
The New AWT Event Model | BOOK INDEX | Other AWT Improvements

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

4.3 The New AWT Event Model

The Java 1.1 change that will probably affect Java programmers the most is the new event processing model adopted for the AWT windowing and graphical user interface (GUI) toolkit. If you created applets or GUIs with Java 1.0, you know that it was necessary to subclass GUI components in order to handle events. This model worked okay for simple programs, but proved increasingly awkward as programs became more complex. Furthermore, with the development of the JavaBeans API, the AWT package needed an event model that would allow AWT GUI components to serve as beans. For these reasons, Java 1.1 defines a new model for dispatching and handling events.

As explained in Chapter 7, *Events*, the new event handling model is essentially a "callback" model. When you create a GUI component, you tell it what method or methods it should invoke when a particular event occurs on it (e.g., when the user clicks on a button or selects an item from a list). This model is very easy to use in C and C++ because those languages allow you to manipulate method pointers--to specify a callback, all you need to do is pass a pointer to the appropriate function. In Java, however, methods are not data and cannot be manipulated in this way. Only objects can be passed like this in Java, so to define a Java callback, you must define a class that implements some particular interface. Then, you can pass an instance of that class to a GUI component as a way of specifying the callback. When the desired event occurs, the GUI component invokes the appropriate method of the object you have specified.

As you might imagine, this new event handling model can lead to the creation of many small helper classes. (Sometimes these helper classes are known as "adaptor classes" because they serve as the interface between the body of an application and the GUI for that application. They are the "adaptors" that allow the GUI to be "plugged in" to the application.) This proliferation of small classes could become quite a burden, were it not for the introduction of inner classes, which, as noted above, allows this kind of special-purpose class to be nested and defined exactly where it is needed within your program.

Despite the major AWT event-handling changes, Java 1.1 does retain backwards compatibility with the event-handling model of Java 1.0. It is an all-or-nothing type of backwards compatibility, however--the two models are so different from each other that it is not really possible to mix them within the same application.

PREVIOUS | HOME | NEXT
Inner Classes | BOOK INDEX | Deprecated Features

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

4.2 Inner Classes

While the bulk of the changes in Java 1.1 are additions to the core Java API, there has also been a major addition to the language itself. The language has been extended to allow class definitions to be nested within other classes, and even to be defined locally, within blocks of code. Altogether, there are four new types of classes that can be defined in Java 1.1; these four new types are sometimes loosely referred to as "inner classes."

Chapter 5, *Inner Classes and Other New Language Features* explains in detail how to define and use each of the four new types of classes. As we'll see, inner classes are useful primarily for defining simple "helper" or "adaptor" classes that serve a very specific function at a particular place in a program, and are not intended to be general-purpose "top-level" classes. By using inner classes nested within other classes, you can place the definition of these special-purpose helper classes closer to where they are actually used in your programs. This makes your code clearer, and also prevents you from cluttering up the package namespace with small special purpose classes that are not of interest to programmers using your package. We'll also see that inner classes are particularly useful in conjunction with the new AWT event model in Java 1.1.

One important feature of inner classes is that no changes to the Java Virtual Machine are required to support them. When a Java 1.1 compiler encounters an inner class, it transforms the Java 1.1 source code in a way that converts the nested class to a regular top-level class. Once that transformation has been performed, the code can be compiled just as it would have been in Java 1.0.

4. What's New in Java 1.1

Contents:

Java 1.1 is a *huge* new release. The number of packages in the API has increased from 8 in Java 1.0 to 23 in Java 1.1, and the number of classes has more than doubled from 211 to 503. On top of these changes to the core Java class libraries, there have been some important changes to the language itself. Also, the JDK--the Java Development Kit from Sun--includes a number of new tools in version 1.1.

The new features of Java 1.1 include:

Inner classes

> Changes to the Java language itself to allow classes to be nested within each other, and within blocks of code.

Java Beans

> A framework for defining reusable modular software components.

Internationalization

A variety of new features that make it possible to write programs that run around the globe.

New event model

A new model for handling events in graphical user interfaces that should make it easier to create those interfaces.

Other new AWT features

The Java 1.1 AWT includes support for printing, cut-and-paste, popup menus, menu shortcuts, and focus traversal. It has improved support for colors, fonts, cursors, scrolling, image manipulation, and clipping.

Applets

JAR files allow all of an applet's files to be grouped into a single archive. Digital signatures allow trusted applets to run with fewer security restrictions. The HTML `<APPLET>` tag has new features.

Object serialization

Objects can now be easily "serialized" and sent over the network or written to disk for persistent storage.

Reflection

Java programs can now "reflect" upon themselves or upon an arbitrary class to determine the methods and fields defined by the class, the arguments passed to a method, and so on. The Reflection API also allows the invocation of methods specified by name.

Security

Java 1.1 includes a new package that supports digital signatures, message digests, key management, and access control lists.

Java Database Connectivity (JDBC)

A new package that allows Java programs to send SQL queries to database servers. It includes a "bridge" that allows it to inter-operate with existing ODBC database servers.

Remote Method Invocation (RMI)

> An interface that supports distributed Java applications in which a program running on one computer can invoke methods of Java objects that exist on a different computer.

These and other new features are summarized in the sections below. Many of them are also described in more detail elsewhere in this book.

4.1 Java 1.1 Package-by-Package

The packages and classes of the Java class library are interlinked and interdependent. Many of the major new features of Java 1.1 rely on classes from multiple packages in the Java API. Before we examine those new features in detail, therefore, we need to understand the big picture of Java 1.1. The paragraphs below discuss each of the 23 packages that constitute the core API for Java 1.1; they introduce the new packages and explain the changes to existing packages.

`java.applet`

> Despite the introduction of JAR files, digitally signed applets, and new attributes of the `<APPLET>` tag, the `java.applet` package has not changed in any significant way.

`java.awt`

> The `java.awt` package contains new classes and interfaces to support printing, popup menus, and menu shortcuts, and to improve support for layout management, cursors, scrolling, colors, and clipping. Several classes provide support for the new AWT event model, but most event support is contained in one of several new sub-packages of `java.awt`.

`java.awt.datatransfer`

> The classes and interfaces in this package define a generic framework for inter-application (and intra-application) data transfer. This package also includes classes to support a clipboard-based cut-and-paste data transfer model. In the future, this package may be extended to include support for data transfer through a drag-and-drop metaphor. One of the two underlying data transfer mechanisms supported by this package relies on the Object Serialization API of the `java.io` package.

`java.awt.event`

> This package defines the classes and interfaces of the new AWT event handling model. The

classes and interfaces of this package fall into three categories:

- Event classes--the classes that actually represent events.
- Event "listeners"--interfaces that define methods that must be implemented by objects interested in being notified when an event of a particular type occurs.
- Event "adaptors"--trivial no-op implementations of the event listener interfaces that are suitable for easy subclassing.

All the events and event listeners defined in this package extend the `EventObject` class or the `EventListener` interface defined in `java.util`.

`java.awt.image`

This package has two new image filter classes that implement improved image scaling. Changes have also been made to the `MemoryImageSource` and `PixelGrabber` classes.

`java.awt.peer`

The changes to this package for the most part simply reflect changes to `java.awt`. There are new interfaces that represent a platform-dependent popup menu and scrolling area, for example.

`java.beans`

This package constitutes the much-touted JavaBeans API for creating and using embeddable, reusable software components. The classes and interfaces in this package can be used at three different levels:

- To create application builder tools that programmers (or even non-programmers) can use to compose applications out of individual Java beans.
- To develop Java beans for use in such application builders.
- To develop applications (without using a builder tool) that use Java beans.

Most of the classes and interfaces of the package are for use by application builders or by developers of advanced beans. Programmers using beans or writing simple beans do not need to be familiar with most of the package.

Application builders that manipulate beans rely on the Reflection API defined in java.lang.reflect, and many beans take advantage of the Object Serialization API defined in the java.io package. The JavaBeans API uses the same event model that the Java 1.1 AWT does, and event-related classes and interfaces in this package are extensions of a class and an interface defined in java.util.

java.io

The java.io package has become by far the largest of the core Java packages. This is because Java 1.1 adds:

- A complete suite of new "character stream" classes to complement most of the existing "byte stream" input and output classes. These new "reader" and "writer" streams offer improved efficiency and support internationalization for textual input and output.
- New classes and interfaces to support object serialization.
- A number of new IOException types.

java.lang

This package has several new Exception and Error types, as well as new Byte, Short, and Void classes. With the addition of these new classes, all primitive Java data types (including the void type) have corresponding object types. This is important for the java.lang.reflect package, which defines the new Reflection API. In addition, the Class class has been greatly enhanced for use with the Reflection API. Class and ClassLoader have methods to locate "resources" associated with a class, such as images, audio files, Properties files, and so on. Resources are important for internationalization in Java 1.1.

java.lang.reflect

This new package enables a Java program to examine the structure of Java classes and to "reflect upon" its own structure. java.lang.reflect contains classes that represent the fields, methods, and constructors of a class, and enable a program to obtain complete information about any object, array, method, constructor, or field. The java.beans package relies heavily upon this package.

java.math

This new package contains only two classes, which support arithmetic on arbitrary-size integers and arbitrary-precision floating-point numbers. The BigInteger class also defines methods for modular arithmetic, primality testing, and other features required for cryptography.

`java.net`

The changes to the `java.net` package are quite low-level. They include the addition of multicast sockets, Unix-style socket options, and new exception types that provide finer granularity when handling networking exceptions.

`java.rmi`

This package defines the fundamental classes and interfaces used for Remote Method Invocation. Most of the classes in this package are exception types. Subpackages of `java.rmi` provide additional, more specialized functionality. When objects must be passed as arguments to remote methods, RMI relies on the object serialization functionality provided in the `java.io` package.

`java.rmi.dgc`

This small package defines the classes and interfaces required for distributed garbage collection (DGC).

`java.rmi.registry`

This is another small package that defines the classes and interfaces required for a Java client to look up a remote object by name or for a Java server to advertise the service it provides.

`java.rmi.server`

This package is the largest of the RMI packages and is at the heart of Remote Method Invocation. It defines the classes and interface that allow a Java program to create an object that can be used remotely by other Java programs.

`java.security`

This package contains the classes and interfaces that represent the fundamental abstractions of cryptographic security: public and private keys, certificates, message digests, and digital signatures. This package does not provide implementations of these abstractions; by design, the Java Security API is implementation independent. Java 1.1 does include a default implementation, but vendor-specific implementations may also be used in conjunction with this package. The default security implementation relies on the `BigInteger` class defined in the `java.math` package.

`java.security.acl`

This package defines high-level interfaces, and some exceptions, for manipulating access control lists.

`java.security.interfaces`

This package defines a few interfaces that are required for the Java Security API's implementation-independent design.

`java.sql`

This package is the Java Database Connectivity (JDBC) API. The classes and interfaces it contains allow Java programs to send SQL queries to databases and retrieve the results of those queries.

`java.text`

The classes and interfaces in this package are used for internationalization. The package includes classes for formatting dates, times, numbers, and textual messages in a manner appropriate for the default locale, or for any specified locale. It also includes classes for collating strings according to the rules of a given locale and iterating through the characters, words, and sentences of a string in a locale-specific manner.

`java.util`

As its name indicates, the `java.util` package contains miscellaneous utility classes. In Java 1.1, new classes have been added to this package to support the AWT and Java Beans event model, to define "locales" and "resource bundles" used for internationalization, and to manipulate dates, times, and time zones.

`java.util.zip`

This package implements classes for computing checksums on streams of data, and for compressing and archiving (and uncompressing and unarchiving) streams of data, using ZLIB compression library and ZIP and GZIP file formats.

PREVIOUS | HOME | NEXT
Summary | BOOK INDEX | Inner Classes

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

5.2 Nested Top-Level Classes and Interfaces

As explained above, a nested top-level class or interface is just like a regular package-member class or interface, except that, for convenience, it has been nested within another class or interface. Note that nested top-level classes and interfaces must be declared `static`. They can only be nested within other top-level classes and interfaces (i.e., they cannot be declared within inner classes), but they can be nested to any depth.

Example 5.1 shows how you might define a nested top-level "helper" interface. Note the use of the `static` keyword in the declaration of the interface. The example also shows how this interface is used both within the class that contains it and by external classes. Note the use of its hierarchical name in the external class.

Example 5.1: Defining and Using a Nested Top-Level Interface

```
public class LinkedList {
  // This nested top-level helper interface is defined as a static member.
  public interface Linkable {
    public Linkable getNext();
    public void setNext(Linkable node);
  }
  // The head of the list is a Linkable object.
  Linkable head;
  // Method bodies omitted.
  public void insert(Linkable node) { ... }
  public remove(Linkable node) { ... }
}
// This class defines a type of node that we'd like to use in
// a linked list.  Note the nested interface name in the implements clause.
class LinkableInteger implements LinkedList.Linkable
{
  // Here's the node's data and constructor.
  int i;
  public LinkableInteger(int i) { this.i = i; }
  // Here are the data and methods required to implement the interface.
  LinkedList.Linkable next;
  public LinkedList.Linkable getNext() { return next; }
  public void setNext(LinkedList.Linkable node) { next = node; }
}
```

The `import` statement can be used to import nested top-level classes and interfaces from the class that defines them, just as it can be used to import package member top-level classes and interfaces from the package that defines them. Example 5.2 shows a new definition of the `LinkableInteger` class from Example 5.1 that uses an `import` statement to allow it to refer to the `Linkable` interface by its simple, unqualified name (i.e., the name of the enclosing class is no longer needed).

Example 5.2: Importing a Static Member Class

```
import LinkedList.*;     // Or use import LinkedList.Linkable;
// Since we use an import statement, we can just type
// "Linkable" instead of "LinkedList.Linkable".
class LinkableInteger2 implements Linkable
{
  int i;
  public LinkableInteger2(int i) { this.i = i; }
  Linkable next;
  public Linkable getNext() { return next; }
  public void setNext(Linkable node) { next = node; }
}
```

Nested Top-Level Classes and .class Files

When you compile the *LinkedList.java* file shown in Example 5.1, you'll find that two class files are generated. The first is named *LinkedList.class*, as expected. The second, however, is named *LinkedList$Linkable.class*. The `$` in this name is automatically inserted by the Java 1.1 compiler.

The Java Virtual Machine knows nothing about nested top-level classes and interfaces or the various types of inner classes. Therefore, the Java compiler must convert these new types into standard, non-nested class files that the Java interpreter can understand. This is done through source-code transformations that insert `$` characters into nested class names. These source-code transformations may also insert hidden fields, methods, and constructor arguments into the affected classes. Unless you are writing a Java 1.1 compiler, however, you do not need to know the details of these source-code transformations, and you will typically not even notice them, except in the names of class files. [2]

[2] See the *Java Language Specification* if you want complete details on the source-code transformations performed by the Java 1.1 compiler to support inner classes.

PREVIOUS | HOME | NEXT
An Overview of Inner Classes | BOOK INDEX | Member Classes

5.3 Member Classes

While nested top-level classes are nested within a containing class, we've seen that they are still top-level classes, and that the nesting is purely a matter of organizational convenience. The same is not true of member classes, however. These classes are also nested, but they are not declared `static`, and in this case, the nesting is significant. The main features of member classes are:

- Every instance of a member class is internally associated with an instance of the class that defines or contains the member class.
- The methods of a member class can implicitly refer to the fields defined within the member class, as well as those defined by any enclosing class, including `private` fields of the enclosing class.

Like nested top-level classes, member classes are commonly used for helper classes required by the enclosing class. You use a member class instead of a nested top-level class when the member class requires access to the instance fields of the enclosing class, or when every instance of the helper class must refer to an instance of the enclosing class. When you use a member class, this reference from member class to enclosing class is implemented automatically for you.

Let's return to the `LinkedList` example that we saw above. Suppose we want to add the ability to loop through the elements in the linked list using the `java.util.Enumeration` interface. To do this, we define a separate class that implements this interface, and then add a method to the `LinkedList` class that returns an instance of the separate `Enumeration` class. Example 5.3 shows a typical Java 1.0-style implementation. [3]

[3] For simplicity, this example implements a very simple `Enumeration` class that is not thread-safe and that may return incorrect results if items are added to or removed from the list while an `Enumeration` object is in use.

Example 5.3: A LinkedList Enumerator, as a Separate Top-Level Class

```
import java.util.*;
public class LinkedList {
  // Our nested top-level interface.  Body omitted here...
  public interface Linkable { ... }
  // The head of the list.
  Linkable head;
  // Method bodies omitted here.
  public void addToHead(Linkable node) { ... }
  public Linkable removeHead() { ...  }
  // This method returns an Enumeration object for this LinkedList.
  public Enumeration enumerate() {
    return new LinkedListEnumerator(this);
  }
}
// This class defines the Enumeration type we use to list the elements in
// a LinkedList.  Note that each LinkedListEnumerator object is associated
// with a particular LinkedList object which is passed to the constructor.
class LinkedListEnumerator implements Enumeration {
  private LinkedList container;
  private LinkedList.Linkable current;
  public LinkedListEnumerator(LinkedList l) {
    container = l;
    current = container.head;
  }
  public boolean hasMoreElements() { return (current != null); }
  public Object nextElement() {
    if (current == null) throw new NoSuchElementException("LinkedList");
    Object value = current;
    current = current.getNext();
    return value;
  }
}
```

The point to notice about the `LinkedListEnumerator` class in Example 5.3 is that we must explicitly pass a `LinkedList` object to its constructor.

The problem with Example 5.3 is that `LinkedListEnumerator` is defined as a separate top-level class, when it really would be more elegant to define it as part of the `LinkedList` class itself. In Java 1.1, this is easily done using a member class, as shown in Example 5.4.

Example 5.4: A LinkedList Enumerator, as a Member Class

```
import java.util.*;
public class LinkedList
{
  // Our nested top-level interface.  Body omitted here...
  public interface Linkable { ... }
  // The head of the list.
  // This field could be private except for inner class-related compiler bugs.
  /* private */ Linkable head;
  // Method bodies omitted here.
  public void addToHead(Linkable node) { ... }
  public Linkable removeHead() { ...  }
  // This method returns an Enumeration object for this LinkedList.
  // Note: no LinkedList object is explicitly passed to the constructor.
  public Enumeration enumerate() { return new Enumerator(); }
  // And here is the implementation of the Enumeration interface,
  // defined as a private member class.
  private class Enumerator implements Enumeration {
    Linkable current;
    // Note: the constructor implicitly refers to 'head' in containing class.
    public Enumerator() { current = head; }
    public boolean hasMoreElements() {  return (current != null); }
    public Object nextElement() {
      if (current == null) throw new NoSuchElementException("LinkedList");
      Object value = current;
      current = current.getNext();
      return value;
    }
  }
}
```

In this version of the example, notice how the `Enumerator` class is nested within the `LinkedList` class. There is a real elegance to defining the helper class so close to where it is used by the containing class. [4] Of course, if you compiled this example you'd find that the `Enumerator` member class is compiled to a file named *LinkedList$Enumerator.class*--while one class is nested within the other in source code form, the same is not true of their compiled byte-code forms.

> [4] John Rose, the author of Sun's inner class specification, points out that the advantages of inner classes are not only their elegance, but also their "conciseness, expressiveness, and modularity." He says, "Even prosy-minded programmers who don't care a fig for prissy elegance...will appreciate the fact that they can define their adapter classes right next to the code that needs them, and that they won't have to manually wire the adapter to the main object...and that they won't have to pollute the name space of the package..."

Notice that no instance of the containing `LinkedList` class is passed to the `Enumerator()` constructor of the member class. A member class can refer to the members of its enclosing class implicitly; no explicit reference is necessary. Also note that the `Enumerator` class makes use of the `head` field of the enclosing class, even though `head` is declared `private`. Because the member class is defined within the enclosing class, it is "inside" the class as far as the definition of `private` fields and methods is concerned. In general, member classes, as well as local and anonymous classes can use the `private` fields and methods (and classes!) of their containing class. Similarly, a containing class can use the `private` fields, methods, and classes of the classes it contains. And any two classes that are enclosed by the same third class can access each other's `private` members. [5]

> [5] As noted earlier, however, bugs in *javac* in current versions of JDK 1.1 prevent this kind of access to `private` members. Until these bugs are fixed, you should use use package visibility instead of `private` visibility.

How Member Classes Work

The `Enumerator` member class of `LinkedList` can refer to the `head` field of `LinkedList` because every instance of a member class implicitly refers to an instance of the class that contains it--this is one of the fundamental features of member classes. It works because the compiler automatically inserts a `private` field in the member class to hold the required reference to the containing object. The compiler also automatically inserts a hidden argument to all constructors of a member class and passes the containing object as the value of this argument. [6] Once the compiler automatically adds this `private` field and constructor argument to the code in Example 5.4, you can see that we end up with code very much like what we saw in Example 5.3!

> [6] If you're curious about this, use *javap -p* to disassemble the class file of a member class. It shows you both the inserted `private` field and the extra constructor argument.

Because the Java Virtual Machine has no notion of inner classes, the Java 1.1 compiler also must take special action to allow member classes (and local and anonymous classes) to use the `private` fields and methods in their enclosing classes (and vice versa). When a `private` field or method is used in a way that is allowed in Java 1.1, but is not allowed by the Java interpreter, the compiler automatically inserts a special accessor method to allow the access.

New Syntax for Member Classes

The most important feature of a member class is that it can implicitly refer to fields in its containing object. We saw this in the following constructor from Example 5.4:

```
public Enumerator() { current = head; }
```

In this example, `head` is a field of the `LinkedList` class, and we assign it to the `current` field of the `Enumerator` class. What if we want to make these references explicit? We could try code like this:

```
public Enumerator() { this.current = this.head; }
```

This code does not compile, however. `this.current` is fine; it is an explicit reference to the `current` field in the newly created `Enumerator` object. It is the `this.head` expression that causes the problem--it refers to a field named `head` in the `Enumerator` object. There is no such field, so the compiler generates an error. To prevent this problem, Java 1.1 defines new syntax for explicitly referring to the containing instance of the current instance of a member class. If we wanted to be explicit in our constructor, we'd use the new syntax like this:

```
public Enumerator() { this.current = LinkedList.this.head; }
```

Similarly, we can use `LinkedList.this` to refer to the containing `LinkedList` object itself. In general, the syntax is *`classname`*`.this`, where *`classname`* is the name of a containing class. Note that member classes can themselves contain member classes, nested to any depth, and no member class can have the same name as any containing class. Thus, the use of the class name prepended to `this` is a perfectly general way to refer to any containing instance, as the following nested class example demonstrates:

```
public class A {
  public String name = "a";
  public class B {
    public String name = "b";
    public class C {
      public String name = "c";
      public void print_names() {
        System.out.println(name);        // "c": name field of class C
        System.out.println(this.name);   // "c": name field of class C
        System.out.println(C.this.name); // "c": name field of class C
        System.out.println(B.this.name); // "b": name field of class B
        System.out.println(A.this.name); // "a": name field of class A
      }
    }
  }
}
```

Another new piece of Java 1.1 syntax has to do with the way member classes are created. As we've seen, member classes work the way they do because the compiler automatically adds an argument to each member class constructor. This argument passes a reference to the containing instance to the newly created member instance. Now look again at our definition of the `enumerate()` method in Example 5.4:

```
public Enumeration enumerate() { return new Enumerator(); }
```

Nowhere in this `new` expression do we specify what the containing instance of the new `Enumerator` instance should be. In this case, the `this` object is used as the containing instance, which is what you would expect to happen. It is also what you want to occur in most cases. Nevertheless, Java 1.1 supports a new syntax that lets you explicitly specify the containing instance when creating an instance of a member class. We can make our method more explicit by using the following code:

```
public Enumeration enumerate() { return this.new Enumerator(); }
```

The syntax is *`containing_instance`* `.new`, where *`containing_instance`* is an expression that evaluates to an instance of the class that defines the desired member class.

Let's look at another example of this syntax. Recall that we declared the `Enumerator` member class to be `private` in Example 5.4. We did this for reasons of modularity and encapsulation. Suppose, however, that we had given `Enumerator` package visibility. In that case, it would be accessible outside of the `LinkedList` class, and we could instantiate our own copy of it. In order to create an instance of the member class `LinkedList.Enumerator`, however, we must specify the instance of `LinkedList` that contains it (and is implicitly passed to its constructor). Our code might look like the following:

```
// First create a linked list, and add elements to it (omitted).
LinkedList list = new LinkedList();
// Create an enumerator for the linked list.  Note the syntax of the
// 'new' expression.
Enumerator e = list.new Enumerator();
```

As a more complex example, consider the following lines of code used to create an instance of class `C` nested within an instance of class B nested within an instance of class A:

```
A a = new A();         // Create an instance of A.
A.B b = a.new B();     // Create an instance of B within the instance of A.
A.B.C c = b.new C();   // Create an instance of C within the instance of B.
c.print_names();       // Invoke a method of the instance of c.
```

Note that in the new expressions we name the class to be created relative to the instance that will contain it. That is, we say b.new C(), not b.new A.B.C() .

There is one final piece of new Java 1.1 syntax related to member class instantiation and initialization. Before we consider it, however, let me point out that you should rarely, if ever, need to use this syntax. It represents one of the pathological cases that snuck into the language along with all the elegant features of inner classes.

The new syntax for the new operator described above allows us to specify the containing instance that is passed to the constructor of a member class. There is one circumstance in which a constructor is invoked without the use of the new operator--when it is invoked with the super keyword from a subclass constructor. As strange as it may seem, it *is* possible for a top-level class to extend a member class. This means that the subclass does not have a containing instance, but that its superclass does. When the subclass constructor invokes the superclass constructor, it must specify the containing instance. It does this by prepending the containing instance and a period to the super keyword. If we had not declared our Enumerator class to be a private member of LinkedList, then we could subclass it. Although it is not clear why we would want to do so, we could write code like the following:

```
// A top-level class that extends a member class
class SpecialEnumerator extends LinkedList.Enumerator {
  // The constructor must take the LinkedList argument explicitly,
  // and then must pass it implicitly to the superclass constructor
  // using the new 'super' syntax.
  public SpecialEnumerator(LinkedList l) { l.super(); }
  // Here we override one or the other of the LinkedList.Enumerator
  // methods to have some kind of special behavior.
     . . .
}
```

Scope Versus Inheritance

Having noted that a top-level class can extend a member class, it is important to point out that with the introduction of member classes there are two entirely separate hierarchies that must be considered for any class. The first is the class hierarchy, from superclass to subclass, that defines the fields that a member class inherits. The second is the containment hierarchy, from containing class to contained class, that defines the fields that are in the scope of (and are therefore accessible to) the member class.

The two hierarchies are entirely distinct from each other, and it is important that you do not confuse them. This should not be a problem if you refrain from creating name conflicts where a field or method in a superclass has the same name as a field or method in a containing class. If such a name conflict does arise, however, the inherited field or method hides (i.e., takes precedence over) the field or method of the same name in the containing class or classes. This behavior is logical because when a class inherits a field or method, that field or method effectively becomes part of that class. Therefore, inherited fields and methods are in the scope of the class that inherits them, and take precedence over fields and methods by the same name in enclosing scopes.

Because this can be quite confusing, Java does not leave it to chance that you get it right! Whenever there is a name conflict between an inherited field or method and a field or method in a containing class, Java 1.1 requires that you *explicitly* specify which one you mean. For example, if a member class `B` inherits a field named `x` and is contained within a class `A` that also defines a field named `x`, you must use `this.x` to specify the inherited field, or `A.this.x` to specify the field in the containing class. An attempt to use the field `x` without an explicit specification of the desired instance causes a compilation error.

A good way to prevent confusion between the class hierarchy and the containment hierarchy is to avoid deep containment hierarchies. If a class is nested more than two levels deep, it is probably going to cause more confusion than it is worth. Furthermore, if a class has a deep class hierarchy (i.e., if it has many superclass ancestors), consider defining it as a top-level class rather than as a member class.

Restrictions on Member Classes

There are two important restrictions on member classes. First, they cannot have the same name as any containing class or package. This is an important rule, and one that is not shared by fields and methods.

Second, member classes, like all inner classes, cannot contain any static members (fields, methods, or classes). The justification for this restriction is that `static` fields, methods, and classes are "top level" constructs, and it is therefore reasonable that they only be defined at the "top level"--i.e., within top-level classes. Defining a static, top-level member within a non-top-level member class would simply promote confusion and bad programming style. It is clearer (and therefore required) to define all static members within a top-level class. (Which may be a nested top-level class, of course.)

Member Classes and Visibility Modifiers

A member class, like any member of a class, may be assigned one of three visibility levels: `public`, [7] `protected`, or `private`. If none of these visibility modifiers is specified, the default "package" visibility is used. However, as mentioned earlier, there have been no changes to the Java Virtual Machine to support member classes, and member classes are compiled to class files just like top-level classes are. As far as the Java interpreter is concerned, therefore, member classes, like top-level classes, can only have public or package visibility. Thus, a member class declared `protected` is actually treated as a public class, and a member class declared `private` actually has package visibility. While this is unfortunate, and is something you should be aware of, it does not constitute a major security flaw.

> [7] Because member classes are nested, and because of their nature as "helper" classes, it is unusual to ever declare a member class `public`.

Note that this does not mean that you should never declare a member class as `protected` or `private`. Although the interpreter cannot enforce these visibility attributes, the desired attributes are noted in the class file. This means that any conforming Java 1.1 compiler can enforce the visibility attributes and prevent the member classes from being accessed in unintended ways.

PREVIOUS
Nested Top-Level Classes and Interfaces

HOME
BOOK INDEX

NEXT
Local Classes

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

5.4 Local Classes

A local class is a class declared locally within a block of Java code. Typically, a local class is defined within a method, but local classes can also be defined within static initializers and instance initializers of a class. (Instance initializers are a new feature of Java 1.1 that we'll see later in this chapter.)

Because Java code can only appear within class definitions, all local classes are nested within a containing class. For this reason, local classes share many of the features of member classes. It is usually more appropriate, however, to think of them as an entirely separate kind of inner class. A local class has approximately the same relationship to a member class as a local variable has to an instance variable of a class. Local classes have the following interesting features:

- They are only visible and usable within the block of code in which they are defined.
- They can use any `final` local variables or method parameters that are visible from the scope in which they are defined. [8]

> [8] As we'll see shortly, one of the other language changes in Java 1.1 is that local variables and method parameters can be declared `final`.

The first defining characteristic of a local class is obviously that it is local. Like a local variable, the name of a local class is only valid within the block of code in which it was defined (and within any blocks nested with that block, of course). In many cases, this is not a problem. If a helper class is used only within a single method of a containing class, there is no reason that that helper cannot be coded as a local class rather than a member class. Example 5.5 shows how we can modify the `enumerate()` method of the `LinkedList` class we saw in previous examples so that it defines the `Enumerator` class as a local class, rather than as a member class. By doing this, we move the definition of the helper class even closer to the location where it is used and hopefully improve the clarity of the code even further.

Example 5.5: Defining and Using a Local Class

```
import java.util.*;
public class LinkedList
{
  // Our nested top-level interface.  Body omitted here...
  public interface Linkable { ... }
  // The head of the list
  /* private */ Linkable head;
  // Method bodies omitted here.
  public void addToHead(Linkable node) { ... }
  public Linkable removeHead() { ...  }
  // This method creates and returns an Enumeration object for this
  // LinkedList.
  public Enumeration enumerate()
  {
    // Here's the definition of Enumerator as a local class.
    class Enumerator implements Enumeration {
      Linkable current;
      public Enumerator() { this.current = LinkedList.this.head; }
      public boolean hasMoreElements() {  return (current != null); }
      public Object nextElement() {
        if (current == null) throw new NoSuchElementException("LinkedList");
        Object value = current;
        current = current.getNext();
        return value;
      }
    }
    // Create and return an instance of the Enumerator class defined here.
    return new Enumerator();
  }
}
```

How Local Classes Work

A local class is able to refer to fields and methods in its containing class for exactly the same reason that a member class can--it is passed a hidden reference to the containing class in its constructor, and it saves that reference away in a `private` field added by the compiler. Also like member classes, local classes can use `private` fields and methods of their containing class, because the compiler inserts any required accessor methods. [9]

> [9] As previously noted, bugs in the compiler prevent this from working correctly in the current versions of the JDK.

What makes local classes different from member classes is that they have the ability to refer to local variables from the scope that defines them. The crucial restriction on this ability, however, is that local classes can only reference local variables and parameters that are declared `final`. The reason for this is apparent from the implementation. A local class can use local variables because the compiler automatically gives the class a `private` instance field to hold a copy of each local variable the class refers to. The compiler also adds hidden arguments to each of the local class constructors to initialize these automatically created `private` fields to the appropriate values. So, in fact, a local class does not actually access local variables, but merely its own private copies of them. The only way this can work correctly is if the local variables are declared `final`, so that they are guaranteed not to change. With this guarantee, the local class can be assured that its internal copies of the variables are "in sync" with the real local variables.

New Syntax for Local Classes

In Java 1.0, only fields, methods, and classes may be declared `final`. The addition of local classes to Java 1.1 has required a liberalization in the use of the `final` modifier. It can now be applied to local variables, arguments to methods, and even to the exception parameter of a `catch` statement (Local classes can refer to `catch` parameters, just as they can refer to method parameters, as long as they are in scope and are declared `final`.) The meaning of the `final` modifier remains the same in these new uses: once the local variable or argument has been assigned a value, that value may not be changed.

Instances of local classes, like instances of member classes, have an enclosing instance that is implicitly passed to all constructors of the local class. Local classes can use the same new `this` syntax that member classes do to explicitly refer to members of enclosing classes. Local classes cannot use the new `new` and `super` syntax used by member classes, however.

Restrictions on Local Classes

Like member classes, and for the same reasons, local classes cannot contain fields, methods, or classes that are declared `static`. `static` members must be declared at the top level. Since nested interfaces are implicitly `static`, local classes may not contain nested interface definitions.

Another restriction on local classes is that they cannot be declared `public`, `protected`, `private`, or `static`. These modifiers are all used for members of classes, and are not allowed with local variable declarations; for the same reason they are not allowed with local class declarations.

And finally, a local class, like a member class, cannot have the same name as any of its enclosing classes.

Typical Uses of Local Classes

One common application of local classes is to implement "event listeners" for use with the new event model implemented by AWT and JavaBeans in Java 1.1. An event listener is a class that implements a specific "listener" interface. This listener is registered with an AWT component, such as a `Button`, or with some other "event source." When the `Button` (or other source) is clicked (or activated in some way), it responds to this event by invoking a method of the event listener. Since Java does not support method pointers, implementing a pre-defined interface is Java's way of defining a "callback" that is notified when some event occurs. The classes used to implement these interfaces are often called *adapter classes*. Working with adapter classes can become quite cumbersome when they all must be defined as top-level classes. But the introduction of local classes makes them much easier to use. Example 5.6 shows a local class used as an adapter class for handling GUI events. [10] This example also shows how a local class can make use of a method parameter that is declared `final`.

> [10] As we'll see in the next section, this adapter class could be written more succinctly as an anonymous class.

Example 5.6: Using a Local Class as an Adapter

```
import java.awt.*;
import java.awt.event.*;
// This class implements the functionality of some application.
public class Application {
  // These are some constants used as the command argument to doCommand().
  static final int save = 1;
  static final int load = 2;
  static final int quit = 3;
  // This method dispatches all the commands to the application.
  // Body omitted.
  void doCommand(int command) { }
  // Other methods of the application omitted...
}
// This class defines a GUI for the application.
class GUI extends Frame {
  Application app;  // holds a reference to the Application instance
  // Constructor and other methods omitted here...

  // This is a convenience method used to create menu items with
  // a specified label, keyboard shortcut, and command to be executed.
  // We declare the "command" argument final so the local
  // ActionListener class can refer to it.
  MenuItem createMenuItem(String label, char shortcut, final int command)
  {
    // First we create a MenuItem object.
    MenuItem item = new MenuItem(label, new MenuShortcut(shortcut));
    // Then we define a local class to serve as our ActionListener.
    class MenuItemListener implements ActionListener {
      // Note that this method uses the app field of the enclosing class
      // and the (final) command argument from its containing scope.
      public void actionPerformed(ActionEvent e) {
        app.doCommand(command);
      }
    }
    // Next, we create an instance of our local class that will be
    // the particular action listener for this MenuItem.
    ActionListener listener = new MenuItemListener();
    // Then we register the ActionListener for our new MenuItem.
    item.addActionListener(listener);
    // And return the item, ready to be inserted into a menu.
    return item;
  }
}
```

`createMenuItem()` is the method of interest in Example 5.6. It creates a `MenuItem` object with the specified label and keyboard shortcut, and then uses a local class to create an `ActionListener` object for the menu item.

The `ActionListener` is responsible for translating the selection of the `MenuItem` into an invocation of the application's `doCommand()` method. Note that the `command` method parameter is declared `final` so it can be used by the local class. Also note that the local class uses the `app` field of the class that contains it. Because this is an instance variable instead of a local variable, it does not need to be declared `final`.

A local class can use fields defined within the local class itself or inherited by the local class, `final` local variables and `final` parameters in the scope of the local class definition, and fields defined by or inherited by the containing class. Example 5.7 is a program that demonstrates these various fields and variables that are accessible to a local class. If you can make sense of the code, you have a good understanding of local classes.

Example 5.7: Fields and Variables Accessible to a Local Class

```
class A { protected char a = 'a'; }
class B { protected char b = 'b'; }
public class C extends A
{
  private char c = 'c';         // Private fields visible to local class.
  public static char d = 'd';
  public void createLocalObject(final char e)
  {
    final char f = 'f';
    int i = 0;                  // i not final; not usable by local class.
    class Local extends B
    {
      char g = 'g';
      public void printVars()
      {
        // All of these fields and variables are accessible to this class.
        System.out.println(g);  // (this.g) g is a field of this class.
        System.out.println(f);  // f is a final local variable.
        System.out.println(e);  // e is a final local argument.
        System.out.println(d);  // (C.this.d) d -- field of containing class.
        System.out.println(c);  // (C.this.c) c -- field of containing class.
        System.out.println(b);  // b is inherited by this class.
        System.out.println(a);  // a is inherited by the containing class.
      }
    }
    Local l = this.new Local(); // Create an instance of the local class
    l.printVars();              // and call its printVars() method.
  }

  public static void main(String[] args)
  {
    // Create an instance of the containing class, and invoke the
    // method that defines and creates the local class.
    C c = new C();
    c.createLocalObject('e');   // pass a value for final parameter e.
  }
}
```

PREVIOUS
Member Classes

HOME
BOOK INDEX

NEXT
Anonymous Classes

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

5.5 Anonymous Classes

An anonymous class is essentially a local class without a name. Instead of defining a local class and then instantiating it, you can often use an anonymous class to combine these two steps. Anonymous classes are very commonly used as adapter classes, like the one we saw in Example 5.6. As we'll see, anonymous classes (and their corresponding anonymous objects) are created through another extension to the syntax of the `new` operator. Thus, an anonymous class is defined by a Java expression, not a Java statement. This means that an anonymous class definition can be included within a larger Java expression such as an assignment or method call.

Example 5.8 shows the use of an anonymous class to implement the `java.io.FilenameFilter` interface. The implementation in this example is used to list only the files whose names end with *.java* from a specified directory.

Example 5.8: Implementing an Interface with an Anonymous Class

```
import java.io.*;
// A simple program to list all Java source code files in a directory
// specified as a command-line argument.
public class Lister
{
  public static void main(String[] args)
  {
    File f = new File(args[0]);  // f represents the specified directory.
    // List the files in the directory, using the specified filter object.
    // The anonymous class is defined as part of a method call expression.
    String[] list = f.list(new FilenameFilter() {
      public boolean accept(File f, String s) {
        return s.endsWith(".java");
      }
    });
    for(int i = 0; i < list.length; i++)  // output the list
      System.out.println(list[i]);
  }
}
```

As you can see in Example 5.8, the syntax for defining an anonymous class and creating an instance of that class is a regular `new` expression, followed by a class body definition in curly braces. If the name following the `new` keyword is the name of a class, the anonymous class is a subclass of the named class. If the name following `new` specifies an interface, as in our example, the anonymous class is an implementation of the interface. In this case, the anonymous class is always a subclass of `Object`--there is no way to specify an `extends` clause (or an `implements` clause). In addition, since this syntax creates an anonymous class, there is obviously no way to specify a name for the newly defined class.

Because an anonymous class has no name, it is not possible to define constructors for it within the class body. This is one of the basic restrictions on anonymous classes. Any arguments you specify between the parentheses following the superclass name in an anonymous class definition are implicitly passed to the superclass constructor. Anonymous classes are commonly used to subclass simple classes that do not take any constructor arguments, so the parentheses in the anonymous class definition syntax are often empty, as we saw in Example 5.8. When you use an anonymous class to implement an interface, the constructor argument list is always empty, of course, since the superclass constructor, `Object()`, expects no arguments.

One of the most elegant things about anonymous classes is that they allow you to define a one-shot class exactly where it is needed. Other important features are the succinctness of the syntax and the fact that no clutter is created by an unnecessary name.

Since anonymous classes have no names, you may wonder what the class files that represent them are named. If you compile the example shown in Example 5.8 you'll find that it produces two class files, *Lister.class* and *Lister$1.class*. Anonymous classes are given numbers to produce unique class file names based on the name of the containing class.

Anonymous Class Indentation and Formatting

The common indentation and formatting conventions we are familiar with for languages like Java and C begin to break down somewhat once we start placing class definitions within arbitrary expressions. Based on their experience with inner classes, the engineers at JavaSoft recommend the following formatting rules, which are followed in Example 5.8:

- The opening curly brace should not be on a line by itself; instead, it should follow the close parenthesis of the `new` operator. Similarly, the `new` operator should, when possible, appear on the same line as the assignment or other expression of which it is a part.

- The body of the anonymous class should be indented relative to the beginning of the line that contains the `new` keyword.

- The closing curly brace of an anonymous class should not be on a line by itself either; it should be followed by whatever tokens are required by the rest of the expression. Often this is a semicolon or a close parenthesis followed by a semicolon. This extra punctuation serves as a flag to the reader that this is not just an ordinary block of code, and makes it easier to make sense of anonymous classes in a code listing.

Anonymous Classes versus Local Classes

As we've discussed, an anonymous class behaves just like a local class, and is distinguished from a local class merely in the syntax used to define and instantiate it. In your own code, when you have to choose between using an anonymous class or using a local class, the decision often comes down to stylistic matters. You should use whichever syntax makes your code clearer. In general, you should consider using an anonymous class instead of a local class if:

- The class has a very short body.
- Only one instance of the class is needed.
- The class is used right after it is defined.
- The name of the class does not make your code any easier to understand.

When considering the use of an anonymous class, there are two important restrictions to bear in mind:

- An anonymous class has no name, and the syntax for defining one combines definition with instantiation. Thus, using an anonymous class instead of a local class is not appropriate if you need to create more than a single instance of the class each time the containing block is executed.
- It is not possible to define constructors for anonymous classes. If your class requires a constructor, you must use a local class instead. However, as we'll see, the Java syntax has been extended to allow "instance initializers" to be defined for a class; an instance initializer can often substitute for a constructor.

Consider Example 5.9, which shows the `Enumeration` class we've used throughout this chapter implemented as an anonymous class within the `enumerate()` method of the `LinkedList` class. Compare it with Example 5.5, which shows the same class implemented as a local class. In this case, because of the size of the class in question, using local class syntax probably results in code that is a little clearer. Still, if you are a fan of anonymous classes, you might choose to code the example in this way.

Example 5.9: An Enumeration Implemented with an Anonymous Class

```
import java.util.*;
public class LinkedList
{
  // Our nested top-level interface.  Body omitted here...
  public interface Linkable { ... }
  // The head of the list.
  /* private */ Linkable head;
  // Method bodies omitted here.
  public void addToHead(Linkable node) { ... }
  public Linkable removeHead() { ...  }
  // This method creates and returns an Enumeration object for this
  // LinkedList.
  public Enumeration enumerate()
  {
    // Instantiate and return this implementation.
    return new Enumeration() {
      Linkable current = head; // This used to be in the constructor, but
                               // anonymous classes don't have constructors.
      public boolean hasMoreElements() {  return (current != null); }
      public Object nextElement() {
        if (current == null) throw new NoSuchElementException("LinkedList");
        Object value = current;
        current = current.getNext();
        return value;
      }
    };  // Note the required semicolon.  It terminates the return statement.
  }
}
```

As another example, consider Example 5.10, which shows the `createMenuItem()` method of Example 5.6 rewritten to use an anonymous class instead of a local class. In this case, using an anonymous class is probably the right thing to do.

Example 5.10: Using an Anonymous Class as an Adapter

```
MenuItem createMenuItem(String label, char shortcut, final int command)
{
  // First we create a MenuItem object.
  MenuItem item = new MenuItem(label, new MenuShortcut(shortcut));
  // Then we register an anonymous ActionListener for our new MenuItem.
  item.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) { app.doCommand(command); }
  });
  // And return the item, ready to be inserted into a menu.
  return item;
}
```

New Java Syntax for Anonymous Classes

We've already seen examples of the syntax for defining and instantiating an anonymous class. More formally, we can write it as the following:

```
new class-name ( [ argument-list ] ) { class-body }
```

or

```
new interface-name () { class-body }
```

There is one additional new piece of syntax to support anonymous classes. As noted, anonymous classes cannot define constructors, since they do not have names. Therefore Java 1.1 adds a feature known as an *instance initializer*, which is similar to the static initializer of Java 1.0. Example 5.11 illustrates this new syntax.

Example 5.11: Java 1.1 Instance Initializers

```
public class InitializerDemo
{
  // This is an instance variable.
  public int[] array1;
  // This is an instance initializer.  It is an arbitrary block of code.
  // It runs for every new instance, after the superclass constructor
  // and before the class constructor, if any.  It can serve the same
  // function as a constructor with no arguments.
  {
    array1 = new int[10];
    for(int i = 0; i < 10; i++) array1[i] = i;
  }
  // The line below contains another instance initializer.  The instance
  // initializers for an object are run in the order in which they appear
  // in the class definition.
  int[] array2 = new int[10]; { for(int i=0; i<10; i++) array2[i] = i*2; }
  static int[] static_array = new int[10];
  // By contrast, the block below is a static initializer.  Note the static
  // keyword.  It runs only once, when the class is first loaded.
  static {
    for(int i = 0; i < 10; i++) static_array[i] = i;
  }
}
```

An instance initializer is simply a block of code inside curly braces that is embedded in a class definition, where a field or method definition normally appears. [11] A class (any class--not just anonymous classes) can have any number of instance initializers. The instance initializers and any variable initializers that appear in field definitions for the class are executed in the order they appear in the Java source code. These initializers are automatically run after the superclass constructor has returned but before the constructor, if any, of the current class runs.

[11] Notice that Java 1.1 now allows blocks of code to be inserted in class definitions, and local class definitions to be inserted in blocks of code.

Because an instance initializer can serve the same function as a no-argument constructor method, these initializers are particularly useful for anonymous classes. They can also be useful in non-anonymous classes. Instance initializers allow you to initialize an object's fields near the definition of those fields, rather than deferring that initialization to a constructor defined further away in the class. Used in this way, they can sometimes improve code readability.

Restrictions on Anonymous Classes

Because anonymous classes are just a type of local class, they share the same restrictions: an anonymous class cannot define any `static` fields, methods, or classes. Since nested interfaces are implicitly `static`, they cannot be defined within anonymous classes. Similarly, interfaces cannot be defined anonymously.

Anonymous classes, like local classes, cannot be `public`, `private`, `protected`, or `static`. In fact, the syntax for defining anonymous classes does not allow any modifiers to be specified.

PREVIOUS HOME NEXT
Local Classes BOOK INDEX Other New Features of Java 1.1

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

5.6 Other New Features of Java 1.1

While the addition of inner classes is by far the most important and far-reaching change to the Java language in Java 1.1, there have been several other changes to the language as well. They are:

- `final` local variables, method parameters and `catch` statement parameters
- Instance initializers
- "Blank finals"--`final` variable and field declarations without initializer expressions
- Anonymous arrays
- Class literals

As you can see, the first two items in this list are language changes that are related to, though not exclusively used by, the inner class changes. We covered `final` local variables and parameters in our discussion of local classes above. And we covered instance initializers in the discussion of anonymous classes. The following subsections discuss the remaining three changes.

Blank Finals

We've already seen that local variables, method parameters, and exception parameters of `catch` statements may be declared `final`. A related change is that `final` fields do not require initializers. In Java 1.0, any `final` field had to be initialized as part of the field declaration. In Java 1.1, this restriction has been relaxed. A field or local variable can be declared `final` without specifying an intial value as part of the declaration. These "blank finals," as they are called, must have a value assigned to them before they are ever used, of course. And, once a value has been assigned to a blank final, that value can never be changed. This allows you, for example, to use an instance initializer or a constructor to compute a value for a `final` field.

Blank finals are particularly useful in defining immutable data types. They allow a class to have immutable fields that are initialized based on run-time arguments to a constructor. Once assigned, these fields cannot be accidentally or maliciously changed.

Anonymous Arrays

In Java 1.0, you can create and initialize an array with code like the following:

```
int[] a = {1, 2, 3, 4, 5};
```

Unfortunately, this syntax is only allowed in initializer expressions that follow the declaration of a field or variable of array type. That is, you *cannot* write code like this:

```
int[] a;
a = {1, 2, 3, 4, 5};            // Error
int total = sum({1,2,3,4,5});  // Error
```

You cannot write code like that in Java 1.1 either, but you can write code using a similar array initializer syntax. When you use the `new` operator to create an array, you may omit the dimension that specifies the number of array elements to create and instead follow the empty square bracket pair (`[]`) with a list of initial values in curly braces. Such an expression creates an array large enough to hold all of the elements specified between the braces, and initializes the array to contain those elements. The elements in braces must all be of the type specified after the `new` keyword, of course.

Code that uses anonymous arrays looks like this:

```
int[] a;
a = new int[] {1, 2, 3, 4, 5};
int total = sum(new int[] {1, 2, 3, 4, 5});
System.out.println(new char[] {'h', 'e', 'l', 'l', 'o'});
```

As you can see, this new syntax allows you to create and initialize arrays without using a variable initializer, or without even assigning the array to a variable at all. That is why arrays created and initialized this way are called *anonymous arrays*.

Class Literals

Another major change in Java 1.1 is the introduction of the Reflection API in the `java.lang.reflect` package. As part of this new package, the `java.lang.Class` class has been broadened to represent not just Java classes, but all Java data types. In other words, there are now special

`Class` objects that represent each of the Java primitive types.

You can access these special `Class` objects through the `TYPE` field of each of the primitive wrapper classes. For example, the static variable `Boolean.TYPE` holds the `Class` object that represents the `boolean` data type. And the `Float.TYPE` static variable holds the `Class` object that represents the `float` data type. A new class `Void` has been added, and `Void.TYPE` represents the type `void`.

The changes described in the paragraph above are all changes to the Java class libraries, rather than changes to the Java language itself. The language change is a related one, however. In Java 1.1, you can obtain the `Class` object for any class or primitive type by following the class name or type name by a period and the `class` keyword. For example, `String.class` evaluates to the `Class` object that represents the `java.lang.String` class. Similarly, `int.class` evaluates to the special class object `Integer.TYPE` that represents the `int` data type.

In Java 1.0, it is much more cumbersome (and less efficient) to obtain a `Class` object--you have to use the static `Class.forName()` method, so you end up with expressions like:

```
Class c = Class.forName("java.util.Vector");
```

Where in Java 1.1 you can simply write:

```
Class c = java.util.Vector.class;
```

Remember that `class` is a keyword in Java, so this syntax does not simply constitute a reference to a static variable pre-defined in each class.

This new syntax is meant to simplify use of the new reflection facilities in Java 1.1. It is also necessary because using `Class.forName()` with inner classes requires knowledge of the way the compiler transforms the names of inner classes (i.e., where it replaces "`.`" with "`$`"). While compiler writers need to know about these transformation rules, Java programmers should not. Thus the new `.class` syntax provides a way to obtain a `Class` object that works with inner classes, as well as with top-level classes and interfaces.

6. Applets

Contents:

This chapter demonstrates the techniques of applet writing. It proceeds from a trivial "Hello World" applet to more sophisticated applets. Along the way, it explains how to:

- Draw graphics in your applet.
- Handle and respond to simple user input.
- Read and use values of applet parameters, allowing customization of an applet.
- Load and display images and load and play sounds.
- Package an applet and related files into a JAR file.
- Attach a digital signature to an applet.

Study the examples carefully. They are the important part of this chapter! You may find it useful to refer to the quick reference in Chapter 17, *The java.applet Package* while reading these examples.

Note that this chapter merely introduces the framework for writing applets. Applets, like other Java programs, use features from throughout the Java API. See Chapter 7, *Events*, in particular, for details on event processing in Java applets and applications.

6.1 Introduction to Applets

An applet, as the name implies, is a kind of mini-application, designed to be run by a Web browser, or in the context of some other "applet viewer." Applets differ from regular applications in a number of ways. One of the most important is that there are a number of security restrictions on what applets are allowed to do. An applet often consists of untrusted code, so it cannot be allowed access to the local file system, for example. The details of applet security and the restrictions placed on applets are discussed at the end of this chapter.

From a programmer's standpoint, one of the biggest differences between applets and applications is that applets do not have a `main()` method, or other single entry point from which the program starts running. Instead, to write an applet, you subclass the `Applet` class and override a number of standard methods. At appropriate times, under well-defined circumstances, the Web browser or applet viewer invokes the methods you have defined. The applet is not in control of the thread of execution; it simply responds when the browser or viewer tells it to. For this reason, the methods you write must take the necessary action and return promptly--they are not allowed to enter time-consuming (or infinite) loops. In order to perform a time-consuming or repetitive task, such as animation, an applet must create its own thread, over which it does have complete control.

The task of writing an applet, then, comes down to defining the appropriate methods. A number of these methods are defined by the `Applet` class:

`init()`

> Called when the applet is first loaded into the browser or viewer. It is typically used to perform applet initialization, in preference to a constructor method. (The Web browser doesn't pass any arguments to an applet's constructor method, so defining one isn't too useful.)

`destroy()`

> Called when the applet is about to be unloaded from the browser or viewer. It should free any resources, other than memory, that the applet has allocated.

`start()`

> Called when the applet becomes visible and should start doing whatever it is that it does. Often

used with animation and with threads.

`stop()`

Called when the applet becomes temporarily invisible, for example, when the user has scrolled it off the screen. Tells the applet to stop performing an animation or other task.

`getAppletInfo()`

Called to get information about the applet. Should return a string suitable for display in a dialog box.

`getParameterInfo()`

Called to obtain information about the parameters the applet responds to. Should return strings describing those parameters.

In addition to these `Applet` methods, there are a number of other methods, inherited from superclasses of `Applet`, that the browser invokes at appropriate times, and that an applet should override. The most obvious of these methods is `paint()`, which the browser or viewer invokes to ask the applet to draw itself on the screen. In Java 1.1, a related method is `print()`, which an applet should override if it wants to display itself on paper differently than it does on the screen. There are quite a few other methods that applets should override to respond to events. For example, if an applet wants to respond to mouse clicks, it should override `mouseDown()`. (As we'll see in Chapter 7, *Events*, however, there are other, preferred, ways to receive mouse events in Java 1.1.)

The `Applet` class also defines some methods that are commonly used by applets:

`getImage()`

Loads an image file from the network and returns an `Image` object.

`getAudioClip()`

Loads a sound clip from the network and returns an `AudioClip` object.

`getParameter()`

Looks up and returns the value of a named parameter, specified in the HTML file that refers to the applet with the `<PARAM>` tag.

`getCodeBase()`

Returns the base URL from which the applet class file was loaded.

`getDocumentBase()`

Returns the base URL of the HTML file that refers to the applet.

`showStatus()`

Displays a message in the status line of the browser or applet viewer.

`getAppletContext()`

Returns the `AppletContext` object for the applet. `AppletContext` defines the useful `showDocument()` method that asks the browser to load and display a new Web page.

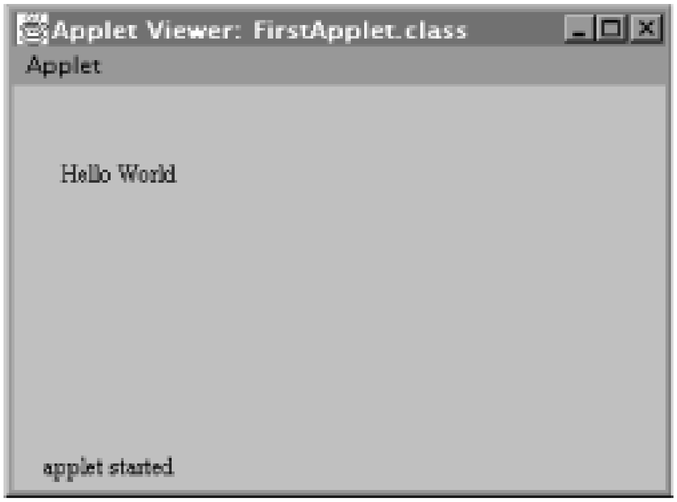
PREVIOUS | HOME | NEXT
Other New Features of Java 1.1 | BOOK INDEX | A First Applet

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

6.2 A First Applet

Figure 6.1 shows what is probably the simplest possible applet you can write in Java. Example 6.1 lists its code. This example introduces the `paint()` method, which is invoked by the applet viewer (or Web browser) when the applet needs to be drawn. This method should perform graphical output--such as drawing text or lines or displaying images--for your applet. The argument to `paint()` is a `Graphics` object that you use to do the drawing.

Figure 6.1: A simple applet



Hello World

applet started

Example 6.1: The Simplest Applet

```
import java.applet.*;   // Don't forget this import statement!
import java.awt.*;          // Or this one for the graphics!
public class FirstApplet extends Applet {
  // This method displays the applet.
  // The Graphics class is how you do all drawing in Java.
  public void paint(Graphics g) {
    g.drawString("Hello World", 25, 50);
  }
}
```

To display an applet, you need an HTML file that references it. Here is an HTML fragment that can be used with our first applet:

```
<APPLET code="FirstApplet.class" width=150 height=100>
</APPLET>
```

With an HTML file that references the applet, you can now view the applet with an applet viewer or Web browser. Note that the `width` and `height` attributes of this HTML tag are required. See Chapter 15, *Java-Related HTML Tags* for more details on the HTML `<APPLET>` tag. For most applet examples in this book, we show only the Java code and not the corresponding HTML file that goes with it. Typically, that HTML file contains a tag as simple as the one shown here.

6.3 Drawing Graphics

Example 6.2 shows a fancier version of our simple applet. As you can see from Figure 6.2 , we've made the graphical display more interesting. This applet also does all of its drawing in the `paint()` method. It demonstrates the use of `Font` and `Color` objects.

This example also introduces the `init()` method, which is used, typically in place of a constructor, to perform any one-time initialization that is necessary when the applet is first created. The `paint()` method may be invoked many times in the life of an applet, so this example uses `init()` to create the `Font` object that `paint()` uses.

Figure 6.2: A fancier applet

[Graphic: Figure 6-2]

Example 6.2: An Applet with Fancier Graphics

```
import java.applet.*;
import java.awt.*;
public class SecondApplet extends Applet {
  static final String message = "Hello World";
  private Font font;
  // One-time initialization for the applet
  // Note: no constructor defined.
  public void init() {
    font = new Font("Helvetica", Font.BOLD, 48);
  }
  // Draw the applet whenever necessary.  Do some fancy graphics.
  public void paint(Graphics g) {
    // The pink oval
    g.setColor(Color.pink);
    g.fillOval(10, 10, 330, 100);
    // The red outline. Java doesn't support wide lines, so we
    // try to simulate a 4-pixel wide line by drawing four ovals.
    g.setColor(Color.red);
    g.drawOval(10, 10, 330, 100);
    g.drawOval(9, 9, 332, 102);
    g.drawOval(8, 8, 334, 104);
    g.drawOval(7, 7, 336, 106);
    // The text
    g.setColor(Color.black);
    g.setFont(font);
    g.drawString(message, 40, 75);
  }
}
```

PREVIOUS | HOME | NEXT
A First Applet | BOOK INDEX | Handling Events

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | Y | Z

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | Y | Z

HOME

Exploring Java

Exploring Java

By Patrick Niemeyer & Joshua Peck; 1-56592-184-271-9, 500 pages (est.)
2nd Edition July 1997 (est.)

Table of Contents

Search the text of *Exploring Java*.

Java Fundamental Classes Reference

Java Fundamental Classes Reference

By Mark Grand and Jonathan Knudsen; 1-56592-241-7, 1152 pages
1st Edition May 1997

Table of Contents

Part I: Using the Fundamental Classes

This part of the book, Chapters 2 through 10, provides a brief guide to many of the features of the fundamental classes in Java. These tutorial-style chapters are meant to help you learn about some of the basic functionality of the Java API. They provide short examples where appropriate that illustrate the use of various features.

Part II: Reference

This part of the book is a complete reference to all of the fundamental classes in the core Java API. The material is organized alphabetically by package, and within each package, alphabetically by class. The reference page for a class tells you everything you need to know about using that class. It provides a detailed description of the class as a whole, followed by a complete description of every variable, constructor, and method defined by the class.

Part III: Appendixes

This part provides information about the Unicode 2.0 standard and the UTF-8 encoding used by Java.

Search the text of *Java Fundamental Classes Reference*.

Java AWT Reference

By John Zukowski; 1-56592-240-9, 1074 pages
1st Edition April 1997

Table of Contents

Search the text of *Java AWT Reference*.

Java Language Reference

Java Language Reference

By Mark Grand; 1-56592-326-X, 450 pages (est.)
2nd Edition July 1997 (est.)

Table of Contents

Search the text of *Java Language Reference*.

---

15. Java-Related HTML Tags

Contents:

This chapter explains what you need to know about HTML to work with Java applets.

15.1 The <APPLET> Tag

A Java applet is included in a Web page with the `<APPLET>` tag, which has the following syntax. Items in brackets (`[]`) are optional.

```
<APPLET
    CODE = applet-filename
    WIDTH = pixel-width
    HEIGHT = pixel-height
    [OBJECT = serialized-applet-filename]
    [ARCHIVE = jar-file-list]
    [CODEBASE = applet-url]
    [ALT = alternate-text]
    [NAME = applet-name]
    [ALIGN = alignment]
    [VSPACE = vertical-pixel-space]
    [HSPACE = horizontal-pixel-space]
>
[<PARAM NAME = parameter VALUE = value>]
[<PARAM NAME = parameter VALUE = value>]
    ...
[alternate-text]
```

```
</APPLET>
```

`APPLET`

The `<APPLET>` tag specifies an applet to be run within a Web document. A Web browser that does not support Java and does not understand the `<APPLET>` tag ignores this tag and any related `<PARAM>` tags, and simply displays any *`alternate-text`* that appears between `<APPLET>` and `</APPLET>`. A browser that does support Java runs the specified applet, and does *not* display the *`alternate-text`*.

`CODE`

This required attribute specifies the file that contains the compiled Java code for the applet. It must be relative to the `CODEBASE` if that attribute is specified, or relative to the current document's URL. It must not be an absolute URL. In Java 1.1, this attribute can be replaced with an `OBJECT` attribute.

`WIDTH`

This attribute specifies the initial width, in pixels, that the applet needs in the browser's window. It is required.

`HEIGHT`

This attribute specifies the initial height, in pixels, that the applet needs in the browser's window. It is required.

`OBJECT`

As of Java 1.1, this attribute specifies the name of a file that contains a serialized applet that is to be created by deserialization. An applet specified in this way does not have its `init()` method invoked, but does have its `start()` method invoked. Thus, before an applet is saved through serialization, it should be initialized, but should not be started, or, if started, it should be stopped. An applet must have either the `CODE` or `OBJECT` attribute specified, but not both.

`ARCHIVE`

As of Java 1.1, this attribute specifies a comma-separate list of JAR (Java Archive) files that are "preloaded" by the Web browser or applet viewer. These archive files may contain Java class files, images, sounds, properties, or any other resources required by the applet. The Web browser or applet viewer searches for required files in the archives before attempting to load them over the network.

`CODEBASE`

This optional attribute specifies the base URL (absolute or relative) of the applet to be displayed. This should be a directory, not the applet file itself. If this attribute is unspecified, then the URL of the current document is used.

`ALT`

This optional attribute specifies text that should be displayed by browsers that understand the `<APPLET>` tag but do not support Java.

`NAME`

This optional attribute gives a name to the applet instance. Applets that are running at the same time can look each other up by name and communicate with each other.

`ALIGN`

This optional attribute specifies the applet's alignment on the page. It behaves just like the `ALIGN` attribute of the `<IMG>` tag. Its allowed values are: `left`, `right`, `top`, `texttop`, `middle`, `absmiddle`, `baseline`, `bottom`, and `absbottom`.

`VSPACE`

This optional attribute specifies the margin, in pixels, that the browser should put above and below the applet. It behaves just like the `VSPACE` attribute of the `<IMG>` tag.

`HSPACE`

This optional attribute specifies the margin, in pixels, that the browser should put on either side of the applet. It behaves just like the `HSPACE` attribute of the `<IMG>` tag.

PREVIOUS
Working with System Properties
HOME
BOOK INDEX
NEXT
The <PARAM> Tag

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

14.2 Working with System Properties

The system property list is not a static. Other properties can be added to it (and read from it) to allow easy customization of application behavior.

Specifying Individual Properties

You can specify individual properties to be inserted into the system properties list with the `-D` option to the Java interpreter. For example, you might invoke a program like this:

```
% java -Dgames.tetris.level=9 -Dgames.tetris.sound=off games.tetris
```

Note the format of each property specification: the property name, which is often a hierarchical one, followed by an equals sign, followed by the property value. A property value may include spaces, but any `-D` option specifying a property value containing spaces would have to be quoted when passed to *java*, of course.

If you write a platform-specific script file to invoke your Java application, you can use this `-D` option to translate native environment variable settings into Java system properties. On a Unix system, for example, such a script might look like this:

```
#!/bin/sh
exec java -Dgames.tetris.level=$TETRIS_LEVEL \
          -Dgames.tetris.sound=$TETRIS_SOUND \
          games.tetris
```

Using Property Files

Properties in Java are represented by the `java.util.Properties` object, which is essentially a hash table that can be read from and written to a file. A program need not limit itself to the use of system properties. It can also read in its own files of properties to support user preferences and user customization.

For example, when the *appletviewer* program starts up, it reads the properties from the *lib/appletviewer.properties* file in the JDK distribution. This file contains the various messages that `appletviewer` displays to the user and provides the flexibility to translate those messages into other languages. The following lines are an excerpt from *appletviewer.properties*:

```
#
# Applet status messages
#
appletloader.nocode=APPLET tag missing CODE parameter.
appletloader.notfound=load: class %0 not found.
appletloader.nocreate=load: %0 can't be instantiated.
```

Note that comments in a properties file start with #, and that the property specification format is the same as with the `-D` option. Also note that these property values do contain spaces. Some of them also contain the `%` substitution character and are intended for use with the `java.text.MessageFormat` class.

When reading in a file of properties, it can be convenient to merge those properties with the standard system properties, so that the program need only look in one place to find both loaded properties and standard properties (and properties specifed wiht the `-D` option). Every `Properties` object can have a "parent" properties object; if a property is not found in the `Properties` object, it is searched for in the parent. Thus, it is possible to merge in properties with code like this:

```
// Create a new Properties object with system props as its parent.
Properties props = new Properties(System.getProperties());
// Load a file of properties into it.  We may get an exception here...
props.load(new BufferedInputStream(new FileInputStream(propsfilename)));
// Set these new combined properties as the system properties.
System.setProperties(props);
```

Specifying Font Properties

As noted above, a program can read the string value of a system property with the `System.getProperty()` method. There are also some convenience methods that read a property value and convert that value into some other type of object. One of these convenience methods is `Font.getFont()`. This method reads the value of a named property and attempts to parse it into a font specification. The font specification syntax it uses is:

```
name[-style][-size]
```

The*`style`* should be `italic`, `bold` or `bolditalic`. If omitted, a plain font is used. The `size` should be an integer that specifies the font size in points. If omitted, 12-point is used. If the *`style`* is specified, the *`size`* must also be specified. For example, you might specify font properties like the following:

```
games.tetris.titlefont=sanserif-bolditalic-48
games.tetris.mainfont=serif-14
games.tetris.scorefont=monospaced
```

Specifying Color Properties

`Color.getColor()` is another convenience routine that reads a system property and converts it into a `Color` object. To specify a color property, you specify the color as an integer value, typically as a hexadecimal value in the format `0x`*RRGGBB*. For example:

```
# A green foreground
games.tetris.foreground=0x00FF00
# A light gray background
games.tetris.background=0xD0D0D0
```

PREVIOUS | HOME | NEXT
Standard System Properties | BOOK INDEX | Java-Related HTML Tags

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

15.3 An Example HTML File

Example 15.1 shows an applet embedded in an HTML file. This file is a lightly edited version of one of the demos that ships with the JDK. Notice the use of the <PARAM> tags to supply arguments to the applet.

Example 15.1: Example HTML Page Containing an Applet

```
<HTML>
<HEAD>
<TITLE>The Animator Applet (1.1) - example 1</TITLE>
</HEAD>
<BODY>
<H1>The Animator Applet (1.1) - example 1</H1>
<APPLET CODE=Animator.class WIDTH=460 HEIGHT=160>
<PARAM NAME=imagesource VALUE="images/Beans">
<PARAM NAME=backgroundcolor VALUE="0xc0c0c0">
<PARAM NAME=endimage VALUE=10>
<PARAM NAME=soundsource VALUE="audio">
<PARAM NAME=soundtrack VALUE=spacemusic.au>
<PARAM NAME=sounds VALUE="1.au|2.au|3.au|4.au|5.au|6.au|7.au|8.au|9.au|0.au">
<PARAM NAME=pause VALUE=200>
</APPLET>
</BODY>
</HTML>
```

28.24 java.net.URLEncoder (JDK 1.0)

This class defines a single static method which is used to convert a string to its URL-encoded form. That is, spaces are converted to "+", and non-alphanumeric characters other than underscore are output as two hexadecimal digits following a percent sign. Note that this technique only works for 8-bit characters. This method is used to "canonicalize" a URL specification so that it uses only characters from an extremely portable subset of ASCII which can be correctly handled by computers around the world.

```
public class URLEncoder extends Object {
    // No Constructor
    // Class Methods
            public static String encode(String s);
}
```

13.3 Operators

Operators lists the operators of the Java language, along with their precedence, operand types, and associativity.

Table 13.3: Java Operators

Prec.	Operator	Operand Type(s)	Assoc.	Operation Performed
1	++	arithmetic	R	pre-or-post increment (unary)
	--	arithmetic	R	pre-or-post decrement (unary)
	+, -	arithmetic	R	unary plus, unary minus
	~	integral	R	bitwise complement (unary)
	!	boolean	R	logical complement (unary)
	(*type*)	any	R	cast
2	*, /, %	arithmetic	L	multiplication, division, remainder
3	+, -	arithmetic	L	addition, subtraction
	+	string	L	string concatenation
4	<<	integral	L	left shift
	>>	integral	L	right shift with sign extension
	>>>	integral	L	right shift with zero extension
5	<, <=	arithmetic	L	less than, less than or equal
	>, >=	arithmetic	L	greater than, greater than or equal
	instanceof	object, type	L	type comparison
6	==	primitive	L	equal (have identical values)
	!=	primitive	L	not equal (have different values)

	==	object	L	equal (refer to same object)
	!=	object	L	not equal (refer to different objects)
7	&	integral	L	bitwise AND
	&	boolean	L	boolean AND
8	^	integral	L	bitwise XOR
	^	boolean	L	boolean XOR
9	|	integral	L	bitwise OR
	|	boolean	L	boolean OR
10	&&	boolean	L	conditional AND
11	||	boolean	L	conditional OR
12	?:	boolean, any, any	R	conditional (ternary) operator
13	=	variable, any	R	assignment
	*=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, |=	variable, any	R	assignment with operation

Operator precedence controls the order in which operations are performed. Consider the following example:

```
w = x + y * z;
```

The multiplication operator `*` has a higher precedence than the addition operator `+`, so the multiplication is performed before the addition. Furthermore, the assignment operator `=` has the lowest precedence of any operator, so the assignment is done after all the operations on the right-hand side are performed. Operators with the same precedence (like addition and subtraction) are performed in order according to their associativity (usually left-to-right). Operator precedence can be overridden with the explicit use of parentheses. For example:

```
w = (x + y) * z;
```

The associativity of an operator specifies the order that operations of the same precedence are performed in. In Table 13.3 a value of L specifies left-to-right associativity, and a value of R specifies right-to-left associativity. Left-to-right associativity means that operations are performed left-to-right. For example:

```
w = x + y + z;
```

is the same as:

```
w = ((x + y) + z);
```

because the addition operator has left-to-right associativity. On the other hand, the following expressions:

```
x = ~-~y;
q = a?b:c?d:e?f:g;
```

are equivalent to:

```
x = ~(-(~y));
q = a?b:(c?d:(e?f:g));
```

because the unary operators and the ternary conditional `?:` operator have right-to-left associativity.

Java operators are basically identical to C operators, except for these differences:

- The `+` operator applied to `String` values concatenates them. If only one operand of `+` is a `String`, the other one is converted to a string. The conversion is done automatically for primitive types and by calling the `toString` method of non-primitive types.

- Java does not have the comma operator like C does. It does, however, simulate this operator in the limited context of the `for` loop initialization and increment expressions.

- Since all Java integral types are signed, the `>>` operator always does a signed right shift, filling in high bits with the sign bit of the operand. The new `>>>` operator performs an unsigned right shift, filling in high bits of the shifted value with zero bits.

- The & and `|` operators perform bitwise AND and OR operations on integral operands, and perform logical AND and OR operators on `boolean` operands. && and `||` also perform logical AND and OR on `boolean` operands, but do not evaluate the right-hand operand, if the result of the operation is fully determined by the left-hand operand.

- The `instanceof` operator returns `true` if the object on the left-hand side is an instance of the class or implements the interface on the right-hand side. Otherwise it returns `false`. If the left-hand side is `null`, it returns `false`.

PREVIOUS | HOME | NEXT
Character Escape Sequences | BOOK INDEX | Modifiers

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

13.6 Java Documentation Comment Syntax

The Java language supports special "doc comments," which begin with `/**` and end with `*/`. These comments are not actually treated specially by the compiler, but can be extracted and automatically turned into HTML documentation by the *javadoc* program.

Because the lines of a doc comment are embedded within a Java comment, any leading spaces and asterisks (`*`) are stripped from each comment line before processing. A doc comment may contain HTML markup tags, such as <PRE> and <TT> for code usage examples, but should not contain HTML structural tags such as <H2> or <HR>. Doc comments should immediately precede the declaration of the class, field, or method that they are associated with. The first sentence of a doc comment should be a summary sentence, suitable for display on its own. The following sentences may document the feature in more detail.

Following the initial summary sentence and any additional documentation, a doc comment may use special tags, which all begin with the `@` character and allow *javadoc* to provide additional formatting for the documentation. The available tags are listed below. When you use a special *javadoc* tag, it must be the first thing on its line within the doc comment. The text that follows a tag may span more than one line, and continues until the next *javadoc* tag is encountered or until the comment ends. If you use more than one tag of the same type, they should be on subsequent lines. For example, a class with multiple authors, or a method with multiple arguments would use multiple `@author` or `@param` tags.

`@see` *classname*

> This tag adds a "See Also:" entry to the documentation that contains a hyperlink to the specified class. It may be used before classes, methods, or fields.

`@see` *full-classname*

> This tag adds a "See Also:" entry to the documentation that contains a hyperlink to the specified class. It may be used before classes, methods, or fields.

`@see` *full-classname#method-name*

This tag adds a "See Also:" entry to the documentation that contains a hyperlink to the specified method of the specified class. It may be used before classes, methods, or fields.

`@version` *text*

This tag adds a "Version:" entry containing the specified text to the documentation. May only be used before a class definition. *javadoc* ignores this tag unless the `-version` command-line argument is specified.

`@author` *text*

This tag adds an "Author:" entry containing the specified text to the documentation. May only be used before a class definition. *javadoc* ignores this tag unless the `-author` command-line argument is specified.

`@param` *parameter-name description*

This tag adds the specified parameter and its specified description to the "Parameters:" section of the current method. If the description is longer than one line, it may be continued on the next. May only be used before a method definition.

`@return` *description*

Adds a "Returns:" section containing the specified description to the documentation. May only be used before a method definition.

`@exception` *full-classname description*

Adds a "Throws:" entry to the documentation. The entry contains the specified class name of the exception and the description specified, which should explain the significance of the exception. May only be used before a method definition.

`@deprecated` *explanation*

As of Java 1.1, this tag specifies that the following class, method, or field has been deprecated. *javac* notes this information in the class file it produces and issues a warning when a program uses the deprecated feature. *javadoc* adds a "Deprecated" entry to the documentation that includes the specified explanation.

`@since` *version*

> As of Java 1.1, this undocumented tag is used to specify when the class, method, or field that follows it was added to the API. It should be followed by a version number or other version specification. The JDK 1.1 version of *javadoc* appears to ignore this tag.

PREVIOUS	HOME	NEXT
Reserved Words	BOOK INDEX	System Properties

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

1. Yet Another Language?

Contents:

The greatest challenges and most exciting opportunities for software developers today lie in harnessing the power of networks. Applications created today, whatever their intended scope or audience, will almost certainly be run on machines linked by a global network of computing resources. The increasing importance of networks is placing new demands on existing tools, and fueling the demand for a rapidly growing list of completely new kinds of applications.

We want software that works--consistently, anywhere, on any platform--and that plays well with other applications. We want dynamic applications that take advantage of a connected world, capable of accessing disparate and distributed information sources. We want truly distributed software that can be extended and upgraded seamlessly. We want intelligent applications--like autonomous agents that can roam the Net for us, ferreting out information and serving as electronic emissaries. We know, at least to some extent, what we want. So why don't we have it?

The problem has been that the tools for building these applications have fallen short. The requirements of speed and portability have been, for the most part, mutually exclusive, and security has largely been ignored or misunderstood. There are truly portable languages, but they are mostly bulky, interpreted, and slow. These languages are popular as much for their high level functionality as for their portability. And there are fast languages, but they usually provide speed by binding themselves to particular platforms, so they can meet the portability issue only half way. There are even a few recent safe languages, but they are primarily offshoots of the portable languages and suffer from the same problems.

1.1 Enter Java

The Java programming language, developed at Sun Microsystems under the guidance of Net luminaries James Gosling and Bill Joy, is designed to be a machine-independent programming language that is both safe enough to traverse networks and powerful enough to replace native executable code. Java addresses the issues raised here and may help us start building the kinds of applications we want.

Right now, most of the enthusiasm for Java stems from its capabilities for building embedded applications for the World Wide Web; these applications are called *applets*. This book will teach you how to build applets. But there is more to Java than applets, and I'll also try to show you the "more." The book will also show you how to use the tools of Java to accomplish real programming tasks, such as building networked applications and creating functional user interfaces. By the end of the book, you will be able to use these tools to build powerful Java applets and standalone applications.

Java's Origins

The seeds of Java were planted in 1990 by Sun Microsystems patriarch and chief researcher, Bill Joy. Since Sun's inception in the early '80s, it has steadily pushed one idea: "The network is the computer." At the time though, Sun was competing in a relatively small workstation market, while Microsoft was beginning its domination of the more mainstream, Intel-based PC world. When Sun missed the boat on the PC revolution, Joy retreated to Aspen, Colorado, to work on advanced research. He was committed to accomplishing complex tasks with simple software, and founded the aptly named Sun Aspen Smallworks.

Of the original members of the small team of programmers assembled in Aspen, James Gosling is the one who will be remembered as the father of Java. Gosling first made a name for himself in the early '80s as the author of Gosling Emacs, the first version of the popular Emacs editor that was written in C and ran under UNIX. Gosling Emacs became popular, but was soon eclipsed by a free version, GNU Emacs, written by Emacs's original designer. By that time, Gosling had moved on to design Sun's NeWS window system, which briefly contended with the X Window System for control of the UNIX graphic user interface (GUI) desktop in 1987. While some people would argue that NeWS was superior to X, NeWS lost out because Sun kept it proprietary and didn't publish source code, while the primary developers of X formed the X Consortium and took the opposite approach.

Designing NeWS taught Gosling the power of integrating an expressive language with a network-aware windowing GUI. It also taught Sun that the Internet programming community will refuse to accept proprietary standards, no matter how good they may be. The seeds of Java's remarkably permissive licensing scheme were sown by NeWS's failure. Gosling brought what he had learned to Bill Joy's nascent Aspen project, and in 1992, work on the project led to the founding of the Sun subsidiary, FirstPerson, Inc. Its mission was to lead Sun into the world of consumer electronics.

The FirstPerson team worked on developing software for information appliances, such as cellular phones and personal digital assistants (PDA). The goal was to enable the transfer of information and real-time applications over cheap infrared and packet-based networks. Memory and bandwidth limitations dictated small and efficient code. The nature of the applications also demanded they be safe and robust. Gosling and his teammates began programming in C++, but they soon found themselves confounded by a language that was too complex, unwieldy, and insecure for the task. They decided to start from scratch, and Gosling began working on something he dubbed "C++ minus minus."

With the floundering of the Apple Newton, it became apparent that the PDA's ship had not yet come in, so Sun shifted FirstPerson's efforts to interactive TV (ITV). The programming language of choice for ITV set-top boxes was the near ancestor of Java, a language called Oak. Even with its elegance and ability to provide safe interactivity, Oak could not salvage the lost cause of ITV. Customers didn't want it, and Sun soon abandoned the concept.

At that time, Joy and Gosling got together to decide on a new strategy for their language. It was 1993, and the explosion of interest in the Internet, and the World Wide Web in particular, presented a new opportunity. Oak was small, robust, architecture independent, and object oriented. As it happens, these are also the requirements for a universal, network-savvy programming language. Sun quickly changed focus, and with a little retooling, Oak became Java.

Future Buzz?

I don't think it's overdoing it to say that Java has caught on like wildfire. Even before its first official release, while Java was still a nonproduct, nearly every major industry player jumped on the Java bandwagon. Java licensees include Microsoft, Intel, IBM, and virtually all major hardware and software vendors.

As we begin looking at the Java architecture, you'll see that much of what is exciting about Java comes from the self-contained, virtual machine environment in which Java applications run. Java has been carefully designed so that this supporting architecture can be implemented either in software, for existing computer platforms, or in customized hardware, for new kinds of devices. Sun and other industry giants have announced their intentions to produce cheap, fast Java chips, the first of which should be available by the time you read this. Hardware implementations of Java could power inexpensive network terminals, PDAs, and other information appliances, to take advantage of transportable Java applications.

Many people see Java as part of a trend toward cheap, Net-based, "operating system-less" appliances that will extend the Net into more and more consumer-related areas. Only time will tell what people will do with Java, but it's probably worth at least a passing thought that the applet you write today might well be running on someone's wristwatch tomorrow. If that seems too futuristic, remember that you can already get a "smart card" (essentially a credit card) that has a Java interpreter embedded in it. Such a card could do everything from financial transactions (paying a hotel bill) to unlocking a door (the door to your hotel room) to rerouting phone calls (so your hotel room receives your business calls). The card is already here; it won't be long before the rest of the software has been built. A Java wristwatch is certainly not far away.

PREVIOUS | HOME | NEXT
Acknowledgments | BOOK INDEX | A Virtual Machine

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

2. A First Applet

Contents:

Before we turn our attention to the details of the language, let's take a crash course and jump right into some Java code. In this chapter, we'll build a contrived but friendly little applet that illustrates a number of techniques we use throughout the book. I'll take this opportunity to introduce general features of the Java language and of Java applets. However, many details won't be fleshed out here, but in subsequent chapters.

This chapter also serves as a brief introduction to the object-oriented and multithreaded features of Java. If these concepts are new to you, you can take comfort in the knowledge that encountering them for the first time in Java should be a straightforward and pleasant experience. If you have worked with another object-oriented or multithreaded programming environment, clear your mind; you will especially appreciate Java's simplicity and elegance.

I can't stress enough the importance of experimentation as you learn new concepts. If you follow along with the online examples, be sure to take some time and compile them locally. Play with them; change their behavior, break them, fix them, and, as Java developer Arthur van Hoff would say: "Have fun!"

2.1 Hello Web!

In the tradition of all good introductory programming texts, we begin with Java's equivalent of the archetypal "Hello World" application. In the spirit of our new world, we'll call it "Hello Web!"

I'll take four passes at this example, adding features and introducing new concepts along the way. Here's a minimalist version:

```
public class HelloWeb extends java.applet.Applet {
    public void paint( java.awt.Graphics gc ) {
        gc.drawString("Hello Web!", 125, 95 );
    }
}
```

Place this text in a file called *HelloWeb.java*. Now compile this source:

```
% javac HelloWeb.java
```

This produces the Java byte-code binary class file *HelloWeb.class*.

We need an HTML document that contains the appropriate `<applet>` tag to display our example. Place the following text in a file called *HelloWeb.html* in the same directory as the binary class file:

```
<html>
<head>
</head>
<body>
    <applet code=HelloWeb width=300 height=200></applet>
</body>
</html>
```

Finally, you can point your Java-enabled Web browser at this document with a URL such as:

```
http://yourServer/wherever/HelloWeb.html
```

or

```
file:/wherever/HelloWeb.html
```

You should see the proclamation shown in Figure 2.1. Now congratulate yourself: you have written your first applet! Take a moment to bask in the glow of your monitor.

Figure 2.1: Hello Web! applet

[Graphic: Figure 2-1]

`HelloWeb` may be a small program, but there is actually quite a bit going on behind the scenes. Those five lines represent the tip of an iceberg. What lies under the surface are layers of functionality provided by the Java language and its foundation class libraries. In this chapter, I'll cover a lot of ground quickly in an effort to show you the big picture. I'll try to offer enough detail for a complete understanding of what is happening in each example without exhaustive explanations until the appropriate chapters. This holds for both elements of the Java language and the object-oriented concepts that apply to them. Later chapters will provide more detailed cataloging of Java's syntax, components, and object-oriented features.

Classes

The previous example defines a *class* named `HelloWeb`. Classes are the fundamental building blocks of most object-oriented languages. A class in Java is akin to the C++ concept of a class. Specifically, it's a group of data items (à la a C struct), with associated functions that perform operations on this data. The data items in a class are called *fields* or *variables* ; the functions are called *methods*. A class might represent something concrete, like a button on a screen or the information in a spreadsheet, or it could be something more abstract, such as a sorting algorithm or possibly the sense of ennui in your MUD character. A hypothetical spreadsheet class might, for example, have variables that represent the values of its individual cells and methods that perform operations on those cells, such as "clear a row" or "compute values."

Our `HelloWeb` class is the container for our Java applet. It holds two general types of variables and methods: those we need for our specific applet's tasks and some special predesignated ones we provide to interact with the outside world. The Java run-time environment, in this case a Java-enabled Web browser, periodically calls methods in `HelloWeb` to pass us information and prod us to perform actions, as depicted in Figure 2.2. Our simple `HelloWeb` class defines a single method called `paint()`. The `paint()` method is called by Java when it's time for our application to draw itself on the screen.

Figure 2.2: Method invocation in the Java environment

[Graphic: Figure 2-2]

You will see that the `HelloWeb` class derives some of its structure from another class called `Applet`. This is why we refer to `HelloWeb` as an applet.

Class Instances and Objects

A class represents a particular thing; it contains methods and variables that assist in that representation. Many individual working copies of a given class can exist while an application is active. These individual incarnations are called *instances* of the class. Two instances of a given class may contain different states, but they always have the same methods.

As an example, consider a `Button` class. There is only one `Button` class, but many actual working instances of buttons can be in an application. Furthermore, two `Button` instances might contain different data, perhaps giving each a different appearance or specifying a different message for each to send when pushed. In this sense, a class can be considered a mold for making the object it represents: something like a cookie cutter stamping out working instances of itself in the memory of the computer. As you'll see later, there's a bit more to it than that--a class can in fact share information among its instances--but this explanation suffices for now.

The term *object* is very general and in some other contexts is used almost interchangeably with class. Objects are the abstract entities all object-oriented languages refer to in one form or another. I will use object as a generic term for an instance of a class. I might, therefore, refer to an instance of the `Button` class as a `Button`, a `Button` object, or, indiscriminately, as an object.

A Java-enabled Web browser creates an instance of our `HelloWeb` class when we first use our applet. If we had included the `HelloWeb` applet tag in our HTML document twice (causing it to appear twice on the screen), the browser would create and manage two separate HelloWeb objects (two separate instances of the `HelloWeb` class).

Variables

In Java, every class defines a new *type*. A variable can be of this type and then hold instances of that class. A variable could, for example, be of type `Button` and hold an instance of the `Button` class, or of type `SpreadSheetCell` and hold a `SpreadSheetCell` object, just as it could be any of the more familiar types such as `integer` or `float`. In this way, by having variables containing complex objects, a class may use other classes as tools within itself. Using classes in this way is called *composition*. Our examples in this chapter are somewhat unrealistic in that we are building only a single class of our own. However, we will be using many classes as tools within our applet.

You have seen only one variable so far in our simple `HelloWeb` example. It's found in the declaration of our lonely `paint()` method:

```
public void paint( java.awt.Graphics gc ) {...}
```

Just like functions in C (and many other languages), a method in Java declares a list of variables that hold its arguments, and it specifies the types of those arguments. Our `paint()` method takes one argument named (somewhat tersely) `gc`, which is of type `Graphics`. When the `paint()` method is invoked, a `Graphics` object is assigned to `gc`, which we use in the body of the method. I'll say more about `paint()` and the `Graphics` class in a moment.

But first, a few words about variables. I have loosely referred to variables as holding objects. In reality, variables that have complex types (class types) don't so much contain objects as point to them. Class-type variables are *references* to objects. A reference is a pointer to, or another name for, an object.

Simply declaring a variable doesn't imply that any storage is allocated for that variable or that an instance of its type even exists anywhere. When a reference-type variable is first declared, if it's not assigned to an instance of a class, it doesn't point to anything. It's assigned the default value of `null`, meaning "no value." If you try to use a variable with a `null` value as if it were pointing to a real object, a run-time error (NullPointerException) occurs.

This discussion begs the question as to where to get an instance of a class to assign to a variable in the first place. The answer, as you will see later, is through the use of the `new` operator. In our first two passes at this example, we are dealing only with objects handed to us prefabricated from somewhere outside of our class. We examine object creation later in the chapter.

Inheritance

Java classes are arranged in a parent-child hierarchy, in which the parent and child are known as the *superclass* and *subclass*, respectively. In Java, every class has exactly one superclass (a single parent), but possibly many subclasses. Of course, a class's superclass probably has its own superclass.

The declaration of our class in the previous example uses the keyword `extends` to specify that `HelloWeb` is a subclass of the `Applet` class:

```
public class HelloWeb extends java.applet.Applet {...}
```

A subclass may be allowed to inherit some or all of the variables and methods of its superclass. Through *inheritance*, the subclass can use those members as if it has declared them itself. A subclass can add variables and methods of its own, and it can also override the meaning of inherited variables and methods. When we use a subclass, overridden variables and methods are hidden (replaced) by the subclass's own versions of them. In this way, inheritance provides a powerful mechanism whereby a subclass can refine or extend its superclass.

For example, the hypothetical spreadsheet class might be subclassed to produce a new scientific spreadsheet class with extra mathematical functions and special built-in constants. In this case, the source code for the scientific spreadsheet might declare methods for the added mathematical functions and variables for the special constants, but the new class automatically has all the variables and methods that constitute the normal functionality of a spreadsheet; they are inherited from the parent spreadsheet class. This means the scientific spreadsheet maintains its identity as a spreadsheet, and we can use it anywhere the simpler spreadsheet is used.

Our `HelloWeb` class is a subclass of the `Applet` class and inherits many variables and methods not explicitly declared in our source code. These members function in the same way as the ones we add or override.

Applet

The `Applet` class provides the framework for building applets. It contains methods that support the basic functionality for a Java application that is displayed and controlled by a Java-enabled Web browser or other Java-enabled software.

We override methods in the `Applet` class in a subclass to implement the behavior of our particular applet. This may sound restrictive, as if we are limited to some predefined set of routines, but that is not the case at all. Keep in mind that the methods we are talking about are means of getting information from the outside world. A realistic application might involve hundreds or even thousands of classes, with legions of methods and variables and multiple threads of execution. The vast majority of these are related to the particulars of our job. The inherited methods of the `Applet` class, and of other special components, serve as a framework on which to hang code that handles certain types of events and performs special tasks.

The `paint()` method is an important method of the `Applet` class; we override it to implement the way in which our particular applet displays itself on the screen. We don't override any of the other inherited members of `Applet` because they provide basic functionality and reasonable defaults for this (trivial) example. As `HelloWeb` grows, we'll delve deeper into the inherited members and override additional methods. Inherited members will allow us to get information from the user and give us more control over what our applet does. We will also add some arbitrary, application-specific methods and variables for the needs of `HelloWeb`.

If you want to verify for yourself what functionality the `Applet` class is providing our example, you can try out the world's least interesting applet: the `Applet` base class itself. Just use the class name `java.applet.Applet` in your HTML code, instead of `HelloWeb`:

```
<applet code=java.applet.Applet width=300 height=200></applet>
```

You should get a blank area of screen. I told you it's not very interesting.

Relationships and Finger Pointing

We can correctly refer to `HelloWeb` as an `Applet` because subclassing can be thought of as creating an "is a" relationship, in which the subclass is a kind of its superclass. `HelloWeb` is therefore a kind of `Applet`. When we refer to a kind of object, we mean any instance of that object's class or any of its subclasses. Later, we will look more closely at the Java class hierarchy and see that `Applet` is itself a subclass of the `Panel` class, which is further derived from a class called `Container`, and so on, as shown in Figure 2.3.

Figure 2.3: Part of the Java class hierarchy

[Graphic: Figure 2-3]

In this sense, an `Applet` is a kind of `Panel`, which is, itself, a kind of `Container` and each of these can ultimately be considered to be a kind of `Component`. You'll see later that it's from these classes that `Applet` inherits its basic graphical user interface functionality and the ability to have other graphical components embedded within it.

`Component` is a subclass of `Object`, so all of these classes are a kind of `Object`. As you'll see later, the `Object` class is at the top of the Java class hierarchy; `Object` doesn't have a superclass. Every other class in the Java API inherits behavior from `Object`, which defines a few basic methods, as you'll see in Chapter 5, *Objects in Java*. The terminology here can become a bit muddled. I'll continue to use the word "object" (lowercase o) in a generic way to refer to an instance of any class; I'll use `Object` to refer specifically to that class.

Packages

In our previous example, the `Applet` class is referenced by its fully qualified name `java.applet.Applet`:

```
public class HelloWeb extends java.applet.Applet {...}
```

The prefix on the class name identifies it as belonging to the `java.applet` package. Packages provide a means for organizing Java classes. A *package* is a group of Java classes that are related by purpose or by application. Classes in the same package have special access privileges with respect to one another and may be designed to work together. Package names are hierarchical and are used somewhat like Internet domain and host names, to distinguish groups of classes by organization and application. Classes may be dynamically loaded over networks from arbitrary locations; within this context, packages provide a crude namespace of Java classes.[1]

> [1] There are many efforts under way to find a general solution to the problem of locating resources in a globally distributed computing environment. The Uniform Resource Identifier Working Group of the IETF has proposed Uniform Resource Names (URNs). A URN would be a more abstract and persistent identifier that would be resolved to a URL through the use of a name service. We can imagine a day when there will exist a global namespace of trillions of persistent objects forming the infrastructure for all computing resources. Java provides an important evolutionary step in this direction.

`java.applet` identifies a particular package that contains classes related to applets. `java.applet.Applet` identifies a specific class, the `Applet` class, within that package. The `java.` hierarchy is special. Any package that begins with `java.` is part of the core Java API and is available on any platform that supports Java. Figure 2.4 illustrates the core Java packages, showing a representative class or two from each package.

Figure 2.4: The core Java packages

[Graphic: Figure 2-4]

Some notable core packages include: `java.lang`, which contains fundamental classes needed by the Java language itself; `java.awt`, which contains classes of the Java Abstract Windowing Toolkit; and `java.net`, which contains the networking classes.

A few classes contain methods that are not written in Java, but are instead part of the native Java implementation on a particular platform. Approximately 22 such classes are in the `java` package hierarchy; these are the only classes that have to be ported to a new platform. They form the basis for all interaction with the operating system. All other classes are built on or around these and are completely platform independent.

The paint( ) Method

The source for our `HelloWeb` class defines just one method, `paint()`, which overrides the `paint()` method from the `Applet` class:

```
public void paint( java.awt.Graphics gc ) {
    gc.drawString("Hello Web!", 125, 95 );
}
```

The `paint()` method is called by Java when it's time for our applet to draw itself on the screen. It takes a single argument, a `Graphics` object, and doesn't return any type of value (`void`) to its caller.

Modifiers are keywords placed before classes, variables, and methods to alter their accessibility, behavior, or semantics. `paint()` is declared as `public`, which means it can be invoked (called) by methods in classes other than `HelloWeb`. In this case, it's the Java windowing environment that is calling our `paint()` method. A method or variable declared as `private` is inaccessible from outside of its class.

The `Graphics` object, an instance of the `Graphics` class, represents a particular graphical drawing area and is also called a graphics context. It contains methods the applet calls to draw in this area, and variables that represent characteristics such as clipping or drawing modes. The particular `Graphics` object we are passed in the `paint()` method corresponds to our applet's area of the screen.

The `Graphics` class provides methods for rendering primitive shapes, images, and text. In `HelloWeb`, we invoke the `drawString()` method of our `Graphics` object to scrawl our message at the specified coordinates. (For a description of the methods available in the `Graphics` class, see Chapter 13, *Drawing With the AWT*.)

As in C++, a method or variable of an object is accessed in a hierarchical way by appending its name with a "." (dot) to the object that holds it. We invoked the `drawString()` method of the `Graphics` object (referenced by our `gc` variable) in this way:

```
gc.drawString( "Hello Web!", 125, 95 );
```

You may need to get used to the idea that our application is drawn by a method that is called by an outside agent at arbitrary times. How can we do anything useful with this? How do we control what gets done and when? These answers will be forthcoming. For now, just think about how you would structure applications that draw themselves on command.

PREVIOUS Availability | HOME BOOK INDEX | NEXT Hello Web! II: The Sequel

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

3. Tools of the Trade

Contents:

As I described at the end of Chapter 1, *Yet Another Language?*, by now you should have a number of options for Java development environments. The examples in this book were developed using the Solaris version of the Java Development Kit (JDK), so I'm going to describe those tools here. When I refer to the compiler or interpreter, I'll be referring to the command-line versions of these tools, so the book is decidedly biased toward those of you who are working in a UNIX or DOS-like environment with a shell and filesystem. However, the basic features I'll be describing for Sun's Java interpreter and compiler should be applicable to other Java environments as well.

In this chapter, I'll describe the tools you'll need to compile and run Java applications. I'll also cover the HTML `<applet>` tag and other information you'll need to know to incorporate Java applets in your Web pages.

3.1 The Java Interpreter

A Java interpreter is software that implements the Java virtual machine and runs Java applications. It can be a separate piece of software like the one that comes with the JDK, or part of a larger application like the Netscape Navigator Web browser. It's likely that the interpreter itself is written in a native, compiled language for your particular platform. Other tools, like Java compilers and development environments, can (and one could argue, should) be written in Java.

The Java interpreter performs all of the activities of the Java run-time system. It loads Java class files and interprets the compiled byte-code. It verifies compiled classes that are loaded from untrusted sources by applying the rules discussed in Chapter 1, *Yet Another Language?*. In an implementation that supports dynamic, or "just in time," compilation, the interpreter also serves as a specialized compiler that turns Java byte-code into native machine instructions.

Throughout the rest of this book, we'll be building both standalone Java programs and applets. Both are kinds of Java applications run by a Java interpreter. The difference is that a standalone Java application has all of its parts; it's a complete program that runs independently. An applet, as I described in Chapter 1, *Yet Another Language?*, is more like an embeddable program module; it relies on an applet viewer for support. Although Java applets are, of course, compiled Java code, the Java interpreter can't directly run them because they are used as part of a larger application. An applet-viewer application could be a Web browser like Sun's HotJava or Netscape Navigator, or a separate applet viewer application like the one that comes with Sun's Java Development Kit. All of Sun's tools, including HotJava, are written entirely in Java. Both HotJava and the applet viewer are standalone Java applications run directly by the Java interpreter; these programs implement the additional structure needed to run Java applets.

Sun's Java interpreter is called *java*. To start a standalone application with it, you specify an initial class to be loaded. You can also specify options to the interpreter, as well as any command-line arguments that are needed for the application:

```
% java [interpreter options] class name [program arguments]
```

The class should be specified as a fully qualified class name including the class package, if any. Note, however, that you don't include the *.class* file extension. Here are a few examples:

```
% java animals.birds.BigBird
% java test
```

java searches for the class in the current *class path*, which is a list of locations where packages of classes are stored. I'll discuss the class path in detail in the next section, but for now you should know that you can set the class path with the `-classpath` option.

There are a few other interpreter options you may find useful. The `-cs` or `-checksource` option tells *java* to check the modification times on the specified class file and its corresponding source file. If the class file is out of date, it's automatically recompiled from the source. The `-verify`, `-noverify`, and `-verifyremote` options control the byte-code verification process. By default, *java* runs the byte-code verifier only on classes loaded from an untrusted source; this is the `-verifyremote` option. If you specify `-verify`, the byte-code verifier is run on all classes; `-noverify` means that the verifier is never run.

Once the class is loaded, *java* follows a very C-like convention and looks to see if the class contains a method called `main()`. If it finds an appropriate `main()`method, the interpreter starts the application by executing that method. From there, the application can start additional threads, reference other classes, and create its user interface or other structures, as shown in Figure 3.1.

Figure 3.1: The Java interpreter starting a Java application

[Graphic: Figure 3-1]

In order to run, `main()` must have the right *method signature*. A method signature is a collection of information about the method, as in a C prototype or a forward function declaration in other languages. It includes the method's name, type, and visibility, as well as its arguments and return type. In this case, `main()` must be a `public`, `static` method that takes an array of `String` objects as its argument and does not return any value (`void`):

```
public static void main ( String [] myArgs )
```

Because `main()` is a `public` method, it can be accessed directly from any other class using the name of the class that contains it. We'll discuss the implications of visibility modifiers such as `public` in Chapter 5, *Objects in Java*.

The `main()` method's single argument, the array of `String` objects, holds the command-line arguments passed to *java*. As in C, the name that we give the parameter doesn't matter, only the type is important. Unlike C, the content of `myArgs` is a true array. There's no need for an argument count parameter, because `myArgs` knows how many arguments it contains and can happily provide that information:

```
int argc = myArgs.length;
```

Java also differs from C in another respect here: `myArgs[0]`is the first command-line argument, not the name of the application. If you're accustomed to parsing C command-line arguments, you'll need to be careful not to trip over this difference.

The Java virtual machine continues to run until the `main()`method of its initial class file has returned, and until any threads that it started are complete. Special threads designated as "daemon" threads are silently killed when the rest of the application has completed.

PREVIOUS HOME NEXT
Hello Web! IV: Netscape's Revenge BOOK INDEX The Class Path

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

4. The Java Language

Contents:

In this chapter, we'll introduce the framework of the Java language and some of its fundamental tools. I'm not going to try to provide a full language reference here. Instead, I'll lay out the basic structures of Java with special attention to how it differs from other languages. For example, we'll take a close look at arrays in Java, because they are significantly different from those in some other languages. We won't, on the other hand, spend much time explaining basic language constructs like loops and control structures. We won't talk much about Java's object-oriented features here, as that's covered in Chapter 5, *Objects in Java*.

As always, we'll try to provide meaningful examples to illustrate how to use Java in everyday programming tasks.

4.1 Text Encoding

Java is a language for the Internet. Since the people of the Net speak and write in many different human languages, Java must be able to handle a number of languages as well. One of the ways in which Java supports international access is through Unicode character encoding. Unicode uses a 16-bit character encoding; it's a worldwide standard that supports the scripts (character sets) of most languages.[1]

[1] For more information about Unicode, see the following URL: http://www.unicode.org/. Ironically, one listed "obsolete and archaic" scripts not currently supported by the Unicode

standard is Javanese--a historical language of the people of the Island of Java.

Java source code can be written using the Unicode character encoding and stored either in its full form or with ASCII-encoded Unicode character values. This makes Java a friendly language for non-English speaking programmers, as these programmers can use their native alphabet for class, method, and variable names in Java code.

The Java `char` type and `String` objects also support Unicode. But if you're concerned about having to labor with two-byte characters, you can relax. The `String` API makes the character encoding transparent to you. Unicode is also ASCII-friendly; the first 256 characters are identical to the first 256 characters in the ISO8859-1 (Latin-1) encoding and if you stick with these values, there's really no distinction between the two.

Most platforms can't display all currently defined Unicode characters. As a result, Java programs can be written with special Unicode escape sequences. A Unicode character can be represented with the escape sequence:

```
\uxxxx
```

`xxxx` is a sequence of one to four hexadecimal digits. The escape sequence indicates an ASCII-encoded Unicode character. This is also the form Java uses to output a Unicode character in an environment that doesn't otherwise support them.

Java stores and manipulates characters and strings internally as Unicode values. Java also comes with classes to read and write Unicode-formatted character streams, as you'll see in Chapter 8, *Input/Output Facilities*.

PREVIOUS | HOME | NEXT
The Applet Tag | BOOK INDEX | Comments

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

5. Objects in Java

Contents:

In this chapter, we'll get to the heart of Java and explore the object-oriented aspects of the language. Object-oriented design is the art of decomposing an application into some number of objects--self-contained application components that work together. The goal is to break the problem down into a number of smaller problems that are simpler and easier to understand. Ideally, the components can be implemented directly as objects in the Java language. And if things are truly ideal, the components correspond to well-known objects that already exist, so they don't have to be created at all.

An object design methodology is a system or a set of rules created by someone to help you identify objects in your application domain and pick the real ones from the noise. In other words, such a methodology helps you factor your application into a good set of reusable objects. The problem is that though it wants to be a science, good object-oriented design is still pretty much an art form. While you can learn from the various off-the-shelf design methodologies, none of them will help you in all situations. The truth is that experience pays.

I won't try to push you into a particular methodology here; there are shelves full of books to do that.[1] Instead, I'll provide a few hints to get you started. Here are some general design guidelines, which should be taken with a liberal amount of salt and common sense:

> [1] Once you have some experience with basic object-oriented concepts, you might want to take a look at *Design Patterns: Elements of Reusable Object Oriented Software* by Gamma/Helm/Johnson/Vlissides (Addison-Wesley). This book catalogs useful object-oriented designs that have been refined over the years by experience. Many appear in the design of the Java APIs.

- Think of an object in terms of its interface, not its implementation. It's perfectly fine for an object's internals to be unfathomably complex, as long as its "public face" is easy to understand.

- Hide and abstract as much of your implementation as possible. Avoid public variables in your objects, with the possible exception of constants. Instead define "accessor" methods to set and return values (even if they are simple types). Later, when you need to, you'll be able to modify and extend the behavior of your objects without breaking other classes that rely on them.

- Specialize objects only when you have to. When you use an object in its existing form, as a piece of a new object, you are composing objects. When you change or refine the behavior of an object, you are using inheritance. You should try to reuse objects by composition rather than inheritance whenever possible because when you compose objects you are taking full advantage of existing tools. Inheritance involves breaking down the barrier of an object and should be done only when there's a real advantage. Ask yourself if you really need to inherit the whole public interface of an object (do you want to be a "kind" of that object), or if you can just delegate certain jobs to the object and use it by composition.

- Minimize relationships between objects and try to organize related objects in packages. To enhance your code's reusability, write it as if there *is* a tomorrow. Find what one object needs to know about another to get its job done and try to minimize the coupling between them.

5.1 Classes

Classes are the building blocks of a Java application. A *class* can contain methods, variables, initialization code, and, as we'll discuss later on, even other classes. It serves as a blueprint for making class *instances*, which are run-time objects that implement the class structure. You declare a class with the `class` keyword. Methods and variables of the class appear inside the braces of the class declaration:

```
class Pendulum {
    float mass;
    float length = 1.0;
    int cycles;

    float position ( float time ) {
        ...
    }
    ...
}
```

The above class, `Pendulum`, contains three variables: `mass`, `length`, and `cycles`. It also defines a method called `position()` that takes a `float` value as an argument and returns a `float` value. Variables and method declarations can appear in any order, but variable initializers can't use forward references to uninitialized variables.

Once we've defined the `Pendulum` class, we can create a `Pendulum` object (an instance of that class) as follows:

```
Pendulum p;
p = new Pendulum();
```

Recall that our declaration of the variable `p` does not create a `Pendulum` object; it simply creates a variable that refers to an object of type `Pendulum`. We still have to create the object dynamically, using the `new` keyword. Now that we've created a `Pendulum` object, we can access its variables and methods, as we've already seen many times:

```
p.mass = 5.0;
float pos = p.position( 1.0 );
```

Variables defined in a class are called *instance variables*. Every object has its own set of instance variables; the values of these variables in one object can differ from the values in another object, as shown in Figure 5.1. If you don't initialize an instance variable when you declare it, it's given a default value appropriate for its type.

Figure 5.1: Instances of the Pendulum class

[Graphic: Figure 5-1]

In Figure 5.1, we have a hypothetical `TextBook` application that uses two instances of `Pendulum` through the reference type variables `bigPendulum` and `smallPendulum`. Each of these `Pendulum` objects has its own copy of `mass`, `length`, and `cycles`.

As with variables, methods defined in a class are *instance methods*. An instance method is associated with an instance of the class, but each instance doesn't really have its own copy of the method. Instead, there's just one copy of the method, but it operates on the values of the instance variables of a particular object. As you'll see later when we talk about subclassing, there's more to learn about method selection.

Accessing Members

Inside of a class, we can access instance variables and call instance methods of the class directly by name. Here's an example that expands upon our `Pendulum`:

```
class Pendulum {
    ...
    void resetEverything() {
        cycles = 0;
        mass = 1.0;
        ...
        float startingPosition = position( 0.0 );
    }
    ...
}
```

Other classes generally access members of an object through a reference, using the C-style dot notation:

```
class TextBook {
    ...
    void showPendulum() {
        Pendulum bob = new Pendulum();
        ...
        int i = bob.cycles;
        bob.resetEverything();
        bob.mass = 1.01;
        ...
    }
    ...
}
```

Here we have created a second class, `TextBook`, that uses a `Pendulum` object. It creates an instance in `showPendulum()` and then invokes methods and accesses variables of the object through the reference `bob`.

Several factors affect whether class members can be accessed from outside the class. You can use the visibility modifiers, `public`, `private`, and `protected` to restrict access; classes can also be placed into packages that affect their scope. The `private` modifier, for example, designates a variable or method for use only by other members inside the class itself. In the previous example, we could change the declaration of our variable `cycles` to `private`:

```
class Pendulum {
    ...
    private int cycles;
    ...
```

Now we can't access `cycles` from `TextBook`:

```
class TextBook {
    ...
    void showPendulum() {
        ...
        int i = bob.cycles;         // Compile time error
```

If we need to access cycles, we might add a `getCycles()` method to the `Pendulum` class. We'll look at access modifiers and how they affect the scope of variables and methods in detail later.

Static Members

Instance variables and methods are associated with and accessed through a particular object. In contrast, members that are declared with the `static` modifier live in the class and are shared by all instances of the class. Variables declared with the `static` modifier are called *static variables* or *class variables* ; similarly, these kinds of methods are called *static methods* or *class methods*.

We can add a static variable to our `Pendulum` example:

```
class Pendulum {
    ...
    static float gravAccel = 9.80;
    ...
```

We have declared the new `float` variable `gravAccel` as `static`. That means if we change its value in any instance of a `Pendulum`, the value changes for all `Pendulum` objects, as shown in Figure 5.2.

Figure 5.2: A static variable

[Graphic: Figure 5-2]

Static members can be accessed like instance members. Inside our `Pendulum` class, we can refer to `gravAccel` by name, like an instance variable:

```
class Pendulum {
    ...
    float getWeight () {
        return mass * gravAccel;
    }
    ...
}
```

However, since static members exist in the class itself, independent of any instance, we can also access them directly through the class. We don't need a `Pendulum` object to set the variable `gravAccel`; instead we can use the class name as a reference:

```
Pendulum.gravAccel = 8.76;
```

This changes the value of `gravAccel` for any current or future instances. Why, you may be wondering, would we want to change the value of `gravAccel`? Well, perhaps we want to explore how pendulums would work on different planets. Static variables are also very useful for other kinds of data shared among classes at run-time. For instance you can create methods to register your objects so that they can communicate or you can count references to them.

We can use static variables to define constant values. In this case, we use the `static` modifier along with the `final` modifier. So, if we cared only about pendulums under the influence of the Earth's gravitational pull, we could change `Pendulum` as follows:

```
class Pendulum {
    ...
    static final float EARTH_G = 9.80;
    ...
```

We have followed a common convention and named our constant with capital letters; C programmers should recognize the capitalization convention, which resembles C `#define` statements. Now the value of `EARTH_G` is a constant; it can be accessed by any instance of `Pendulum` (or anywhere, for that matter), but its value can't be changed at run-time.

It's important to use the combination of static and final only for things that are really constant. That's because, unlike other kinds of variable references, the compiler is allowed to "inline" those values within classes that reference them. This is probably OK for things like PI, which aren't likely to change for a while, but may not be ideal for other kinds of identifiers, such as we'll discuss below.

Static members are useful as flags and identifiers, which can be accessed from anywhere. These are especially useful for values needed in the construction of an instance itself. In our example, we might declare a number of static values to represent various kinds of `Pendulum` objects:

```
class Pendulum {
    ...
    static int SIMPLE = 0, ONE_SPRING = 1, TWO_SPRING = 2;
    ...
```

We might then use these flags in a method that sets the type of a `Pendulum` or, more likely, in a special constructor, as we'll discuss shortly:

```
Pendulum pendy = new Pendulum();
pendy.setType( Pendulum.ONE_SPRING );
```

Remember, inside the `Pendulum` class, we can use static members directly by name as well:

```
class Pendulum {
    ...
    void resetEverything() {
        setType ( SIMPLE );
        ...
    }
    ...
}
```

PREVIOUS | HOME | NEXT
Arrays | BOOK INDEX | Methods

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

7. Basic Utility Classes

Contents:

If you've been reading this book sequentially, you've read all about the core Java language constructs, including the object-oriented aspects of the language and the use of threads. Now it's time to shift gears and talk about the Java Application Programming Interface (API), the collection of classes that comes with every Java implementation. The Java API encompasses all the public methods and variables in the classes that comprise the core Java packages, listed in Table 7.1. This table also lists the chapters in this book that describe each of the packages.

Table 7.1: Packages of the Java API

Package	Contents	Chapter(s)
`java.lang`	Basic language classes	4, 5, 6, 7
`java.io`	Input and output	8
`java.util`	Utilities and collections classes	7
`java.text`	International text classes	7
`java.net`	Sockets and URLs	9
`java.applet`	The applet API	10
`java.awt`	The Abstract Windowing Toolkit	10, 11, 12, 13, 14
`java.awt.image`	AWT image classes	13, 14
`java.beans`	Java Beans API	
`java.rmi`	RMI classes	
`java.security`	Encryption and signing	
`java.sql`	JDBC classes	

As you can see in Table 7.1, we've already examined some of the classes in `java.lang` in earlier chapters on the core language constructs. Starting with this chapter, we'll throw open the Java toolbox and begin examining the rest of the classes in the API.

We'll begin our exploration with some of the fundamental language classes in `java.lang`, including strings and math utilities. Figure 7.1 shows the class hierarchy of the `java.lang` package.

Figure 7.1: The java.lang package

[Graphic: Figure 7-1]

We cover some of the classes in `java.util`, such as classes that support date and time values, random numbers, vectors, and hashtables. Figure 7.2 shows the class hierarchy of the `java.util` package.

Figure 7.2: The java.util package

[Graphic: Figure 7-2]

7.1 Strings

In this section, we take a closer look at the Java `String` class (or more specifically, `java.lang.String`). Because strings are used so extensively throughout Java (or any programming language, for that matter), the Java `String` class has quite a bit of functionality. We'll test drive most of the important features, but before you go off and write a complex parser or regular expression library, you should probably refer to a Java class reference manual for additional details.

Strings are immutable; once you create a `String` object, you can't change its value. Operations that would otherwise change the characters or the length of a string instead return a new `String` object that copies the needed parts of the original. Because of this feature, strings can be safely shared. Java makes an effort to consolidate identical strings and string literals in the same class into a shared string pool.

String Constructors

To create a string, assign a double-quoted constant to a `String` variable:

```
String quote = "To be or not to be";
```

Java automatically converts the string literal into a `String` object. If you're a C or C++ programmer, you may be wondering if `quote` is null-terminated. This question doesn't make any sense with Java strings. The `String` class actually uses a Java character array internally. It's `private` to the `String` class, so you can't get at the characters and change them. As always, arrays in Java are real objects that know their own length, so `String` objects in Java don't require special terminators (not even internally). If you need to know the length of a `String`, use the `length()` method:

```
int length = quote.length();
```

Strings can take advantage of the only overloaded operator in Java, the `+` operator, for string concatenation. The following code produces equivalent strings:

```
String name = "John " + "Smith";
String name = "John ".concat("Smith");
```

Literal strings can't span lines in Java source files, but we can concatenate lines to produce the same effect:

```
String poem =
    "'Twas brillig, and the slithy toves\n" +
    "   Did gyre and gimble in the wabe:\n" +
    "All mimsy were the borogoves,\n" +
    "   And the mome raths outgrabe.\n";
```

Of course, embedding lengthy text in source code should now be a thing of the past, given that we can retrieve a `String` from anywhere on the planet via a URL. In Chapter 9, *Network Programming*, we'll see how to do things like:

```
String poem =
    (String) new URL
       ("http://server/~dodgson/jabberwocky.txt").getContent();
```

In addition to making strings from literal expressions, we can construct a `String` from an array of characters:

```
char [] data = { 'L', 'e', 'm', 'm', 'i', 'n', 'g' };
String lemming = new String( data );
```

Or from an array of bytes:

```
byte [] data = { 97, 98, 99 };
String abc = new String(data, "8859_5");
```

The second argument to the `String` constructor for byte arrays is the name of an encoding scheme. It's used to convert the given bytes to the string's Unicode characters. Unless you know something about Unicode, you can probably use the form of the constructor that accepts only a byte array; the default encoding scheme will be used.

Strings from Things

We can get the string representation of most things with the static `String.valueOf()` method. Various overloaded versions of this method give us string values for all of the primitive types:

```
String one = String.valueOf( 1 );
String two = String.valueOf( 2.0f );
String notTrue = String.valueOf( false );
```

All objects in Java have a `toString()` method, inherited from the `Object` class (see Chapter 5, *Objects in Java*).

For class-type references, `String.valueOf()` invokes the object's `toString()` method to get its string representation. If the reference is `null`, the result is the literal string "null":

```
String date = String.valueOf( new Date() );
System.out.println( date );
// Sun Dec 19 05:45:34 CST 1999

date = null;
System.out.println( date );
// null
```

Things from Strings

Producing primitives like numbers from `String` objects is not a function of the `String` class. For that we need the primitive wrapper classes; they are described in the next section on the `Math` class. The wrapper classes provide `valueOf()` methods that produce an object from a `String`, as well as corresponding methods to retrieve the value in various primitive forms. Two examples are:

```
int i = Integer.valueOf("123").intValue();
double d = Double.valueOf("123.0").doubleValue();
```

In the above code, the `Integer.valueOf()` call yields an `Integer` object that represents the value 123. An `Integer` object can provide its primitive value in the form of an `int` with the `intValue()` method.

Although the techniques above may work for simple cases, they will not work internationally. Let's pretend for a moment that we are programming Java in the rolling hills of Tuscany. We would follow the local customs for representing numbers and write code like the following.

```
double d = Double.valueOf("1.234,56").doubleValue();    // oops!
```

Unfortunately, this code throws a `NumberFormatException`. The `java.text` package, which we'll discuss later, contains the tools we need to generate and parse strings in different countries and languages.

The `charAt()` method of the `String` class lets us get at the characters of a `String` in an array-like fashion:

```
String s = "Newton";

for ( int i = 0; i < s.length(); i++ )     System.out.println( s.charAt( i ) );
```

This code prints the characters of the string one at a time. Alternately, we can get the characters all at once with `toCharArray()`. Here's a way to save typing a bunch of single quotes:

```
char [] abcs = "abcdefghijklmnopqrstuvwxyz".toCharArray();
```

Comparisons

Just as in C, you can't compare strings for equality with "`==`" because as in C, strings are actually references. If your

Java compiler doesn't happen to coalesce multiple instances of the same string literal to a single string pool item, even the expression `"foo" == "foo"` will return `false`. Comparisons with `<`, `>`, `<=`, and `>=` don't work at all, because Java can't convert references to integers.

Use the `equals()` method to compare strings:

```
String one = "Foo";

char [] c = { 'F', 'o', 'o' };
String two = new String ( c );

if ( one.equals( two ) )                // yes
```

An alternate version, `equalsIgnoreCase()`, can be used to check the equivalence of strings in a case-insensitive way:

```
String one = "FOO";
String two = "foo";

if ( one.equalsIgnoreCase( two ) )      // yes
```

The `compareTo()` method compares the lexical value of the `String` against another `String`. It returns an integer that is less than, equal to, or greater than zero, just like the C routine `string()`:

```
String abc = "abc";
String def = "def";
String num = "123";

if ( abc.compareTo( def ) < 0 )         // yes
if ( abc.compareTo( abc ) == 0 )        // yes
if ( abc.compareTo( num ) > 0 )         // yes
```

On some systems, the behavior of lexical comparison is complex, and obscure alternative character sets exist. Java avoids this problem by comparing characters strictly by their position in the Unicode specification.

In Java 1.1, the `java.text` package provides a sophisticated set of classes for comparing strings, even in different languages. German, for example, has vowels with umlauts (those funny dots) over them and a weird-looking beta character that represents a double-s. How should we sort these? Although the rules for sorting these characters are precisely defined, you can't assume that the lexical comparison we used above works correctly for languages other than English. Fortunately, the `Collator` class takes care of these complex sorting problems. In the following example, we use a `Collator` designed to compare German strings. (We'll talk about `Locale`s in a later section.) You can obtain a default `Collator` by calling the `Collator.getInstance()` method that has no arguments. Once you have an appropriate `Collator` instance, you can use its `compare()` method, which returns values just like `String`'s `compareTo()` method. The code below creates two strings for the German translations of "fun" and "later," using Unicode constants for these two special characters. It then compares them, using a `Collator` for the German locale; the result is that "later" (spaeter) sorts before "fun" (spass).

```
String fun = "Spa\u00df";
String later = "sp\u00e4ter";
Collator german = Collator.getInstance(Locale.GERMAN);
if (german.compare(later, fun) < 0)  // yes
```

Using collators is essential if you're working with languages other than English. In Spanish, for example, "ll" and "ch" are treated as separate characters, and alphabetized separately. A collator handles cases like these automatically.

Searching

The `String` class provides several methods for finding substrings within a string. The `startsWith()` and `endsWith()` methods compare an argument `String` with the beginning and end of the `String`, respectively:

```
String url = "http://foo.bar.com/";
if ( url.startsWith("http:") )
    // do HTTP
```

Overloaded versions of `indexOf()` search for the first occurrence of a character or substring:

```
int i = abcs.indexOf( 'p' );            // i = 15
int i = abcs.indexOf( "def" );          // i = 3
```

Correspondingly, overloaded versions of `lastIndexOf()` search for the last occurrence of a character or substring.

Editing

A number of methods operate on the `String` and return a new `String` as a result. While this is useful, you should be aware that creating lots of strings in this manner can affect performance. If you need to modify a string often, you should use the `StringBuffer` class, as I'll discuss shortly.

`trim()` is a useful method that removes leading and trailing white space (i.e., carriage return, newline, and tab) from the `String`:

```
String abc = "   abc   ";
abc = abc.trim();                       // "abc"
```

In the above example, we have thrown away the original `String` (with excess white space), so it will be garbage collected.

The `toUpperCase()` and `toLowerCase()` methods return a new `String` of the appropriate case:

```
String foo = "FOO".toLowerCase();
String FOO = foo.toUpperCase();
```

`substring()` returns a specified range of characters. The starting index is inclusive; the ending is exclusive:

```
String abcs = "abcdefghijklmnopqrstuvwxyz";
String cde = abcs.substring(2, 5);      // "cde"
```

String Method Summary

Many people complain when they discover the Java `String` class is `final` (i.e., it can't be subclassed). There is a lot of functionality in `String`, and it would be nice to be able to modify its behavior directly. Unfortunately, there is also a serious need to optimize and rely on the performance of `String` objects. As I discussed in Chapter 5, *Objects in Java*, the Java compiler can optimize `final` classes by inlining methods when appropriate. The implementation of `final` classes can also be trusted by classes that work closely together, allowing for special cooperative optimizations. If you want to make a new string class that uses basic `String` functionality, use a `String` object in your class and provide methods that delegate method calls to the appropriate `String` methods.

Table 7.2 summarizes the methods provided by the `String` class.

Table 7.2: String Methods

Method	**Functionality**
`charAt()`	Gets at a particular character in the string
`compareTo()`	Compares the string with another string
`concat()`	Concatenates the string with another string
`copyValueOf()`	Returns a string equivalent to the specified character array
`endsWith()`	Checks if the string ends with a suffix
`equals()`	Compares the string with another string
`equalsIgnoreCase()`	Compares the string with another string and ignores case
`getBytes()`	Copies characters from the string into a byte array
`getChars()`	Copies characters from the string into a character array
`hashCode()`	Returns a hashcode for the string
`indexOf()`	Searches for the first occurrence of a character or substring in the string
`intern()`	Fetches a unique instance of the string from a global shared string pool
`lastIndexOf()`	Searches for the last occurrence of a character or substring in a string
`length()`	Returns the length of the string
`regionMatches()`	Checks whether a region of the string matches the specified region of another string
`replace()`	Replaces all occurrences of a character in the string with another character
`startsWith()`	Checks if the string starts with a prefix
`substring()`	Returns a substring from the string
`toCharArray()`	Returns the array of characters from the string
`toLowerCase()`	Converts the string to uppercase
`toString()`	Converts the string to a string
`toUpperCase()`	Converts the string to lowercase

`trim()`	Removes the leading and trailing white space from the string
`valueOf()`	Returns a string representation of a value

java.lang.StringBuffer

The `java.lang.StringBuffer` class is a growable buffer for characters. It's an efficient alternative to code like the following:

```
String ball = "Hello";
ball = ball + " there.";
ball = ball + " How are you?";
```

The above example repeatedly produces new `String` objects. This means that the character array must be copied over and over, which can adversely affect performance. A more economical alternative is to use a `StringBuffer` object and its `append()` method:

```
StringBuffer ball = new StringBuffer("Hello");
ball.append(" there.");
ball.append(" How are you?");
```

The `StringBuffer` class actually provides a number of overloaded `append()` methods, for appending various types of data to the buffer.

We can get a `String` from the `StringBuffer` with its `toString()` method:

```
String message = ball.toString();
```

`StringBuffer` also provides a number of overloaded `insert()` methods for inserting various types of data at a particular location in the string buffer.

The `String` and `StringBuffer` classes cooperate, so that even in this last operation, no copy has to be made. The string data is shared between the objects, unless and until we try to change it in the `StringBuffer`.

So, when should you use a `StringBuffer` instead of a `String`? If you need to keep adding characters to a string, use a `StringBuffer`; it's designed to efficiently handle such modifications. You'll still have to convert the `StringBuffer` to a `String` when you need to use any of the methods in the `String` class. You can print a `StringBuffer` directly using `System.out.println()`because `println()` calls the `toString()` for you.

Another thing you should know about `StringBuffer` methods is that they are thread-safe, just like all public methods in the Java API. This means that any time you modify a `StringBuffer`, you don't have to worry about another thread coming along and messing up the string while you are modifying it. If you recall our discussion of synchronization in Chapter 6, *Threads*, you know that being thread-safe means that only one thread at a time can change the state of a `StringBuffer` instance.

On a final note, I mentioned earlier that strings take advantage of the single overloaded operator in Java, `+`, for concatenation. You might be interested to know that the compiler uses a `StringBuffer` to implement concatenation. Consider the following expression:

```
String foo = "To " + "be " + "or";
```

This is equivalent to:

```
String foo = new
  StringBuffer().append("To ").append("be ").append("or").toString();
```

This kind of chaining of expressions is one of the things operator overloading hides in other languages.

java.util.StringTokenizer

A common programming task involves parsing a string of text into words or "tokens" that are separated by some set of delimiter characters. The `java.util.StringTokenizer` class is a utility that does just this. The following example reads words from the string `text`:

```
String text = "Now is the time for all good men (and women)...";
StringTokenizer st = new StringTokenizer( text );

while ( st.hasMoreTokens() )  {
    String word = st.nextToken();
    ...
}
```

First, we create a new `StringTokenizer` from the `String`. We invoke the `hasMoreTokens()` and `nextToken()` methods to loop over the words of the text. By default, we use white space (i.e., carriage return, newline, and tab) as delimiters.

The `StringTokenizer` implements the `java.util.Enumeration` interface, which means that `StringTokenizer` also implements two more general methods for accessing elements: `hasMoreElements()` and `nextElement()`. These methods are defined by the `Enumeration` interface; they provide a standard way of returning a sequence of values, as we'll discuss a bit later. The advantage of `nextToken()` is that it returns a `String`, while `nextElement()` returns an `Object`. The `Enumeration` interface is implemented by many items that return sequences or collections of objects, as you'll see when we talk about hashtables and vectors later in the chapter. Those of you who have used the C `strtok()`function should appreciate how useful this object-oriented equivalent is.

You can also specify your own set of delimiter characters in the `StringTokenizer` constructor, using another `String` argument to the constructor. Any combination of the specified characters is treated as the equivalent of white space for tokenizing:

```
text = "http://foo.bar.com/";
tok = new StringTokenizer( text, "/:" );

if ( tok.countTokens() < 2 )            // bad URL
```

```
String protocol = tok.nextToken();      // protocol = "http" String host =
tok.nextToken();          // host = "foo.bar.com"
```

The example above parses a URL specification to get at the protocol and host components. The characters "/" and ":" are used as separators. The `countTokens()` method provides a fast way to see how many tokens will be returned by `nextToken()`, without actually creating the `String` objects.

An overloaded form of `nextToken()` accepts a string that defines a new delimiter set for that and subsequent reads. And finally, the `StringTokenizer` constructor accepts a flag that specifies that separator characters are to be returned individually as tokens themselves. By default, the token separators are not returned.

PREVIOUS	HOME	NEXT
Scheduling and Priority	BOOK INDEX	Math Utilities

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

8. Input/Output Facilities

Contents:

In this chapter, we'll continue our exploration of the Java API by looking at many of the classes in the `java.io` package. These classes support a number of forms of input and output; I expect you'll use them often in your Java applications. Figure 8.1 shows the class hierarchy of the `java.io` package.

We'll start by looking at the stream classes in `java.io`; these classes are all subclasses of the basic `InputStream`, `OutputStream`, `Reader`, and `Writer` classes. Then we'll examine the `File` class and discuss how you can interact with the filesystem using classes in `java.io`. Finally, we'll take a quick look at the data compression classes provided in `java.util.zip`.

8.1 Streams

All fundamental I/O in Java is based on *streams*. A stream represents a flow of data, or a channel of communication with (at least conceptually) a writer at one end and a reader at the other. When you are working with terminal input and output, reading or writing files, or communicating through sockets in Java, you are using a stream of one type or another. So you can see the forest without being distracted by the trees, I'll start by summarizing the different types of streams.

Figure 8.1: The java.io package

[Graphic: Figure 8-1]

`InputStream`/`OutputStream`

Abstract classes that define the basic functionality for reading or writing an unstructured sequence of bytes. All other byte streams in Java are built on top of the basic `InputStream` and `OutputStream`.

`Reader`/`Writer`

Abstract classes that define the basic functionality for reading or writing an unstructured sequence of characters. All other character streams in Java are built on top of `Reader` and `Writer`.

`InputStreamReader`/`OutputStreamWriter`

"Bridge" classes that convert bytes to characters and vice versa.

`DataInputStream`/`DataOutputStream`

Specialized stream filters that add the ability to read and write simple data types like numeric primitives and `String` objects.

`BufferedInputStream`/`BufferedOutputStream` /`BufferedReader`/`BufferedWriter`

Specialized streams that incorporate buffering for additional efficiency.

`PrintWriter`

A specialized character stream that makes it simple to print text.

`PipedInputStream`/`PipedOutputStream` /`PipedReader`/`PipedWriter`

"Double-ended" streams that always occur in pairs. Data written into a `PipedOutputStream` or `PipedWriter` is read from its corresponding `PipedInputStream` or `PipedReader`.

`FileInputStream`/`FileOutputStream` /`FileReader`/`FileWriter`

Implementations of `InputStream`, `OutputStream`, `Reader`, and `Writer` that read from and write to files on the local filesystem.

Streams in Java are one-way streets. The `java.io` input and output classes represent the ends of a simple stream, as shown in Figure 8.2. For bidirectional conversations, we use one of each type of stream.

Figure 8.2: Basic input and output stream functionality

[Graphic: Figure 8-2]

`InputStream` and `OutputStream` are `abstract` classes that define the lowest-level interface for all byte streams. They contain methods for reading or writing an unstructured flow of byte-level data. Because these classes are abstract, you can never create a "pure" input or output stream. Java implements subclasses of these for activities like reading and writing files, and communicating with sockets. Because all byte streams inherit the structure of `InputStream` or `OutputStream`, the various kinds of byte streams can be used interchangeably. For example, a method often takes an `InputStream` as an argument. This means the method accepts any subclass of `InputStream`. Specialized types of streams can also be layered to provide higher-level functionality, such as buffering or handling larger data types.

In Java 1.1, new classes based around `Reader` and `Writer` were added to the `java.io` package. `Reader` and `Writer` are very much like `InputStream` and `OutputStream`, except that they deal with characters instead of bytes. As true character streams, these classes correctly handle Unicode characters, which was not always the case with the byte streams. However, some sort of bridge is needed between these character streams and the byte streams of physical devices like disks and networks. `InputStreamReader` and `OutputStreamWriter` are special classes that use an *encoding scheme* to translate between character and byte streams.

We'll discuss all of the interesting stream types in this section, with the exception of `FileInputStream`, `FileOutputStream`, `FileReader`, and `FileWriter`. We'll postpone the discussion of file streams until the next section, where we'll cover issues involved with accessing the filesystem in Java.

Terminal I/O

The prototypical example of an `InputStream` object is the standard input of a Java application. Like `stdin` in C or `cin` in C++, this object reads data from the program's environment, which is usually a terminal window or a command pipe. The `java.lang.System` class, a general repository for system-related resources, provides a reference to standard input in the `static` variable `in`. `System` also provides objects for standard output and standard error in the `out` and `err` variables, respectively. The following example shows the correspondence:

```
InputStream stdin = System.in;
OutputStream stdout = System.out;
OutputStream stderr = System.err;
```

This example hides the fact that `System.out` and `System.err` aren't really `OutputStream` objects, but more specialized and useful `PrintStream` objects. I'll explain these later, but for now we can reference `out` and `err` as `OutputStream` objects, since they are a kind of `OutputStream` by inheritance.

We can read a single byte at a time from standard input with the `InputStream`'s `read()` method. If you look closely at the API, you'll see that the `read()` method of the base `InputStream` class is actually an `abstract` method. What lies behind `System.in` is an implementation of `InputStream`, so it's valid to call `read()` for this stream:

```
try {
    int val = System.in.read();
    ...
}
catch ( IOException e ) {
}
```

As is the convention in C, `read()` provides a byte of information, but its return type is `int`. A return value of `-1` indicates a normal end of stream has been reached; you'll need to test for this condition when using the simple `read()` method. If an error occurs during the read, an `IOException` is thrown. All basic input and output stream commands can throw an `IOException`, so you should arrange to catch and handle them as appropriate.

To retrieve the value as a byte, perform the cast:

```
byte b = (byte) val;
```

Of course, you'll need to check for the end-of-stream condition before you perform the cast. An overloaded form of `read()` fills a byte array with as much data as possible up to the limit of the array size and returns the number of bytes read:

```
byte [] bity = new byte [1024];
int got = System.in.read( bity );
```

We can also check the number of bytes available for reading on an `InputStream` with the `available()` method. Once we have that information, we can create an array of exactly the right size:

```
int waiting = System.in.available();
if ( waiting > 0 ) {
    byte [] data = new byte [ waiting ];
    System.in.read( data );
    ...
}
```

InputStream provides the skip() method as a way of jumping over a number of bytes. Depending on the implementation of the stream and if you aren't interested in the intermediate data, skipping bytes may be more efficient than reading them. The close() method shuts down the stream and frees up any associated system resources. It's a good idea to close a stream when you are done using it.

Character Streams

The InputStream and OutputStream subclasses of Java 1.0.2 included methods for reading and writing strings, but most of them operated by assuming that a sixteen-bit Unicode character was equivalent to an eight-bit byte in the stream. This only works for Latin-1 (ISO8859-1) characters, so the character stream classes Reader and Writer were introduced in Java 1.1. Two special classes, InputStreamReader and OutputStreamWriter, bridge the gap between the world of character streams and the world of byte streams. These are character streams that are wrapped around an underlying byte stream. An encoding scheme is used to convert between bytes and characters. An encoding scheme name can be specified in the constructor of InputStreamReader or OutputStreamWriter. Another constructor simply accepts the underlying stream and uses the system's default encoding scheme. For example, let's parse a human-readable string from the standard input into an integer. We'll assume that the bytes coming from System.in use the system's default encoding scheme.

```
try {
    InputStreamReader converter = new InputStreamReader(System.in);
    BufferedReader in = new BufferedReader(converter);

    String text = in.readLine();
    int i = NumberFormat.getInstance().parse(text).intValue();
}
catch ( IOException e ) { }
catch ( ParseException pe ) { }
```

First, we wrap an InputStreamReader around System.in. This object converts the incoming bytes of System.in to characters using the default encoding scheme. Then, we wrap a BufferedReader around the InputStreamReader. BufferedReader gives us the readLine() method, which we can use to retrieve a full line of text into a String. The string is then parsed into an integer using the techniques described in Chapter 7.

We could have programmed the previous example using only byte streams, and it would have worked for users in the United States, at least. So why go to the extra trouble of using character streams? Character streams were introduced in Java 1.1 to correctly support Unicode strings. Unicode was designed to support almost all of the written languages of the world. If you want to write a program that works in any part of the world, in any language, you definitely want to use streams that don't mangle Unicode.

So how do you decide when you need a byte stream and when you need a character stream? If you want to read or write character strings, use some variety of Reader or Writer. Otherwise a byte stream should suffice. Let's say, for example, that you want to read strings from a file that was written out by a Java 1.0.2 application. In this case you could simply create a FileReader, which will convert the bytes in the file to characters using the system's default encoding scheme. If you have a file in a specific encoding scheme, you can create an InputStreamReader with that encoding scheme and read characters from it. Another example comes from the Internet. Web servers serve files as byte streams. If you want to read Unicode strings from a file with a particular encoding scheme, you'll need an appropriate InputStreamReader wrapped around the socket's InputStream.

Stream Wrappers

What if we want to do more than read and write a mess of bytes or characters? Many of the InputStream, OutputStream, Reader, and Writer classes wrap other streams and add new features. A filtered stream takes another stream in its constructor; it delegates calls to the underlying stream while doing some additional processing of its own.

In Java 1.0.2, all wrapper streams were subclasses of FilterInputStream and FilterOutputStream. The character stream classes introduced in Java 1.1 break this pattern, but they operate in the same way. For example, BufferedInputStream extends FilterInputStream in the byte world, but BufferedReader extends Reader in the character world. It doesn't really matter--both classes accept a stream in their constructor and perform buffering. Like the byte stream classes, the character stream classes include the abstract FilterReader and FilterWriter classes, which simply pass all method calls to an underlying stream.

The FilterInputStream, FilterOutputStream, FilterReader, and FilterWriter classes themselves aren't useful; they must be subclassed and specialized to create a new type of filtering operation. For example, specialized wrapper streams like DataInputStream and DataOutputStream provide additional methods for reading and writing primitive data types.

As we said, when you create an instance of a filtered stream, you specify another stream in the constructor. The specialized stream wraps an additional layer of functionality around the other stream, as shown in Figure 8.3. Because filtered streams themselves are subclasses of the fundamental stream types, filtered streams can be layered on top of each other to provide different combinations of features. For example, you could wrap a PushbackReader around a LineNumberReader that was wrapped around a FileReader.

Figure 8.3: Layered streams

[Graphic: Figure 8-3]

Data streams

DataInputStream and DataOutputStream are filtered streams that let you read or write strings and primitive data types that comprise more than a single byte. DataInputStream and DataOutputStream implement the DataInput and DataOutput interfaces, respectively. These interfaces define the methods required for streams that read and write strings and Java primitive types in a machine-independent manner.

You can construct a DataInputStream from an InputStream and then use a method like readDouble() to read a primitive data type:

```
DataInputStream dis = new DataInputStream( System.in );
double d = dis.readDouble();
```

The above example wraps the standard input stream in a DataInputStream and uses it to read a double value. readDouble() reads bytes from the stream and constructs a double from them. All DataInputStream methods that read primitive types also read binary information.

The DataOutputStream class provides write methods that correspond to the read methods in DataInputStream. For example, writeInt() writes an integer in binary format to the underlying output stream.

The readUTF() and writeUTF() methods of DataInputStream and DataOutputStream read and write a Java String of Unicode characters using the UTF-8 "transformation format." UTF-8 is an ASCII-compatible encoding of Unicode characters commonly used for the transmission and storage of Unicode text.[1]

[1] Check out the URL http://www.stonehand.com/unicode/standard/utf8.html for more information on UTF-8.

We can use a DataInputStream with any kind of input stream, whether it be from a file, a socket, or standard input. The same applies to using a DataOutputStream, or, for that matter, any other specialized streams in java.io.

Buffered streams

The BufferedInputStream, BufferedOutputStream, BufferedReader, and BufferedWriter classes add a data buffer of a specified size to the stream path. A buffer can increase efficiency by reducing the number of physical read or write operations that correspond to read() or write() method calls. You create a buffered stream with an appropriate input or output stream and a buffer size. Furthermore, you can wrap another stream around a buffered stream so that it benefits from the buffering. Here's a simple buffered input stream:

```
BufferedInputStream bis = new BufferedInputStream(myInputStream, 4096);
...
bis.read();
```

In this example, we specify a buffer size of 4096 bytes. If we leave off the size of the buffer in the constructor, a reasonably sized one is chosen for us. On our first call to read(), bis tries to fill the entire 4096-byte buffer with data. Thereafter, calls to read() retrieve data from the buffer until it's empty.

A BufferedOutputStream works in a similar way. Calls to write() store the data in a buffer; data is actually written only when the buffer fills up. You can also use the flush() method to wring out the contents of a BufferedOutputStream before the buffer is full.

Some input streams like BufferedInputStream support the ability to mark a location in the data and later reset the stream to that position. The mark() method sets the return point in the stream. It takes an integer value that specifies the number of bytes that can be read before the stream gives up and forgets about the mark. The reset() method returns the stream to the marked point; any data read after the call to mark() is read again.

This functionality is especially useful when you are reading the stream in a parser. You may occasionally fail to parse a structure and so must try something else. In this situation, you can have your parser generate an error (a homemade ParseException) and then reset the stream to the point before it began parsing the structure:

```
BufferedInputStream input;
...
try {
    input.mark( MAX_DATA_STRUCTURE_SIZE );
    return( parseDataStructure( input ) );
}
catch ( ParseException e ) {
    input.reset();
    ...
}
```

The `BufferedReader` and `BufferedWriter` classes work just like their byte-based counterparts, but operate on characters instead of bytes.

Print streams

Another useful wrapper stream is `java.io.PrintWriter`. This class provides a suite of overloaded `print()` methods that turn their arguments into strings and push them out the stream. A complementary set of `println()` methods adds a newline to the end of the strings. `PrintWriter` is the more capable big brother of the `PrintStream` byte stream. `PrintWriter` is an unusual character stream because it can wrap either an `OutputStream` or another `Writer`. The `System.out` and `System.err` streams are `PrintStream` objects; you have already seen such streams strewn throughout this book:

```
System.out.print("Hello world...\n");
System.out.println("Hello world...");
System.out.println( "The answer is: " + 17 );
System.out.println( 3.14 );
```

In Java 1.1, the `PrintStream` class has been enhanced to translate characters to bytes using the system's default encoding scheme. Although `PrintStream` is not deprecated in Java 1.1, its constructors are. For all new development, use a `PrintWriter` instead of a `PrintStream`. Because a `PrintWriter` can wrap an `OutputStream`, the two classes are interchangeable.

When you create a `PrintWriter` object, you can pass an additional `boolean` value to the constructor. If this value is `true`, the `PrintWriter` automatically performs a `flush()` on the underlying `OutputStream` or `Writer` each time it sends a newline:

```
boolean autoFlush = true;
PrintWriter p = new PrintWriter( myOutputStream, autoFlush );
```

When this technique is used with a buffered output stream, it corresponds to the behavior of terminals that send data line by line.

Unlike methods in other stream classes, the methods of `PrintWriter` and `PrintStream` do not throw `IOException`s. Instead, if we are interested, we can check for errors with the `checkError()` method:

```
System.out.println( reallyLongString );
if ( System.out.checkError() )                // Uh oh
```

Pipes

Normally, our applications are directly involved with one side of a given stream at a time. `PipedInputStream` and `PipedOutputStream` (or `PipedReader` and `PipedWriter`), however, let us create two sides of a stream and connect them together, as shown in Figure 8.4. This provides a stream of communication between threads, for example.

To create a pipe, we use both a `PipedInputStream` and a `PipedOutputStream`. We can simply choose a side and then construct the other side using the first as an argument:

Figure 8.4: Piped streams

[Graphic: Figure 8-4]

```
PipedInputStream pin = new PipedInputStream();
PipedOutputStream pout = new PipedOutputStream( pin );
```

Alternatively :

```
PipedOutputStream pout = new PipedOutputStream( );
PipedInputStream pin = new PipedInputStream( pout );
```

In each of these examples, the effect is to produce an input stream, `pin`, and an output stream, `pout`, that are connected. Data written to `pout` can then be read by `pin`. It is also possible to create the `PipedInputStream` and the `PipedOutputStream` separately, and then connect them with the `connect()` method.

We can do exactly the same thing in the character-based world, using `PipedReader` and `PipedWriter` in place of `PipedInputStream` and `PipedOutputStream`.

Once the two ends of the pipe are connected, use the two streams as you would other input and output streams. You can use `read()` to read data from the `PipedInputStream` (or `PipedReader`) and `write()` to write data to the `PipedOutputStream` (or `PipedWriter`). If the internal buffer of the pipe fills up, the writer blocks and waits until more space is available. Conversely, if the pipe is empty, the reader blocks and waits until some data is available. Internally, the blocking is implemented with `wait()` and `notifyAll()`, as described in Chapter 6, *Threads*.

One advantage to using piped streams is that they provide stream functionality in our code, without compelling us to build new, specialized streams. For example, we can use pipes to create a simple logging facility for our application. We can send messages to the logging facility through an ordinary `PrintWriter`, and then it can do whatever processing or buffering is required before sending the messages off to their ultimate location. Because we are dealing with string messages, we use the character-based `PipedReader` and `PipedWriter` classes. The following example shows the skeleton of our logging facility:

```
import java.io.*;

class LoggerDaemon extends Thread {
    PipedReader in = new PipedReader();

    LoggerDaemon() {
        setDaemon( true );
        start();
    }

    public void run() {
        BufferedReader din = new BufferedReader( in );
        String s;

        try {
           while ( (s = din.readLine()) != null ) {
                // process line of data
                // ...
            }
        }
        catch (IOException e ) { }
    }

    PrintWriter getWriter() throws IOException {
        return new PrintWriter( new PipedWriter( in ) );
    }
}

class myApplication {
    public static void main ( String [] args ) throws IOException {
        PrintWriter out = new LoggerDaemon().getWriter();

        out.println("Application starting...");
        // ...
        out.println("Warning: does not compute!");
        // ...
    }
}
```

`LoggerDaemon` is a daemon thread, so it will die when our application exits. `LoggerDaemon` reads strings from its end of the pipe, the `PipedReader in`. `LoggerDaemon` also provides a method, `getWriter()`, that returns a `PipedWriter` that is connected to its input stream. Simply create a new `LoggerDaemon` and fetch the output stream to begin sending messages.

In order to read strings with the `readLine()` method, `LoggerDaemon` wraps a `BufferedReader` around its `PipedReader`. For convenience, it also presents its `PipedWriter` as a `PrintWriter`, rather than a simple `Writer`.

One advantage of implementing `LoggerDaemon` with pipes is that we can log messages as easily as we write text to a terminal or any other stream. In other words, we can use all our normal tools and techniques. Another advantage is that the processing happens in another thread, so we can go about our business while the processing takes place.

There is nothing stopping us from connecting more than two piped streams. For example, we could chain multiple pipes together to perform a series of filtering operations.

Strings to Streams and Back

The `StringReader` class is another useful stream class. The stream is created from a `String`; `StringReader` essentially wraps stream functionality around a `String`. Here's how to use a `StringReader`:

```
String data = "There once was a man from Nantucket...";
```

```
StringReader sr = new StringReader( data );

char T = (char)sr.read();
char h = (char)sr.read();
char e = (char)sr.read();
```

Note that you will still have to catch `IOException`s thrown by some of the `StringReader`'s methods.

The `StringReader` class is useful when you want to read data in a `String` as if it were coming from a stream, such as a file, pipe, or socket. For example, suppose you create a parser that expects to read tokens from a stream. But you want to provide a method that also parses a big string. You can easily add one using `StringReader`.

Turning things around, the `StringWriter` class lets us write to a character string through an output stream. The internal string grows as necessary to accommodate the data. In the following example, we create a `StringWriter` and wrap it in a `PrintWriter` for convenience:

```
StringWriter buffer = new StringWriter();
PrintWriter out = new PrintWriter( buffer );

out.println("A moose once bit my sister.");
out.println("No, really!");

String results = buffer.toString();
```

First we print a few lines to the output stream, to give it some data, then retrieve the results as a string with the `toString()` method. Alternately, we could get the results as a `StringBuffer` with the `getBuffer()` method.

The `StringWriter` class is useful if you want to capture the output of something that normally sends output to a stream, such as a file or the console. A `PrintWriter` wrapped around a `StringWriter` competes with `StringBuffer` as the easiest way to construct large strings piece by piece. While using a `StringBuffer` is more efficient, `PrintWriter` provides more functionality than the normal `append()` method used by `StringBuffer`.

rot13InputStream

Before we leave streams, let's try our hand at making one of our own. I mentioned earlier that specialized stream wrappers are built on top of the `FilterInputStream` and `FilterOutputStream` classes. It's quite easy to create our own subclass of `FilterInputStream` that can be wrapped around other streams to add new functionality.

The following example, `rot13InputStream`, performs a *rot13* operation on the bytes that it reads. *rot13* is a trivial algorithm that shifts alphanumeric letters to make them not quite human-readable; it's cute because it's symmetric. That is, to "un-rot13" some text, simply *rot13* it again. We'll use the `rot13InputStream` class again in the `crypt` protocol handler example in Chapter 9, *Network Programming*, so we've put the class in the `example.io` package to facilitate reuse. Here's our `rot13InputStream` class:

```
package example.io;
import java.io.*;

public class rot13InputStream extends FilterInputStream {

    public rot13InputStream ( InputStream i ) {
        super( i );
    }
    public int read() throws IOException {
        return rot13( in.read() );
    }

    private int rot13 ( int c ) {
        if ( (c >= 'A') && (c <= 'Z') )            c=(((c-'A')+13)%26)+'A';        if
( (c >= 'a') && (c <= 'z') )
            c=(((c-'a')+13)%26)+'a';
        return c;
    } }
```

The `FilterInputStream` needs to be initialized with an `InputStream`; this is the stream to be filtered. We provide an appropriate constructor for the `rot13InputStream` class and invoke the parent constructor with a call to `super()`. `FilterInputStream` contains a protected instance variable, `in`, where it stores the stream reference and makes it available to the rest of our class.

The primary feature of a `FilterInputStream` is that it overrides the normal `InputStream` methods to delegate calls to the `InputStream` in the variable `in`. So, for instance, a call to `read()` simply turns around and calls `read()` on `in` to fetch a byte. An instance of `FilterInputStream` itself could be instantiated from an `InputStream`; it would pass its method calls on to that stream and serve as a pass-through filter. To make things interesting, we can override methods of the `FilterInputStream` class and do extra work on the data as it passes through.

In our example, we have overridden the `read()` method to fetch bytes from the underlying `InputStream`, `in`, and then perform the *rot13* shift on the data before returning it. Note that the `rot13()` method shifts alphabetic characters, while simply passing all other values, including the end of stream value (`-1`). Our subclass now acts like a *rot13* filter. All other normal functionality of an `InputStream`, like `skip()` and `available()` is unmodified, so calls to these methods are answered by the underlying `InputStream`.

Strictly speaking, `rot13InputStream` only works on an ASCII byte stream, since the underlying algorithm is based on the Roman alphabet. A more generalized character scrambling algorithm would have to be based on `FilterReader` to handle Unicode correctly.

9. Network Programming

Contents:

The network is the soul of Java. Most of what is new and exciting about Java centers around the potential for new kinds of dynamic, networked applications. This chapter discusses the `java.net` package, which contains classes for communications and working with networked resources. These classes fall into two categories: the sockets API and classes for working with Uniform Resource Locators (URLs). Figure 9.1 shows all of the classes in `java.net`.

Figure 9.1: The java.net package

[Graphic: Figure 9-1]

Java's sockets interface provides access to the standard network protocols used for communications between hosts on the Internet. Sockets are the mechanism underlying all other kinds of portable networked communications. Your processes can use sockets to communicate with a server or peer applications on the Net, but you have to implement your own application-level protocols for handling and interpreting the data. Higher-level functionality, like remote procedure calls and distributed objects, are implemented with sockets.

The Java URL classes provide an API for accessing well-defined networked resources, like documents and applications on servers. The classes use an extensible set of prefabricated protocol and content handlers to perform the necessary communication and data conversion for accessing URL resources. With URLs, an application can fetch a complete file or database record from a server on the network with just a few lines of code. Applications like Web browsers, which deal with networked content, use the `URL` class to simplify the task of network programming. They also take advantage of the dynamic nature of Java, which allows handlers for new types of URLs to be added on the fly. As new types of servers and new formats for content evolve, additional URL handlers can be supplied to retrieve and interpret the data without modifying the original application.

In this chapter, I'll try to provide some practical and realistic examples of Java network programming using both APIs. Sadly, the current state of affairs is disappointing. The real release of HotJava isn't available, and Netscape Navigator imposes many restrictions on what you can do. In addition, a few standards that we need haven't been defined. Nevertheless, you can use all of Java's networking capabilities to build your own free-standing applications. I'll point out the shortcomings with Netscape Navigator and the standards scene as I go along.

9.1 Sockets

Sockets are a low-level programming interface for networked communications. They send streams of data between applications that may or may not be on the same host. Sockets originated in BSD UNIX and are, in other languages, hairy and complicated things with lots of small parts that can break off and choke little children. The reason for this is that most socket APIs can be used with almost any kind of underlying network protocol. Since the protocols that transport data across the network can have radically different features, the socket interface can be quite complex. (For a discussion of sockets in general, see *UNIX Network Programming*, by Richard Stevens [Prentice-Hall].)

Java supports a simplified object-oriented interface to sockets that makes network communications considerably easier. If you have done network programming using sockets in C or another structured language, you should be pleasantly surprised at how simple things can be when objects encapsulate the gory details. If this is the first time you've come across sockets, you'll find that talking to another application can be as simple as reading a file or getting user input. Most forms of I/O in Java, including network I/O, use the stream classes described in Chapter 8, *Input/Output Facilities*. Streams provide a unified I/O interface; reading or writing across the Internet is similar to reading or writing a file on the local system.

Java provides different kinds of sockets to support two distinct classes of underlying protocols. In this first section, we'll look at Java's `Socket` class, which uses a *connection-oriented* protocol. A connection-oriented protocol gives you the equivalent of a telephone conversation; after establishing a connection, two applications can send data back and forth; the connection stays in place even when no one is talking. The protocol ensures that no data is lost and that it always arrives in order. In the next section we'll look at the `DatagramSocket` class, which uses a *connectionless* protocol. A connectionless protocol is more like the postal service. Applications can send short messages to each other, but no attempt is made to keep the connection open between messages, to keep the messages in order, or even to guarantee that they arrive.

In theory, just about any protocol family can be used underneath the socket layer: Novell's IPX, Apple's AppleTalk, even the old ChaosNet protocols. But this isn't a theoretical world. In practice, there's only one protocol family people care about on the Internet, and only one protocol family Java supports: the Internet protocols, IP. The `Socket` class speaks TCP, and the `DatagramSocket` class speaks UDP, both standard Internet protocols. These protocols are available on any system that is connected to the Internet.

Clients and Servers

When writing network applications, it's common to talk about clients and servers. The distinction is increasingly vague, but the side that initiates the conversation is usually the *client*. The side that accepts the request to talk is usually the *server*. In the case where there are two peer applications using sockets to talk, the distinction is less important, but for simplicity we'll use the above definition.

For our purposes, the most important difference between a client and a server is that a client can create a socket to initiate a conversation with a server application at any time, while a server must prepare to listen for incoming conversations in advance. The `java.net.Socket` class represents a single side of a socket connection on either the client or server. In addition, the server uses the `java.net.ServerSocket` class to wait for connections from clients. An application acting as a server creates a `ServerSocket` object and waits, blocked in a call to its `accept()` method, until a connection arrives. When it does, the `accept()` method creates a `Socket` object the server uses to communicate with the client. A server carries on multiple conversations at once; there is only a single `ServerSocket`, but one active `Socket` object for each client, as shown in Figure 9.2.

Figure 9.2: Clients and servers, Sockets and ServerSockets

[Graphic: Figure 9-2]

A client needs two pieces of information to locate and connect to another server on the Internet: a hostname (used to find the host's network address) and a port number. The port number is an identifier that differentiates between multiple clients or servers on the same host. A server application listens on a prearranged port while waiting for connections. Clients select the port number assigned to the service they want to access. If you think of the host computers as hotels and the applications as guests, then the ports are like the guests' room numbers. For one guest to call another, he or she must know the other party's hotel name and room number.

Clients

A client application opens a connection to a server by constructing a `Socket` that specifies the hostname and port number of the desired server:

```
try {
    Socket sock = new Socket("wupost.wustl.edu", 25);
}
catch ( UnknownHostException e ) {
    System.out.println("Can't find host.");
}
catch ( IOException e ) {
    System.out.println("Error connecting to host.");
}
```

This code fragment attempts to connect a `Socket` to port 25 (the SMTP mail service) of the host wupost.wustl.edu. The client handles the possibility that the hostname can't be resolved (`UnknownHostException`) and that it might not be able to connect to it (`IOException`).

As an alternative to using a hostname, you can provide a string version of the host's IP address:

```
Socket sock = new Socket("128.252.120.1", 25);    // wupost.wustl.edu
```

Once a connection is made, input and output streams can be retrieved with the `Socket` `getInputStream()` and `getOutputStream()` methods. The following (rather arbitrary and strange) conversation illustrates sending and receiving some data with the streams. Refer to Chapter 8, *Input/Output Facilities* for a complete discussion of working with streams.

```
try {
    Socket server = new Socket("foo.bar.com", 1234);
    InputStream in = server.getInputStream();
    OutputStream out = server.getOutputStream();

    // Write a byte
    out.write(42);

    // Say "Hello" (send newline delimited string)
    PrintStream pout = new PrintStream( out );
    pout.println("Hello!");

    // Read a byte
    Byte back = in.read();

    // Read a newline delimited string
    DataInputStream din = new DataInputStream( in );
    String response = din.readLine();
    server.close();
}
catch (IOException e ) { }
```

In the exchange above, the client first creates a `Socket` for communicating with the server. The `Socket` constructor specifies the server's hostname (foo.bar.com) and a prearranged port number (1234). Once the connection is established, the client writes a single byte to the server using the `OutputStream`'s `write()` method. It then wraps a `PrintStream` around the `OutputStream` in order to send text more easily. Next, it performs the complementary operations, reading a byte from the server using `InputStream`'s `read()` and then creating a `DataInputStream` from which to get a string of text. Finally, it terminates the connection with the `close()` method. All these operations have the potential to generate `IOExceptions`; the `catch` clause is where our application would deal with these.

Servers

After a connection is established, a server application uses the same kind of `Socket` object for its side of the communications. However, to accept a connection from a client, it must first create a `ServerSocket`, bound to the correct port. Let's recreate the previous conversation from the server's point of view:

```
// Meanwhile, on foo.bar.com...
try {
    ServerSocket listener = new ServerSocket( 1234 );

    while ( !finished ) {
        Socket aClient = listener.accept();    // wait for connection

        InputStream in = aClient.getInputStream();
        OutputStream out = aClient.getOutputStream();

        // Read a byte
        Byte importantByte = in.read();

        // Read a string
        DataInputStream din = new DataInputStream( in );
        String request = din.readLine();

        // Write a byte
        out.write(43);

        // Say "Goodbye"
        PrintStream pout = new PrintStream( out );
        pout.println("Goodbye!");

        aClient.close();
    }

    listener.close();

}
catch (IOException e ) { }
```

First, our server creates a `ServerSocket` attached to port 1234. On some systems there are rules about what ports an application can use. Port numbers below 1024 are usually reserved for system processes and standard, well-known services, so we pick a port number outside of this range. The `ServerSocket` need be created only once. Thereafter we can accept as many connections as arrive.

Next we enter a loop, waiting for the `accept()` method of the `ServerSocket` to return an active `Socket` connection from a client. When a connection has been established, we perform the server side of our dialog, then close the connection and return to the top of the loop to wait for another connection. Finally, when the server application wants to stop listening for connections altogether, it calls the `close()` method of the `ServerSocket`.[1]

> [1] A somewhat obscure security feature in TCP/IP specifies that if a server socket actively closes a connection while a client is connected, it may not be able to bind (attach itself) to the same port on the server host again for a period of time (the maximum time to live of a packet on the network). It's possible to turn off this feature, and it's likely that your Java implementation will have done so.

As you can see, this server is single-threaded; it handles one connection at a time; it doesn't call `accept()` to listen for a new connection until it's finished with the current connection. A more realistic server would have a loop that accepts connections concurrently and passes them off to their own threads for processing. (Our tiny HTTP daemon in a later section will do just this.)

Sockets and security

The examples above presuppose the client has permission to connect to the server, and that the server is allowed to listen on the specified socket. This is not always the case. Specifically, applets and other applications run under the auspices of a `SecurityManager` that can impose arbitrary restrictions on what hosts they may or may not talk to, and whether they can listen for connections. The security policy imposed by the current version of Netscape Navigator allows applets to open socket connections only to the host that served them. That is, they can talk back only to the server from which their class files were retrieved. Applets are not allowed to open server sockets themselves.

Now, this doesn't meant an applet can't cooperate with its server to communicate with anyone, anywhere.

A server could run a proxy that lets the applet communicate indirectly with anyone it likes. What the current security policy prevents is malicious applets roaming around inside corporate firewalls. It places the burden of security on the originating server, and not the client machine. Restricting access to the originating server limits the usefulness of "trojan" applications that do annoying things from the client side. You won't let your proxy mail bomb people, because you'll be blamed.

The DateAtHost Client

Many networked workstations run a time service that dispenses their local clock time on a well-known port. This was a precursor of NTP, the more general Network Time Protocol. In the next example, `DateAtHost`, we'll make a specialized subclass of `java.util.Date` that fetches the time from a remote host instead of initializing itself from the local clock. (See Chapter 7, *Basic Utility Classes* for a complete discussion of the `Date` class.)

`DateAtHost` connects to the time service (port 37) and reads four bytes representing the time on the remote host. These four bytes are interpreted as an integer representing the number of seconds since the turn of the century. `DateAtHost` converts this to Java's variant of the absolute time (milliseconds since January 1, 1970, a date that should be familiar to UNIX users) and then uses the remote host's time to initialize itself:

```
import java.net.Socket;
import java.io.*;

public class DateAtHost extends java.util.Date {
    static int timePort = 37;
    static final long offset = 2208988800L; // Seconds from century to
                                            // Jan 1, 1970 00:00 GMT

    public DateAtHost( String host ) throws IOException {
        this( host, timePort );
    }

    public DateAtHost( String host, int port ) throws IOException {
        Socket sock = new Socket( host, port );
        DataInputStream din =
            new DataInputStream(sock.getInputStream());
        int time = din.readInt();
        sock.close();

        setTime( (((1L << 32) + time) - offset) * 1000 );
    } }
```

That's all there is to it. It's not very long, even with a few frills. We have supplied two possible constructors for `DateAtHost`. Normally we'll use the first, which simply takes the name of the remote host as an argument. The second, overloaded constructor specifies the hostname and the port number of the remote time service. (If the time service were running on a nonstandard port, we would use the second constructor to specify the alternate port number.) This second constructor does the work of making the connection and setting the time. The first constructor simply invokes the second (using the `this()` construct) with the default port as an argument. Supplying simplified constructors that invoke their siblings with default arguments is a common and useful technique.

The second constructor opens a socket to the specified port on the remote host. It creates a `DataInputStream` to wrap the input stream and then reads a 4-byte integer using the `readInt()` method. It's no coincidence the bytes are in the right order. Java's `DataInputStream` and `DataOutputStream` classes work with the bytes of integer types in *network byte order* (most significant to least significant). The time protocol (and other standard network protocols that deal with binary data) also uses the network byte order, so we don't need to call any conversion routines. (Explicit data conversions would probably be necessary if we were using a nonstandard protocol, especially when talking to a non-Java client or server.) After reading the data, we're finished with the socket, so we close it, terminating the connection to the server. Finally, the constructor initializes the rest of the object by calling `Date`'s `setTime()` method with the calculated time value.[2]

[2] The conversion first creates a long value, which is the unsigned equivalent of the integer `time`. It subtracts an offset to make the time relative to the epoch (January 1, 1970) rather than the century, and multiples by 1000 to convert to milliseconds.

The `DateAtHost` class can work with a time retrieved from a remote host almost as easily as `Date` is used with the time on the local host. The only additional overhead is that we have to deal with the possible `IOException` that can be thrown by the `DateAtHost` constructor:

```
try {
    Date d = new DateAtHost( "sura.net" );
    System.out.println( "The time over there is: " + d );
    int hours = d.getHours();
    int minutes = d.getMinutes();
    ...
}
catch ( IOException e ) { }
```

This example fetches the time at the host sura.net and prints its value. It then looks at some components of the time using the `getHours()` and `getMinutes()` methods of the `Date` class.

The TinyHttpd Server

Have you ever wanted your very own Web server? Well, you're in luck. In this section, we're going to build TinyHttpd, a minimal but functional HTTP daemon. TinyHttpd listens on a specified port and services simple HTTP "get file" requests. They look something like this:

```
GET /path/filename [optional stuff]
```

Your Web browser sends one or more as lines for each document it retrieves. Upon reading the request, the server tries to open the specified file and send its contents. If that document contains references to images or other items to be displayed inline, the browser continues with additional GET requests. For best performance (especially in a time-slicing environment), TinyHttpd services each request in its own thread. Therefore, TinyHttpd can service several requests concurrently.

Over and above the limitations imposed by its simplicity, TinyHttpd suffers from the limitations imposed by the fickleness of filesystem access, as discussed in Chapter 8, *Input/Output Facilities*. It's important to remember that file pathnames are still architecture dependent--as is the concept of a filesystem to begin with. This example should work, as is, on UNIX and DOS-like systems, but may require some customizations to account for differences on other platforms. It's possible to write more elaborate code that uses the environmental information provided by Java to tailor itself to the local system. (Chapter 8, *Input/Output Facilities* gives some hints about how to do this).

WARNING:

This example will serve files from your host without protection. Don't try this at work.

Now, without further ado, here's TinyHttpd:

```
import java.net.*;
import java.io.*;
import java.util.*;

public class TinyHttpd {
    public static void main( String argv[] ) throws IOException {
        ServerSocket ss = new ServerSocket(Integer.parseInt(argv[0]));
        while ( true )
            new TinyHttpdConnection( ss.accept() );
    }
}

class TinyHttpdConnection extends Thread {
    Socket sock;
    TinyHttpdConnection ( Socket s ) {
        sock = s;
        setPriority( NORM_PRIORITY - 1 );
        start();
    }

    public void run() {
        try {
            OutputStream out = sock.getOutputStream();
            String req =
                new DataInputStream(sock.getInputStream()).readLine();
            System.out.println( "Request: "+req );

            StringTokenizer st = new StringTokenizer( req );
            if ( (st.countTokens() >= 2) &&
                  st.nextToken().equals("GET") ) {
                if ( (req = st.nextToken()).startsWith("/") )
                    req = req.substring( 1 );
                if ( req.endsWith("/") || req.equals("") )
                    req = req + "index.html";

                try {
                    FileInputStream fis = new FileInputStream ( req );
                    byte [] data = new byte [ fis.available() ];
                    fis.read( data );
                    out.write( data );
                }
                catch ( FileNotFoundException e )
                    new PrintStream( out ).println("404 Not Found");
            } else
                new PrintStream( out ).println( "400 Bad Request" );

            sock.close();
        }
        catch ( IOException e )
            System.out.println( "I/O error " + e );
    }
}
```

Compile `TinyHttpd` and place it in your class path. Go to a directory with some interesting documents and start the daemon, specifying an unused port number as an argument. For example:

```
% java TinyHttpd 1234
```

You should now be able to use your Web browser to retrieve files from your host. You'll have to specify the nonstandard port number in the URL. For example, if your hostname is foo.bar.com, and you started the server as above, you could reference a file as in:

```
http://foo.bar.com:1234/welcome.html
```

`TinyHttpd` looks for files relative to its current directory, so the pathnames you provide should be relative to that location. Retrieved some files? Al'righty then, let's take a closer look.

`TinyHttpd` is comprised of two classes. The public `TinyHttpd` class contains the `main()` method of our standalone application. It begins by creating a `ServerSocket`, attached to the specified port. It then loops, waiting for client connections and creating instances of the second class, a `TinyHttpdConnection` thread, to service each request. The `while` loop waits for the `ServerSocket` `accept()` method to return a new `Socket` for each client connection. The `Socket` is passed as an argument to construct the `TinyHttpdConnection` thread that handles it.

`TinyHttpdConnection` is a subclass of `Thread`. It lives long enough to process one client connection and then dies. `TinyHttpdConnection`'s constructor does three things. After saving the `Socket` argument for its caller, it adjusts its own priority and then invokes `start()` to bring its `run()` method to life. By lowering its priority to `NORM_PRIORITY-1` (just below the default priority), we ensure that the threads servicing established connections won't block `TinyHttpd`'s main thread from accepting new requests. (On a time-slicing system, this is less important.)

The body of `TinyHttpdConnection`'s `run()` method is where all the magic happens. First, we fetch an `OutputStream` for talking back to our client. The second line reads the `GET` request from the `InputStream` into the variable `req`. This request is a single newline-terminated `String` that looks like the `GET` request we described earlier. Since this is the only time we read from this socket, it's hard to resist the urge to be terse. Alternatively, we could break that statement into three steps: getting the `InputStream`, creating the `DataInputStream` wrapper, and reading the line. The three-line version is certainly more readable and should not be noticeably slower.

We then parse the contents of `req` to extract a filename. The next few lines are a brief exercise in string manipulation. We create a `StringTokenizer` and make sure there are at least two tokens. Using `nextToken()`, we take the first token and make sure it's the word `GET`. (If both conditions aren't met, we have an error.) Then we take the next token (which should be a filename), assign it to `req` , and check whether it begins with "/". If so, we use `substring()` to strip the first character, giving us a filename relative to the current directory. If it doesn't begin with "/", the filename is already relative to the current directory. Finally, we check to see if the requested filename looks like a directory name (i.e., ends in slash) or is empty. In these cases, we append the familiar default filename *index.html*.

Once we have the filename, we try to open the specified file and load its contents into a large byte array. (We did something similar in the `ListIt` example in Chapter 8, *Input/Output Facilities*.) If all goes well, we write the data out to client on the `OutputStream`. If we can't parse the request or the file doesn't exist, we wrap our `OutputStream` with a `PrintStream` to make it easier to send a textual message. Then we return an appropriate HTTP error message. Finally, we close the socket and return from `run()`, removing our `Thread`.

Taming the daemon

The biggest problem with `TinyHttpd` is that there are no restrictions on the files it can access. With a little trickery, the daemon will happily send any file in your filesystem to the client. It would be nice if we could restrict `TinyHttpd` to files that are in the current directory, or a subdirectory. To make the daemon safer, let's add a security manager. I discussed the general framework for security managers in Chapter 7, *Basic Utility Classes*. Normally, a security manager is used to prevent Java code downloaded over the Net from doing anything suspicious. However, a security manager will serve nicely to restrict file access in a self-contained application.

Here's the code for the security manager class:

```
import java.io.*;

class TinyHttpdSecurityManager extends SecurityManager {

    public void checkAccess(Thread g) { };
    public void checkListen(int port) { };
    public void checkLink(String lib) { };
    public void checkPropertyAccess(String key) { };
    public void checkAccept(String host, int port) { };
    public void checkWrite(FileDescriptor fd) { };
    public void checkRead(FileDescriptor fd) { };

    public void checkRead( String s ) {
        if ( new File(s).isAbsolute() || (s.indexOf("..") != -1) )
            throw new
               SecurityException("Access to file : "+s+" denied.");
    }
}
```

The heart of this security manager is the `checkRead()` method. It checks two things: it makes sure that the pathname we've been given isn't an absolute path, which could name any file in the filesystem; and it makes sure the pathname doesn't have a double dot (`..`) in it, which refers to the parent of the current directory. With these two restrictions, we can be sure (at least on a UNIX or DOS-like filesystem) that we have restricted access to only subdirectories of the current directory. If the pathname is absolute or contains "`..`", `checkRead()` throws a `SecurityException`.

The other do-nothing method implementations--e.g., `checkAccess()`--allow the daemon to do its work without interference from the security manager. If we don't install a security manager, the application runs with no restrictions. However, as soon as we install any security manager, we inherit implementations of many "check" routines. The default implementations won't let you do anything; they just throw a security exception as soon as they are called. We have to open holes so the daemon can do its own work; it still has to accept connections, listen on sockets, create threads, read property lists, etc. Therefore, we override the default checks with routines that allow these things.

Now you're thinking, isn't that overly permissive? Not for this application; after all, `TinyHttpd` never tries to load foreign classes from the Net. The only code we are executing is our own, and it's assumed we won't do anything dangerous. If we were planning to execute untrusted code, the security manager would have to be more careful about what to permit.

Now that we have a security manager, we must modify `TinyHttpd` to use it. Two changes are necessary: we must install the security manager and catch the security exceptions it generates. To install the security manager, add the following code at the beginning of `TinyHttpd`'s `main()` method:

```
System.setSecurityManager( new TinyHttpdSecurityManager() );
```

To catch the security exception, add the following `catch` clause after `FileNotFoundException`'s `catch` clause:

```
catch ( SecurityException e )
    new PrintStream( out ).println( "403 Forbidden" );
```

Now the daemon can't access anything that isn't within the current directory or a subdirectory. If it tries to, the security manager throws an exception and prevents access to the file. The daemon then returns a standard HTTP error message to the client.

`TinyHttpd` still has room for improvement. First, it consumes a lot of memory by allocating a huge array to read the entire contents of the file all at once. A more realistic implementation would use a buffer and send large amounts of data in several passes. `TinyHttpd` also fails to deal with simple things like directories. It wouldn't be hard to add a few lines of code (again, refer to the `ListIt` example in Chapter 8, *Input/Output Facilities*) to read a directory and generate linked HTML listings like most Web servers do.

10. Understand the Abstract Windowing Toolkit

Contents:

The Abstract Windowing Toolkit (AWT), or "another windowing toolkit," as some people affectionately call it, provides a large collection of classes for building graphical user interfaces in Java. With AWT, you can create windows, draw, work with images, and use components like buttons, scrollbars, and pull-down menus in a platform independant way. The `java.awt` package contains the AWT GUI classes. The `java.awt.image` package provides some additional tools for working with images.

AWT is the largest and most complicated part of the standard Java distribution, so it shouldn't be any surprise that we'll take several chapters (five, to be precise) to discuss it. Here's the lay of the land:

- Chapter 10, *Understand the Abstract Windowing Toolkit* covers the basic concepts you need to understand how AWT builds user interfaces.
- In Chapter 11, *Using and Creating GUI Components*, we discuss the basic components from which user interfaces are built: lists, text fields, checkboxes, and so on.
- Chapter 12, *Layout Managers* discusses layout managers, which are responsible for arranging components within a display.
- Chapter 13, *Drawing With the AWT* discusses the fundamentals of drawing, including simple image displays.
- Chapter 14, *Working With Images*, the last chapter to discuss the AWT in detail, covers the image generation and processing tools that are in the `java.awt.image` package. We'll throw in audio for good measure.

We can't cover the full functionality of AWT in this book; if you want complete coverage, see the Java AWT Reference (O'Reilly). Instead, we'll cover the basics of the tools you are most likely to use and show some examples of what can be done with some of the more advanced features. Figure 10.1 shows the user interface portion of the `java.awt` package.

Figure 10.1: User-interface classes of the java.awt package

[Graphic: Figure 10-1]

As its name suggests, AWT is an abstraction. Its classes and functionality are the same for all Java implementations, so Java applications built with AWT should work in the same way on all platforms. You could choose to write your code on under Windows NT/95, and then run it on an X Window System, or a Macintosh. To achieve platform independence, AWT uses interchangeable toolkits that interact with the host windowing system to display user-interface components, thus shielding your application code from the details of the environment it's running in. Let's say you ask AWT to create a button. When your application or applet runs, a toolkit appropriate to the host environment renders the button appropriately: on Windows, you can get a button that looks like other Windows buttons; on a Macintosh, you can get a Mac button; and so on.

Working with user-interface components in AWT is meant to be easy. While the low-level (possibly native) GUI toolkits may be complex, you won't have to work with them directly unless you want to port AWT to a new platform or provide an alternative "look and feel" for the built-in components. When building a user interface for your application, you'll be working with prefabricated components. It's easy to assemble a collection of user-interface components (buttons, text areas, etc.), and arrange them inside containers to build complex layouts. You can also use simple components as building blocks for making entirely new kinds of interface gadgets that are completely portable and reusable.

AWT uses layout managers to arrange components inside containers and control their sizing and positioning. Layout managers define a strategy for arranging components instead of relying on absolute positioning. For example, you can define a user interface with a collection of buttons and text areas and be reasonably confident it will always display correctly. It doesn't matter that Windows, UNIX, and the Macintosh render your buttons and text areas differently; the layout manager should still position them sensibly with respect to each other.

Unfortunately, the reality is that most of the complaints about Java center around AWT. AWT is very different from what many people are used to and lacks some of the advanced features other GUI environments provide (at least for now). It's also true that most of the bugs in current implementations of Java lie in the AWT toolkits. As bugs are fixed and developers become accustomed to AWT, we would expect the number of complaints to diminish. Java 1.1 is a big improvement over previous versions. But at the time of this writing, there are some rough edges.

10.1 GUI Concepts in Java

Chapter 11, *Using and Creating GUI Components* contains examples using most of the components in the `java.awt` package. But before we dive into those examples, we need to spend a bit of time talking about the concepts AWT uses for creating and handling user interfaces. This material should get you up to speed on GUI concepts and on how they are used in Java.

Components

A component is the fundamental user-interface object in Java. Everything you see on the display in a Java application is an AWT component. This includes things like windows, drawing canvases, buttons, checkboxes, scrollbars, lists, menus, and text fields. To be used, a component must usually be placed in a Container. Container objects group components, arrange them for display, and associate them with a particular display device. All components are derived from the abstract `java.awt.Component` class, as you saw in Figure 10.1. For example, the `Button` class is a subclass of the `Component` class, which is derived directly from `Object`.

For the sake of simplicity, we can split the functionality of the `Component` class into two categories: appearance and behavior. The `Component` class contains methods and variables that control an object's general appearance. This includes basic attributes such as whether or not it's visible, its current size and location, and certain common graphical defaults like font and color. The `Component` class also contains methods implemented by specific subclasses to produce the actual graphics the object displays. When a component is first displayed, it's associated with a particular display device. The `Component` class encapsulates access to its display area on that device. This includes methods for accessing graphics and for working with off-screen drawing buffers for the display.

By a `Component`'s behavior, we usually mean the way it responds to user-driven events. When the user performs an action (like pressing the mouse button) within a component's display area, an AWT thread delivers an event object that describes "what happened." The event is delivered to objects that have registered themselves as "listeners" for that type of event from that component. For example, when the user clicks on a button, the button delivers an `ActionEvent`. To receive those events, an object registers with the button as an `ActionListener`.

Events are delivered by invoking designated event-handler methods within the receiving object (the "listener"). Objects prepare themselves to receive events by implementing methods for the types of events in which they are interested and then registering as listeners with the sources of the events. There are specific types of events that cover different categories of component and user interaction. For example, `MouseEvent`s describe activities of the mouse within a component's area, `KeyEvent`s describe key presses, and functionally higher level events such as `ActionEvent`s indicate that a user interface component has done its job.

Although events are crucial to the workings of AWT, they aren't limited to building user interfaces. Events are an important interobject communications mechanism that may be used by completely nongraphical parts of an application as well. They are particularly important in the context of Java Beans, which use events as an extremely general notification mechanism. We will describe events thoroughly in this chapter because they are so fundamental to the way in which user interfaces function in Java, but it's good to keep the bigger picture in mind.

Containers often take on certain responsibilities for the components that they hold. Instead of having every component handle events for its own bit of the user interface, a container may register itself or another object to receive the events for its child components, and "glue" those events to the correct application logic.

A component informs its container when it does something that might affect other components in the container, such as changing its size or visibility. The container can then tell the layout manager that it is time to rearrange the child components.

Containers in Java are themselves a kind of component. Because all components share this structure, container objects can manage and arrange `Component` objects without knowing what they are and what they are doing. Components can be swapped and replaced with new versions easily and combined into composite user-interface objects that can be treated as individual components. This lends itself well to building larger, reusable user-interface items.

Peers

We have just described a nice system in which components govern their own appearance, and events are delivered to objects that are "listening" to those components. Unfortunately, getting data out to a display medium and receiving events from input devices involve crossing the line from Java to the real world. The real world is a nasty place full of architecture dependence, local peculiarities, and strange physical devices like mice, trackballs, and '69 Buicks.

At some level, our components will have to talk to objects that contain native methods to interact with the host operating environment. To keep this interaction as clean and well-defined as possible, Java uses a set of *peer* interfaces. The peer interface makes it possible for a pure Java-language AWT component to use a corresponding real component--the peer object--in the native environment. You won't generally deal directly with peer interfaces or the objects behind them; peer handling is encapsulated within the `Component` class. It's important to understand the architecture, though, because it imposes some limitations on what you can do with components.

For example, when a component such as a `Button` is first created and displayed on the screen, code in the `Component` class asks an AWT `Toolkit` class to create a corresponding peer object, as shown in Figure 10.2. The `Toolkit` is a *factory* that knows how to create objects in the native display system; Java uses this factory design pattern to provide an abstraction that separates the implementation of component objects from their functionality. The `Toolkit` object contains methods for making instances of each type of component peer. (As a developer, you will probably never work with a native user-interface directly.) `Toolkit`s can be swapped and replaced to provide new implementations of the components without affecting the rest of Java.

Figure 10.2: A toolkit creating a peer object

[Graphic: Figure 10-2]

Figure 10.2 shows a `Toolkit` producing a peer object for a `Button`. When you add a button to a container, the container calls the `Button`'s `addNotify()` method. In turn, `addNotify()` calls the `Toolkit`'s `createButton()` method to make the `Button`'s peer object in the real display system. Thereafter, the component class keeps a reference to the peer object and delegates calls to its methods.

The `java.awt.peer` package, shown in Figure 10.3, parallels the `java.awt` package and contains an interface for each type of component. For example, the `Button` component has a `ButtonPeer` interface, which defines the capabilities of a button.

Figure 10.3: The java.awt.peer package

[Graphic: Figure 10-3]

The peer objects themselves can be built in whatever way is necessary, using a combination of Java and native code. (We will discuss the implementation of peers a bit more in the AWT performance section of this chapter.) A Java-land button doesn't know or care how the real-world button is implemented or what additional capabilities it may have; it knows only about the existence of the methods defined in the `ButtonPeer` interface. Figure 10.4 shows a call to the `setLabel()` method of a `Button` object, which results in a call to the corresponding `setLabel()` method of the native button object.

Figure 10.4: Invoking a method in the peer interface

[Graphic: Figure 10-4]

In this case, the only action a button peer must be able to perform is to set its label text; `setLabel()` is the only method in the ButtonPeer interface. How the native button acts, responds to user input, etc. is entirely up to it. It might turn green when pressed or make a "ka-chunk" sound. The component in Java-land has no control over these aspects of the native button's behavior--and this has important implications. This abstraction allows AWT to use native components from whatever platform it resides on. However, it also means that a lot of a component's functionality is locked away where we can't get to it. We'll see that we can usually intercept events before the peer object has a chance to act on them, but we usually can't change much of the object's basic behavior.

A component gets its peer when it's added to a container. Containers are associated with display devices through Toolkit objects, and thus control the process of peer creation. We'll talk more the ramifications of this when we discuss addNotify() later. (See "Component peers and addNotify()".)

The Model/View/Controller Framework

Before we continue our discussion of GUI concepts, I want to make a brief aside and talk about the Model/View/Controller (MVC) framework. MVC is a method of building reusable components that logically separates the structure, representation, and behavior of a component into separate pieces. MVC is primarily concerned with building user-interface components, but the basic ideas can be applied to many design issues; its principles can be seen throughout Java. Java doesn't implement all of MVC, whose origins are in Smalltalk, but MVC's influence is apparent throughout the language.

The fundamental idea behind MVC is the separation of the data model for an item from its presentation. For example, we can draw different representations (e.g., bar graphs, pie charts) of the data in a spreadsheet. The data is the "model"; the particular representation is the "view." A single model can have many views that show different representations of the data. The behavior of a user-interface component usually changes its model and affects how the component is rendered as a view. If we were to create a button component for example, its data model could be as simple as a boolean value for whether it's up or down. The behavior for handling mouse-press events would alter the model, and the display would examine that data when it draws the on-screen representation.

The way in which AWT objects communicate, by passing events from sources to listeners, is part of this MVC concept of separation. Event listeners are "observers" and event sources are "observables." When an observable changes or performs a function, it notifies all of its observers of the activity.[1]

> [1] Although they are not used by AWT, Java provides generic Observer class and Observable interface in the java.util package. (In practice, these are more a design hint than of day to day usefulness.)

This model of operation is also central to the way in which Java works with graphics, as you'll see in Chapter 11, *Using and Creating GUI Components*. Image information from a producer, such as a graphics engine or video stream, is distributed to consumers that can represent different views of the data. Consumers register with a producer and receive updates as the image is created or when the image has changed.

The factory concept used by the Toolkit objects is related to MVC; factories use Java interfaces to separate the implementation of an object from its behavior. An object that implements an interface doesn't have to fit into a particular class structure; it needs only to provide the methods defined by the interface. Thus, an AWT Toolkit is a factory that produces native user-interface components that correspond to Java components. The native components don't need to match AWT's class structure, provided they implement the appropriate interface.

Painting and Updating

Components can be asked to draw themselves at any time. In a more procedural programming environment, you might expect a component would be involved in drawing only when first created or when it changes its appearance. In Java, components act in a way that is more closely tied to the underlying behavior of the display environment. For example, when you obscure a component with a window and then reexpose it, an AWT thread asks the component to redraw itself.

AWT asks a component to draw itself by calling its `paint()` method. `paint()` may be called at any time, but in practice, it's called when the object is first made visible, whenever it changes its appearance, and whenever some tragedy in the display system messes up its area. Because `paint()` can't make any assumptions about why it was called, it must redraw the component's entire display.

However, redrawing the whole component is unnecessary if only a small part changes, especially in an anticipated way. In this case, you'd like to specify what part of the component has changed, and redraw that part alone. Painting a large portion of the screen is time consuming, and can cause flickering that is especially annoying if you're redrawing the object frequently, as with animation. When a component realizes it needs to redraw itself, it should ask AWT to schedule a call to its `update()` method. `update()` can do drawing of its own, but often, it simply defines a clipping region--by calling `clipRect()`--on its graphics context; to limit the extent of the painted area and then calling `paint()` explicitly. A simple component doesn't have to implement its own `update()` method, but that doesn't mean the method doesn't exist. In this case, the component gets a default version of `update()` that simply clears the component's area and calls `paint()`.

A component never calls its `update()` method directly. Instead, when a component requires redrawing, it schedules a call to `update()` by invoking `repaint()`. The `repaint()` method asks AWT to schedule the component for repainting. At some point in the future, a call to `update()` occurs. AWT is allowed to manage these requests in whatever way is most efficient. If there are too many requests to handle, or if there are multiple requests for the same component, AWT can reschedule a number of repaint requests into a single call to `update()`. This means that you can't predict exactly when `update()` will be called in response to a `repaint()`; all you can expect is that it happens at least once, after you request it.

Normally, calling repaint() is an implicit request to be updated as soon as possible. There is another form of `repaint()` that allows you to specify a time period within which you would like an update, giving the system more flexibility in scheduling the request. An application can use this method to govern its refresh rate. For example, the rate at which you render frames for an animation might vary, depending on other factors (like the complexity of the image). You could impose an effective maximum frame rate by calling repaint() with a time (the inverse of the frame rate) as an argument. If you then happen to make more than one repaint request within that time period, AWT is not obliged to physically repaint for each one. It can simply condense them to carry out a single update within the time you have specified.

Both `paint()` and `update()` take a single argument: a `Graphics` object. The `Graphics` object represents the component's graphics context. It corresponds to the area of the screen on which the component can draw and provides the methods for performing primitive drawing and image manipulation. We'll look at the `Graphics` class in detail in Chapter 11, *Using and Creating GUI Components*.

All components paint and update themselves using this mechanism. However, you really care about it only when doing your own drawing, and in practice, you should be drawing only on a `Canvas`, `Panel`, `Applet`, or your own subclasses of Component. Other kinds of objects, like buttons and scrollbars, have lots of behavior built into their peers. You may be able to draw on one of these objects, but unless you specifically catch the appropriate events and redraw (which could get complicated), your handiwork is likely to disappear.

`Canvas`es, `Panel`s, and lightweight components (which we will discuss fully later in this chapter) are "blank slates" for you to implement your own behavior and appearance. For example, by itself, the AWT `Canvas` has no outward appearance; it takes up space and has a background color, but otherwise, it's empty. By subclassing `Canvas` and adding your own code, you can create a more complicated object like a graph or a flying toaster. A lightweight component is even "emptier" than that. It doesn't have a real `Toolkit` peer for its implementation; you get to specify all of the behavior and appearance yourself. We'll talk more about using `Canvas` and lightweight components to create new kinds of GUI objects later in this chapter.

A `Panel` is like a `Canvas`, but it's also a `Container`, so that it can hold other user-interface components. In the same way, a lightweight container is a simple extension of the AWT `Container` class. (More about that when we talk about containers below.)

Enabling and Disabling Components

Standard AWT components can be turned on and off by calling the `enable()` and `disable()` methods. When a component like a `Button` or `TextField` is disabled, it becomes "ghosted" or "greyed-out" and doesn't respond to user input.

For example, let's see how to create a component that can only be used once. This requires getting ahead of the story; I won't explain some aspects of this example until later. Earlier, I said that a `Button` generates an `ActionEvent` when it is pressed. This event is delivered to the listeners' `actionPerformed()` method. The code below disables whatever component generated the event:

```
public boolean void actionPerformed(ActionEvent e ) {
    ...
    ((Component)e.getSource()).disable();
}
```

This code calls `getSource()` to find out which component generated the event. We cast the result to `Component` because we don't necessarily know what kind of component we're dealing with; it might not be a button, because other kinds of components can generate action events. Once we have our component, we disable it.

You can also disable an entire container. Disabling a `Panel`, for instance, disables all the components it contains. Unfortunately, disabling components and containers is handled by the AWT Toolkit at a low level. It is currently not possible to have custom (pure Java) components notified when their native containers are disabled. This flaw should be corrected in a future release.

Focus, please

In order to receive keyboard events, a component has to have keyboard *focus*. The component with the focus is simply the currently selected input component. It receives all keyboard event information until the focus changes. A component can ask for focus with the `Component`'s `requestFocus()` method. Text components like `TextField` and `TextArea` do this automatically whenever you click the mouse in their area. A component can find out when it gains or loses focus through the `FocusListener` interface (see the table of events below). If you want to create your own text-oriented component, you could implement this behavior yourself. For instance, you might request focus when the mouse is clicked in your component's area. After receiving focus, you could change the cursor or do something else to highlight the component.

Many user interfaces are designed so that the focus automatically jumps to the "next available" component when the user presses the TAB key. This behavior is particularly common in forms; users often expect to be able to tab to the next text entry field. AWT handles automatic focus traversal for you when it is applicable. You can get control over the behavior through the `transferFocus()` and `isFocusTraversable()` methods of `Component`. `transferFocus()` passes the focus to the next appropriate component. You can use `transferFocus()` to control the order of tabbing between components by overriding it in the container and implementing your own policy. `isFocusTraversable()` returns a boolean value specifying whether or not the component should be considered eligible for receiving a transfer focus. Your components can override this method to determine whether or not they can be "tabbed to."

Other Component Methods

The `Component` class is very large; it has to provide the base level functionality for all of the various kinds of Java GUI objects. We don't have room to document every method of the component class here, but we'll flesh out our discussion by covering some more of the important ones.

`Container getParent()`

> Return the container that holds this component.

`String getName() / void setName(String name)`

> Get or assign the String name of this component. Giving a component a name is useful for debugging. The name shows up when you do a `toString()`.

`setVisible(boolean visible)`

> Make the component visible or invisible, within its container. If you change the components, visibility, remember to call `validate()` on the container; this causes the layout manager to lay out the container again.

```
Color getForeground() / void setForeground(Color c)
Color getBackground() / void setBackground(Color c)
```

> Get and set the foreground and background colors for this component. The foreground color of any component is the default color used for drawing. For example, it is the color used for text in a text field; it is also the default drawing color for the `Graphics` object passed to the component's `paint()` method. The background color is used to fill the component's area when it is cleared by the default implementation of `update()`.

`Font getFont() / void setFont(Font f)`

Get or set the Font for this component. (Fonts are discussed in Chapter 13, *Drawing With the AWT*.) You can set the Font on text components like TextField and TextArea. For Canvases, Panels, and lightweight components, this will also be the default font used for drawing text on the Graphics context passed to paint().

FontMetrics getFontMetrics(Font font)

Find out the characteristics of a font when rendered on this component. FontMetrics allow you to find out the dimensions of text when rendered in that font.

Dimension getSize() / void setSize(int width, int height)

Get and set the current size of the component. Remember to call validate() on the component's container if you change its size (see Containers below). There are other methods in Component to set its location, but normally this is the job of a layout manager.

Cursor getCursor() / void setCursor(Cursor cursor)

Get or set the type of cursor (mouse pointer) used when the mouse is over this component's area. For example:

```
Component myComponent = ...;
Cursor crossHairs = Cursor.getPredefinedCursor( Cursor.CROSSHAIR_CURSOR );
myComponent.setCursor( crossHairs );
```

Containers

Now that you understand components a little better, our discussion of containers should be easy. A container is a kind of component that holds and manages other AWT components. If you look back to Figure 10.1, you can see the part of the java.awt class hierarchy that descends from Container.

The most useful Container types are Frame, Panel, and Applet. A Frame is a top-level window on your display. Frame is derived from Window, which is pretty much the same, but lacks a border. A Panel is a generic container element used to group components inside of Frames and other Panels. The Applet class is a kind of Panel that provides the foundation for applets that run inside Web browsers. As a Panel, an Applet has the ability to contain other user-interface components. All these classes are subclasses of the Container class; you can also use the Container class directly, like a Panel, to hold components inside of another container. This is called a "lightweight container," and is closely related to lightweight components. We'll discuss lightweight components and containers later in this chapter.

Because a Container is a kind of Component, it has all of the methods of the Component class, including the graphical and event-related methods we're discussing in this chapter. But a container also maintains a list of "child" components, which are the components it manages, and therefore has methods for dealing with those components. By themselves, most components aren't very useful until they are added to a container and displayed. The add() method of the Container class adds a component to the container. Thereafter, this component can be displayed in the container's area and positioned by its layout manager. You can also remove a component from a container with the remove() method.

Layout managers

A *layout manager* is an object that controls the placement and sizing of components within the display area of a container. A layout manager is like a window manager in a display system; it controls where the components go and how big they are. Every container has a default layout manager, but you can easily install a new one by calling the container's `setLayout()` method.

AWT comes with a few layout managers that implement common layout schemes. The default layout manager for a `Panel` is a `FlowLayout`, which tries to place objects at their preferred size from left to right and top to bottom in the container. The default for a `Frame` is a `BorderLayout`, which places a limited number of objects at named locations like "North," "South," and "Center." Another layout manager, the `GridLayout`, arranges components in a rectangular grid. The most general (and difficult to use) layout manager is `GridBagLayout`, which lets you do the kinds of things you can do with HTML tables. We'll get into the details of all of these layout managers in Chapter 12, *Layout Managers*.

As I mentioned above, you normally call `add()` to add a component to a container. There is an overloaded version of `add()` that you may need, depending on what layout manager you're using. Often you'll use the version of `add()` that takes a single `Component` as an argument. However, if you're using a layout manager that uses "constraints," like `BorderLayout` or GridBagLayout, you need to specify additional information about where to put the new component. For that you can use the version that takes a constraint object:

```
add( Component component, Object constraint);
```

For example, to add a component to the top of a `BorderLayout`, you might say:

```
add( newComponent, "North");
```

In this case, the constraint object is the string "North." The `GridBagLayout` uses a much more complex constraint object to specify positioning.

Insets

Insets specify a container's margins; the space specified by the container's insets won't be used by a layout manager. Insets are described by an `Insets` object, which has four `int` fields: `top`, `bottom`, `left`, and `right`. You normally don't need to worry about the insets, the container will set them automatically, taking into account extras like the menu bar that may appear at the top of a frame. However, you should modify the insets if you're doing something like adding a decorative border (for example, a set of "index tabs" at the top of a container) that reduces the space available for components. To change the insets, you override the component's `getInsets()` method, which returns an `Insets` object. For example:

```
//reserve 50 pixels at the top, 5 at the sides and 10 at the bottom
public Insets getInsets() {
    return new Insets (50,5,10,5);
}
```

Z-ordering (stacking components)

In most layout schemes, components are not allowed to overlap, but they can. If they do, the order in which components were added to a container matters. When components overlap they are "stacked" in the order in which they were added: the first component added to the container is on top, the last is on the bottom. To give you more control over stacking, two additional forms of the `add()` method take an additional integer argument that lets you specify the component's exact position in the container's stacking order.

validate( ) and layout( )

A layout manager arranges the components in a container only when asked to. Several things can mess up a container after it's initially laid out:

- Changing its size
- Resizing or moving one of its child components
- Adding, showing, removing, or hiding a child component

Any of these actions causes the container to be marked invalid. Saying that a container is invalid simply means it needs to have its child components readjusted by its layout manager. This is accomplished by calling the `Container`'s `validate()` method. `validate()` then turns around and calls the `Container`'s `doLayout()` method, which asks the layout manager to do its job. In addition, `validate()` also notes that the `Container` has been fixed (i.e., it's valid again) and looks at each child component of the container, recursively validating any containers that are also messed up.

So if you have an applet that contains a small `Panel`--say a keypad holding some buttons--and you change the size of the `Panel` by calling its `resize()` method, you should also call `validate()` on the applet. The applet's layout manager may then reposition or resize the keypad within the applet. It also automatically calls `validate()` for the keypad, so that it can rearrange its buttons to fit inside its new area.

There are two things you should note. First, all components, not just containers, maintain a notion of when they are valid or invalid. But most components (e.g., buttons) don't do anything special when they're validated. If you have a custom component that wants to be notified when it is resized, it might be best to make it a container (perhaps a lightweight container) and do your work in the `doLayout()` method.

Next, child containers are validated only if they are invalid. That means that if you have an invalid component nested inside a valid component and you validate a container above them both, the invalid component may never be reached. However, the `invalidate()` method that marks a container as dirty automatically marks parent containers as well, all the way up the container hierarchy. So that situation should never happen.

Component peers and addNotify()

A component gets its peer when it's added to a container. Containers are associated with display devices through `Toolkit` objects, and thus control the process of peer creation. This means that you can't ask certain questions about a component before it's placed in a container. For example, you can't find out about a component's size or its default font until the component knows where it's being displayed (until it has its peer).

You probably also shouldn't be able to ask a component with no peer about other resources controlled by the peer, such as off screen graphics areas and font metrics. Java's developers apparently thought this restriction too onerous so container-less components are associated with the "default" toolkit that can answer some of these kinds of inquiries. In practice, the default toolkit is usually able to provide the right answer, because with current implementations of Java the default toolkit is probably the only toolkit available. This approach may cause problems in the future, if Java's developers add the ability for different containers to have different toolkits.

The same issue (the existence of a component's peer) also comes up when you are making your own kinds of components and need access to some of these peer resources before you can complete the setup. For example, suppose that you want to set the size or some other feature of your component based on the default font used. You can't complete this setup in your constructor, because the peer doesn't exist yet. The solution to all of these problems is proper use of the `addNotify()` method. As its name implies, `addNotify()` can be used to get notification when the peer is created. You can override it to do your own work, as long as you remember to call `super.addNotify()` to complete the peer creation. For example:

```
class FancyLabel  {
    FancyLabel() {
        // No peer yet...
    }
    public void addNotify() {
        super.addNotify();  // complete the peer creation
        // Complete setup based on Fonts
        // and other peer resources.
    }
}
```

Managing Components

There are a few additional tools of the `Container` class that we should mention.

`Component[] getComponents()`

Returns the container's components in an array.

`void list(PrintWriter out, int indent)`

Generates a list of the components in this container and writes them to the specified `PrintWriter`.

`Component getComponentAt(int x, int y)`

Tells you what component is at the specified coordinates in the container's coordinate system.

Listening for Components

Finally, an important tool to be aware of is the `ContainerListener` interface. It lets you receive an event whenever a component is added to or removed from a container. (It lets you hear the tiny cries of the component as it is imprisoned in its container or torn away.) You can use the `ContainerListener` interface to automate the process of setting up components when they are added to your container. For instance, your container might need to register other kinds of event listeners with its components to track the mouse or handle certain kinds of actions.

Windows and Frames

Windows and frames are the "top level" containers for Java components. A `Window` is simply a plain, graphical screen that displays in your windowing system. Windows have no frills; they are mainly suitable for making "splash" screens and dialogs--things that limit the user's control. `Frame`, on the other hand, is a subclass of `Window` that have a border and can hold a menu-bar. Frames are under the control of your window manager, so you can normally drag a Frame around on the screen and resize it, using the normal controls for your environment. Figure 10.5 shows a `Frame` on the left and a `Window` on the right.

Figure 10.5: A typical frame and window

[Graphic: Figure 10-5]

All other components and containers in Java must be held, at some level, inside of a `Window` or `Frame`. Applets, as we've mentioned a few times, are a kind of `Panel`. Even applets must be housed in a Java frame or window, though normally you don't see an applet's parent frame because it is part of (or simply is) the browser or appletviewer displaying the applet.

A `Frame` is the only `Component` that can be displayed without being added to or attached to another `Container`. After creating a `Frame`, you can call the `show()` method to display it. Let's create a standalone equivalent of our `HelloWeb` applet from Chapter 2, *A First Applet*:

Figure 10.6: Standalone equivalent of the HelloWeb applet

[Graphic: Figure 10-6]

```
class HelloWebApp {
    public static void main( String [] args ) {
        Frame myFrame = new Frame("The Title");
        myFrame.add("Center", new Label("Hello Web!", Label.CENTER) );
        myFrame.pack();
        myFrame.show();
    }
}
```

Here we've got our minimal, graphical, standalone Java application. The `Frame` constructor can take a `String` argument that supplies a title, displayed in the `Frame`'s title bar. (Another approach would be to create the `Frame` with no title, and call `setTitle()` to supply the title later.) After creating the `Frame`, we add our `Label` to it and then call `pack()`, which prepares the `Frame` for display. `pack()` does a couple of things, but its most important effect in this case is that it sets the size of the `Frame` to the smallest needed to hold all of its components. Specifically, `pack()` calls:

```
setSize( preferredSize() );
```

Next, we call `show()` to get the `Frame` onto the screen. The `show()` method returns immediately, without blocking. Fortunately, our application does not exit while the `Frame` is showing. To get rid of a `Frame`, call the `dispose()` method. If you want to hide the `Frame` temporarily, call `setVisible(false)`. You can check to see if a `Frame` is showing with the `isShowing()` method.

In this example, we let `pack()` set the size of the `Frame` for us before we `show()` it. If we hadn't, the `Frame` would have come up at an undefined size. If we instead want the `Frame` to be a specific size (not just hugging its child components) we could simply call `setSize()` instead of `pack()`.

```
...
myFrame.setSize( 300, 300 );
myFrame.show();
```

Other Methods for Controlling Frames

The `setLocation()` method of the `Component` class can be used on a `Frame` or `Window` to set its position on the screen. The x and y coordinates are considered relative to the screen's origin (the top left corner).

You can use the `toFront()` and `toBack()` methods, respectively, to pull a `Frame` or `Window` to the front of other windows, or push it to the background. By default, a user is allowed to resize a `Frame`, but you can prevent resizing by calling `setResizable(false)` before showing the `Frame`.

On most systems, frames can be "iconfied"; that is, they can be represented by a little icon image. You can get and set a frame's icon image by calling `getIconImage()` and `setIconImage()`. Remember that as with all components, you can set the cursor by calling the `setCursor()` method.

Using Windows

Windows and frames have a slightly convoluted relationship. We said above that `Frame` is a subclass of `Window`. However, if you look, you'll see that to create a `Window` you have to have a `Frame` available to serve as its parent. The `Window` constructor takes a `Frame` as an argument.

```
Window myWindow = new Window( myFrame );
```

The rationale for this is long and boring. Suffice it to say that this limitation will probably go away in the future.

Prepacking Windows and Frames

Earlier we said that calling `pack()` on a `Frame` sets the frame's size to the preferred size of its layout. However, the `pack()` method is not simply equivalent to a call to `setSize()`. `pack()` is often called before any of the frame's components have their peers. Therefore, calling `pack()` forces the container to choose its `Toolkit` and to create the peers of any components that have been added to it. After that is done, the layout manager can reliably determine its preferred size.

For a large frame with lots of components, packing the frame is a convenient way to do this setup work in advance, before the frame is displayed. Whether or not this is useful depends on whether you'd rather have your application start up faster, or pop up its frames faster, once it is running.

AWT Performance and Lightweight Components

Java's developers initially decided to implement the standard AWT components with a "mostly native" Toolkit. As we described above, that means that most of the important functionality of these classes is delegated to peer objects, which live in the native operating system. Using native peers allows Java to take on the look and feel of the local operating environment. Macintosh users see Mac applications, PC users see Windows' windows, and Unix users can have their Motif motif; warm fuzzy feelings abound. Java's chameleon-like ability to blend into the native environment is considered by many to be an integral part of platform independance. However, there are a few important downsides to this arrangement.

First, as we mentioned earlier, using native peer implementations makes it much more difficult (if not impossible) to subclass these components to specialize or modify their behavior. Most of their behavior comes from the native peer, and therefore can't be overidden or extended easily. As it turns out, this is not a terrible problem because of the ease with which we can make our own components in Java; we will give you an idea of how to start in Chapter 11, *Using and Creating GUI Components*. It is also true that a sophisticated new component, like an HTML viewer, would benefit little in deriving from a more primitive text viewing component like `TextArea`.

Next, porting the native code makes it much more difficult to bring Java to a new platform. For the user, this can only mean one thing: bugs. Quite simply, while the Java language itself has been quite stable, the cross platform behavior of the AWT has been an Achilles' heel. Although the situation is steadily improving, the lack of large, commercial quality Java applications until relatively recently testifies to the difficulties involved. At this time, new development has been saturated with Java for well over a year (a decade in Net time) and very few real applications are with us.

Finally, we come to a somewhat counterintuitive problem with the use of native peers. In most current implementations of Java, the native peers are quite "heavy" and consume a lot of resources. You might expect that relying on native code would be much more efficient than creating the components in Java. However, it can take a long time to instantiate a large number of GUI elements when each requires the creation of a native peer from the toolkit. And in some cases you may find that once they are created, they don't perform as well as the pure Java equivalents that you can create yourself.

An extreme example would be a spreadsheet that uses an AWT `TextField` for each cell. Creating hundreds of `TextFieldPeer` objects would be something between slow and impossible. While simply saying "don't do that" might be a valid answer, this begs the question: how do you create large applications with complex GUIs? Java would not be a very interesting environment if it was only limited to simple tasks. One solution, taken by development environments like Sun's JavaWorkshop, is to use "wrapper" classes for the standard AWT components; the wrapper controls when peer objects are created. Another attack on the problem has been to create "lightweight" components that are written entirely in Java, and therefore don't require native code.

Using Lightweight Components and Containers

A "lightweight" component is simply a component that is implemented entirely in Java. You implement all of its appearance by drawing in the `paint()` and `update()` methods; you implement its behavior by catching user events (usually at a low level) and possibly generating new events. Lightweight components can be used to create new kinds of gadgets, in the same way you might use a `Canvas` or a `Panel`. But they avoid some of the performance penalties inherent in the use of peers, and, perhaps more importantly, they provide more flexibility in their appearance. A lightweight component can have a transparent background, allowing its container to "show through" its own area. It is also more reasonable to have another component or container overlap or draw into a lightweight component's area.

You create a lightweight component by subclassing the `Component` and `Container` classes directly. That is, instead of writing:

```
class myCanvas extends Canvas { ... }
```

you write:

```
class myCanvas extends Container { ... } // lightweight
```

That's often all you need to do to create a lightweight component. When the lightweight component is put into a container, it doesn't get a native peer. Instead, it gets a `LightweightPeer` that serves as a place holder and identifies the component as lightweight and in need of special help. The container then takes over the responsibilities that would otherwise be handled by a native peer: namely, low-level delivery of events and paint requests. The container receives mouse movement events, key strokes, paint requests, etc., for the lightweight component. It then sorts out the events that fall within the component's bounds and dispatches them to it. Similarly, it translates paint requests that overlap the lightweight component's area and forwards them to it.

Figure 10.7: Sending a paint() request to a component

[Graphic: Figure 10-7]

Figure 10.7 shows a component receiving a `paint()` request via its container. This makes it easy to see how a lightweight component can have a transparent background. The component is actually drawing directly onto its container's graphics context. Conversely, anything that the container drew on its background is visible in the lightweight component. For an ordinary container, this will simply be the container's background color. But you can do much cooler things too. (See the PictureButton example at the end of the next chapter.) All of the normal rules still apply; your lightweight component's `paint()` method should render the component's entire image (assume that the container has obliterated it), but your `update()` method can assume that whatever drawing it has done previously is still intact.

Just as you can create a lightweight component by subclassing `Component`, you can create a lightweight container by subclassing `Container`. A lightweight container can hold any components, including other lightweight components and containers. In this case, event handling and paint requests are managed by the first "regular" container in the container hierarchy. (There has to be one somewhere, if you think about it.) This brings us to the cardinal rule of subclassing containers, which is:

"Always call `super.paint()` if you override a container's `paint()` method."

If you don't, the container won't be able to manage lightweight components properly.

To summarize, lightweight components are very flexible, pure Java components that are managed by their container and have a transparent background by default. Lightweight components do not rely on native peers from the AWT toolkit to provide their implementations and so they can not readily take on the look and feel of the local platform. In a sense, a lightweight component is just a convenient way to package an extension to a container's painting and event handling methods. But, again, all of this happens automatically, behind the scenes; you can create and use lightweight components as you would any other kind of component.

We'll see examples of lightweight components and containers in the next chapter.

PREVIOUS
Writing a Protocol Handler

HOME
BOOK INDEX

NEXT
Applets

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

11. Using and Creating GUI Components

Contents:

The last chapter discussed a lot of concepts: how Java's user interface facility is put together, and how the larger pieces work. You should understand what components and containers are, how you use them to create a display, what events are, how components use them to communicate with the rest of your application, and what layout managers are. In other words, we've covered a lot of material that's "good for you."

Now that we're through with the general concepts and background, we'll get to the fun stuff: how to do things with AWT. We will cover all the components that the AWT package supplies, how to use these components in applets and applications, and how to build your own components. We will have lots of code and lots of pretty examples to look at. Here's a list of the components we cover:

- `Button` and `Label`
- Text components: `TextArea` and `TextField`
- `List`
- `Menu` and `Choice`

- PopupMenu
- Checkbox and checkbox grouping
- Scrollbar and ScrollPane
- Dialog and FileDialog

When we discuss how to create your own components, we'll cover these issues:

- Extending canvas
- Lightweight components
- Making components that fire events

11.1 Buttons and Labels

We'll start with the simplest components, buttons, and labels. Frankly, there isn't much to say about them. If you've seen one button, you've seen them all; and you've already seen buttons in the applets of Chapter 2 (HelloWeb3 and HelloWeb4). A button generates an ActionEvent when the user presses it. To receive these events, your program registers an ActionListener, which must implement the actionPerformed() method. The argument passed to actionPerformed() is the event itself. Rather than rehash this material, I'll refer you to Chapter 2.

So much for review. There's one more thing worth saying about buttons, which applies to any component that generates an action event. Java lets us specify an "action command" for buttons (and other components, like menu items, that can generate action events). The action command is less interesting than it sounds. It is just a String that serves to identify the component that sent the event. By default, the action command of a Button is the same as the text of its label; it is included in action events, so you can use it to figure out which button an event came from. To get the action command from an action event, call the event's getActionCommand() method. The following code checks whether the user pressed the "Yes" button:

```
public void actionPerformed(ActionEvent e){
    if (e.getActionCommand().equals("Yes") {
        //the user pressed "Yes"; do something
        ...
    }
}
```

You can change the action command by calling the button's `setActionCommand()` method. The code below changes the button `b`'s action command to "confirm":

```
myButton.setActionCommand("confirm");
```

It's a good idea to get used to setting action commands explicitly, because they prevent your code from breaking when you or some other developer "internationalizes" it. If you rely on the button's label, your code will stop working as soon as that label changes; a French user might see the label "Oui" rather than "Yes." By setting the action command, you eliminate one source of bugs; for example, the button `myButton` above will always generate the action command "confirm," regardless of what its label says.

There's even less to be said about Label components. They're just text strings, housed in a component. There aren't any special events associated with labels; about all you can do is specify the text's alignment, which controls the position of the text within the area that the label occupies when displayed. The following code creates some labels with different alignments:

```
Label l1 = new Label("Lions"); //label with default alignment (CENTER)
Label l2 = new Label("Tigers", LEFT); //left aligned label
Label l3 = new Label (); //label with no text, default alignment
l3.setText("and Bears"); //assigning text to l3
l3.setAlignment(RIGHT); //setting l3's alignment
```

Now we've built three labels, using all three constructors and several of the class's methods. To display the labels, you only have to add them to a container by calling the container's `add()` method.

The other characteristics you might like to set on labels, like changing their font or color, are accomplished using the methods of the Component class Chapter 10, *Understand the Abstract Windowing Toolkit* For example, you can call `setFont()` and `setColor()` on a label, as with any other component.

Given that labels are so simple, why do we need them at all? Why not just call `drawString()` whenever we need text? Remember that a `Label` is a `Component`. That's important; it means that labels have the normal complement of methods for setting fonts and colors that we mentioned above, as well as the ability to be managed sensibly by a layout manager. Therefore, they're much more flexible.

Speaking of layouts--if you use the setText() method to change the text of your label, its size will probably change. So you should remember to call validate() on its container, to lay things out again.[1]

[1] At least as of Java 1.1, labels aren't very smart. Simply validating the container isn't enough. I had to explicitly invalidate the label first.

```
label.setText(...);
label.invalidate();
validate();  // on the container holding the label
```

This ought to be considered a bug.

PREVIOUS	HOME	NEXT
Applets	BOOK INDEX	Text Components

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

12. Layout Managers

Contents:

A layout manager arranges the child components of a container, as shown in Figure 12.1. It positions and sets the size of components within the container's display area according to a particular layout scheme. The layout manager's job is to fit the components into the available area, while maintaining the proper spatial relationships between the components. AWT comes with a few standard layout managers that will collectively handle most situations; you can make your own layout managers if you have special requirements.

Figure 12.1: LayoutManager at work

[Graphic: Figure 12-1]

Every container has a default layout manager; therefore, when you make a new container, it comes with a `LayoutManager` object of the appropriate type. You can install a new layout manager at any time with the `setLayout()` method. Below, we set the layout manager of a container to a `BorderLayout`:

```
setLayout ( new BorderLayout() );
```

Notice that we call the `BorderLayout` constructor, but we don't even save a reference to the layout manager. This is typical; once you have installed a layout manager, it does its work behind the scenes, interacting with the container. You rarely call the layout manager's methods directly, so you don't usually need a reference (a notable exception for `CardLayout`). However, you do need to know what the layout manager is going to do with your components as you work with them.

As I explained earlier, the `LayoutManager` is consulted whenever a container's `doLayout()` method is called (usually when it is validated), to reorganize the contents. It does its job by calling the `setLocation()` and `setBounds()` methods of the individual child components to arrange them in the container's display area. A container is layed out the first time it is displayed, and thereafter whenever the container's `validate()` method is called. Containers that are a subclass of the `Window` class (which include `Frame`) are automatically validated whenever they are packed or resized. Calling `pack()` sets the window's size so it is as small as possible while holding all its components at their preferred sizes.

Every component determines three important pieces of information used by the layout manager in placing and sizing it: a minimum size, a maximum size, and a preferred size. These are reported by the `getMinimumSize()`, `getMaximumSize()`, and `getPreferredSize()`, methods of `Component`, respectively. For example, a plain `Button` object can normally be sized to any proportions. However, the button's designer can provide a preferred size for a good-looking button. The layout manager might use this size when there are no other constraints, or it might ignore it, depending on its scheme. Now, if we give the button a label, the button may need a minimum size in order to display itself properly. The layout manager might show more respect for the button's minimum size and guarantee that it has at least that much space. Similarly, a particular component might not be able to display itself properly if it is too large (perhaps it has to scale up an image); it can use `getMaximumSize()` to report the largest size it considers acceptable.[1]

> [1] Unfortunately, the current set of layout managers doesn't do anything with the maximum size. This may change in a future release.

The preferred size of a `Container` object has the same meaning as for any other type of component. However, since a `Container` may hold its own components and want to arrange them in its own layout, its preferred size is a function of its layout manager. The layout manager is therefore involved in both sides of the issue. It asks the components in its container for their preferred (or minimum) sizes in order to arrange them. Based on those values it also calculates the preferred size that is reported by its own container to that container's parent.

When a layout manager is called to arrange its components, it is working within a fixed area. It usually begins by looking at its container's dimensions, and the preferred or minimum sizes of the child components. It then doles out screen area and sets the sizes of components according to its scheme. You can override the `getMinimumSize()`, `getMaximumSize()`, and `getPreferredSize()` methods of a component, but you should do this only if you are actually specializing the component, and it has new needs. If you find yourself fighting with a layout manager because it's changing the size of one of your components, you are probably using the wrong kind of layout manager or not composing your user interface properly. Remember that it's possible to use a number of `Panel` objects in a given display, where each one has its own `LayoutManager`. Try breaking down the problem: place related components in their own `Panel` and then arrange the panels in the container. When that becomes too complicated, you can choose to use a constraint based layout manager like `GridBagLayout`, which we'll discuss later in this chapter.

12.1 FlowLayout

`FlowLayout` is a simple layout manager that tries to arrange components with their preferred sizes, from left to right and top to bottom in the display. A `FlowLayout` can have a specified justification of `LEFT`, `CENTER`, or `RIGHT`, and a fixed horizontal and vertical padding. By default, a flow layout uses `CENTER` justification, meaning that all components are centered within the area allotted to them. `FlowLayout` is the default for `Panel` components like `Applet`.

The following applet adds five buttons to the default `FlowLayout`; the result is shown in Figure 12.2.

Figure 12.2: A flow layout

[Graphic: Figure 12-2]

```
import java.awt.*;

public class Flow extends java.applet.Applet {
    public void init() {
        // Default for Applet is FlowLayout
        add( new Button("One") );
        add( new Button("Two") );
        add( new Button("Three") );
        add( new Button("Four") );
        add( new Button("Five") );
    }
}
```

If the applet is small enough, some of the buttons spill over to a second or third row.

PREVIOUS | HOME | NEXT
Creating custom components | BOOK INDEX | GridLayout

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

PREVIOUS Chapter 13 NEXT

13. Drawing With the AWT

Contents:

If you've read the last few chapters and seen the examples in the tutorial in Chapter 2, *A First Applet*, then you've probably picked up the basics of how graphical operations are performed in Java. Up to this point, we have done some simple drawing and even displayed an image or two. In this chapter, we will finally give graphics programming its due and go into depth about drawing techniques and the tools for working with images in Java. In the next chapter, we'll explore image processing tools in more detail and we'll look at the classes that let you generate images pixel by pixel on the fly.

13.1 Basic Drawing

The classes you'll use for drawing come from the `java.awt` package, as shown in Figure 13.1.[1].

> [1] The current set of drawing tools has many limitations. In the near future, JavaSoft will be releasing packages for advanced 2D graphics, which will have much greater capabilities. A 3D package is also planned. See Chapter 1, *Yet Another Language?* for information about likely future Java enhancements.

Figure 13.1: Graphics classes of the java.awt package

[Graphic: Figure 13-1]

An instance of the `java.awt.Graphics` class is called a *graphics context*. It represents a drawable surface such as a component's display area or an off-screen image buffer. A graphics context provides methods for performing all basic drawing operations on its area, including the painting of image data. We call the `Graphics` object a graphics context because it also holds contextual information about the drawing area. This information includes parameters like the drawing area's clipping region, painting color, transfer mode, and text font. If you consider the drawing area to be a painter's canvas, you might think of a graphics context as an easel that holds a set of tools and marks off the work area.

There are four ways you normally acquire a `Graphics` object. Roughly, from most common to least, they are:

- From AWT, as the result of a painting request. In this case, AWT acquires a new graphics context for the appropriate area and passes it to your component's `paint()` or `update()` method.

- Directly from an off-screen image buffer. In this case, we ask the image buffer for a graphics context directly. We'll use this when we discuss techniques like double buffering.

- By copying an existing `Graphics` object. Duplicating a graphics object can be useful for more elaborate drawing operations; different copies of a `Graphics` object draw into the same area on the screen, but can have different attributes and clipping regions.

- Directly from an on-screen component. It's possible to ask a component to give you a `Graphics` object for its display area. However, this is almost always a mistake; if you feel tempted to do this, think about why you're trying to circumvent the normal `paint()`/`repaint()` mechanism.

Each time a component's `update()` or `paint()` method is called, AWT provides the component with a new `Graphics` object for drawing in the display area. This means that attributes we set during one painting session, such as drawing color or clipping region, are reset the next time `paint()` or `update()` is called. (Each call to `paint()` starts with a tidy new easel.) For the most common attributes, like foreground color, background color, and font, we can set defaults in the component itself. Thereafter, the graphics contexts for painting in that component come with those properties initialized appropriately.

If we are working in a component's `update()` method, we can assume our on-screen artwork is still intact, and we need only to make whatever changes are needed to bring the display up to date. One way to optimize drawing operations in this case is by setting a clipping region, as we'll see shortly. If our `paint()` method is called, however, we have to assume the worst and redraw the entire display.

Drawing Methods

Methods of the `Graphics` class operate in a standard coordinate system. The origin of a newly created graphics context is the top left pixel of the component's drawing area, as shown in Figure 13.2.

Figure 13.2: Graphics coordinate system

[Graphic: Figure 13-2]

The diagram above illustrates the default coordinate system. The point (0,0) is at the top left corner of the drawing area; the point (width, height) is just outside the drawing area at the bottom right corner. The point at the bottom right corner within the drawing area has coordinates (width-1, height-1). This gives you a drawing area that is `width` pixels wide and `height` pixels high.

The coordinate system can be translated (shifted) with the `translate()` method to specify a new point as the origin. The drawable area of the graphics context can be limited to a region with the `setClip()` method.

The basic drawing and painting methods should seem familiar to you if you've done any graphics programming. The following applet, `TestPattern`, exercises most of the simple shape drawing commands; it's shown in Figure 13.3.

Figure 13.3: The TestPattern applet

[Graphic: Figure 13-3]

```
import java.awt.*;
import java.awt.event.*;
public class TestPattern extends java.applet.Applet {
    int theta = 45;
    public void paint( Graphics g ) {
        int Width = size().width;
        int Height = size().height;
        int width = Width/2;
        int height = Height/2;
        int x = (Width - width)/2;
        int y = (Height- height)/2;
        int [] polyx =  { 0, Width/2, Width, Width/2 };
        int [] polyy =  { Height/2, 0, Height/2, Height };
        Polygon poly = new Polygon( polyx, polyy, 4 );

        g.setColor( Color.black );
        g.fillRect( 0, 0, size().width, size().height );
        g.setColor( Color.yellow );
        g.fillPolygon( poly );
        g.setColor( Color.red );
        g.fillRect( x, y, width, height );
        g.setColor( Color.green );
        g.fillOval( x, y, width, height );
        g.setColor( Color.blue );
        int delta = 90;
        g.fillArc( x, y, width, height, theta, delta );
        g.setColor( Color.white );
        g.drawLine( x, y, x+width, x+height );
    }
    public void init() {
        addMouseListener( new MouseAdapter() {
            public void mousePressed( MouseEvent e ) {
                theta = (theta + 10) % 360;
                repaint();
            }
        } );
    }
}
```

`TestPattern` draws a number of simple shapes and responds to mouse clicks by rotating the filled arc and repainting. Compile it and give it a try. If you click repeatedly on the applet, you may notice that everything flashes when it repaints. `TestPattern` is not very intelligent about redrawing; we'll examine some better techniques in the upcoming section on drawing techniques.

With the exception of `fillArc()` and `fillPolygon()`, each method takes a simple `x`, `y` coordinate for the top left corner of the shape and a `width` and `height` for its size. We have picked values that draw the shapes centered, at half the width and height of the applet.

The most interesting shape we've have drawn is the `Polygon`, a yellow diamond. A `Polygon` object is specified by two arrays that contain the `x` and `y` coordinates of each vertex. In our example, the coordinates of the points in the polygon are (`polyx[0]`, `polyy[0]`), (`polyx[1]`, `polyy[1]`), and so on. There are simple drawing methods in the `Graphics` class that take two arrays and draw or fill the polygon, but we chose to create a `Polygon` object and draw it instead. The reason is that the `Polygon` object has some useful utility methods that we might want to use later. A `Polygon` can, for instance, give you its bounding box and tell you if a given point lies within its area.

AWT also provides a `Shape` interface, which is implemented by different kinds of two dimensional objects. Currently, it is only implemented by the `Rectangle` and `Polygon` classes, but it may be a sign of things to come, particularly when JavaSoft releases the extended 2D drawing package. The `setClip()` method of the `Graphics` class takes a `Shape` as an argument, but for the time being, it only works if that `Shape` is a `Rectangle`.

The `fillArc()` method requires six integer arguments. The first four specify the bounding box for an oval--just like the `fillOval()` method. The final two arguments specify what portion of the oval we want to draw, as a starting angle and an offset. Both the starting angle and the offset are specified in degrees. Zero degrees is at three o'clock; a positive angle is clockwise. For example, to draw the right half of a circle, you might call:

```
g.fillArc(0, 0, radius * 2, radius * 2, -90, 180);
```

See the Dial example in Chapter 11, *Using and Creating GUI Components* (widgets?) for an example of some trigonometric gymnastics with arcs().

Table 13.1 shows the shape-drawing methods of the `Graphics` class. As you can see, for each of the `fill()` methods in the example, there is a corresponding `draw()` method that renders the shape as an unfilled line drawing.

Table 13.1: Shape Drawing Methods in the Graphics Class

Method	**Description**
`draw3DRect()`	Draws a highlighted, 3D rectangle
`drawArc()`	Draws an arc
`drawLine()`	Draws a line
`drawOval()`	Draws an oval
`drawPolygon()`	Draws a polygon, connecting endpoints
`drawPolyline()`	Draws a line connecting a polygon's points
`drawRect()`	Draws a rectangle
`drawRoundRect()`	Draws a rounded-corner rectangle
`fill3DRect()`	Draws a filled, highlighted, 3D rectangle
`fillArc()`	Draws a filled arc
`fillOval()`	Draws a filled oval
`fillPolygon()`	Draws a filled polygon
`fillRect()`	Draws a filled rectangle

`fillRoundRect()`	Draws a filled, rounded-corner rectangle

`draw3Drect()` automatically chooses colors by "darkening" the current color. So you should set the color to something other than black, which is the default (maybe gray or white); if you don't, you'll just get black on both sides. For an example, see the `PictureButton` in Chapter 11, *Using and Creating GUI Components*.

There are a few important drawing methods missing from Table 13.1. For example, the `drawString()` method, which draws text, and the `drawImage()` method, which draws an image. We'll discuss these methods in detail in later sections.

PREVIOUS
Absolute Positioning?

HOME
BOOK INDEX

NEXT
Colors

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

14. Working With Images

Contents:

14.1 Image Processing

Up to this point, we've confined ourselves to working with the high-level drawing commands of the `Graphics` class and using images in a hands-off mode. In this section, we'll clear up some of the mystery surrounding images and see how they are produced and used. The classes in the `java.awt.image` package handle image processing; Figure 14.1 shows the classes in this package.

Figure 14.1: The java.awt.image package

[Graphic: Figure 14-1]

First, we'll return to our discussion about image observers and see how we can get more control over image data as it's processed asynchronously by AWT components. Then we'll open the hood and have a look at image production. If you're interested in creating sophisticated graphics, such as rendered images or video streams, this will teach you about the foundations of image construction in Java.[1]

Objects that work with image data fall into one of three categories: image-data producers, image-data consumers, and image-status observers. Image producers implement the `ImageProducer` interface. They create pixel data and distribute it to one or more consumers. Image consumers implement a corresponding `ImageConsumer` interface. They eat the pixel data and do something useful with it, such as display it on screen or analyze its contents. Image observers, as I mentioned earlier, implement the `ImageObserver` interface. They are effectively nosy neighbors of image consumers that watch as the image data arrives.

Image producers generate the information that defines each pixel of an image. A pixel has both a color and a transparency; the transparency specifies how pixels underneath the image show through. Image producers maintain a list of registered consumers for the image and send them this pixel data in one or more passes, as the pixels are generated. Image producers give the consumers other kinds of information as well, such as the image's dimensions. The producer also notifies the consumer when it has reached a boundary of the image. For a static image, such as GIF or JPEG data, the producer signals when the entire image is complete, and production is finished. For a video source or animation, the image producer could generate a continuous stream of pixel data and mark the end of each frame.

An image producer delivers pixel data and other image-attribute information by invoking methods in its consumers, as shown in Figure 14.2. This diagram illustrates an image producer sending pixel data to three consumers by invoking their `setPixels()` methods.

Figure 14.2: Image observers, producers, and consumers

[Graphic: Figure 14-2]

Each consumer represents a view of the image. A given consumer might prepare the image for display on a particular medium, or it might simply serve as a filter and pass the image data to another consumer down the line.

Figure 14.2 also shows an image observer, watching the status of one of the consumers. The observer is notified as new portions of the image and new attributes are ready. Its job is to track this information and let another part of the application know its status. As I discussed earlier, the image observer is essentially a callback that is notified asynchronously as the image is built. The default `Component` class image observer that we used in our previous examples called `repaint()` for us each time a new section of the image was available, so that the screen was updated more or less continuously as the data arrived. A different kind of image observer might wait for the entire image before telling the application to display it; yet another observer might update a loading meter showing how far the image loading had progressed.

[1] You will also want to pay attention to the forthcoming Java Media API. Java Media will support plug-n-play streaming media.

Implementing an ImageObserver

To be an image observer, you have to implement the single method, `imageUpdate()`, defined by the `java.awt.image.ImageObserver` interface:

```
public boolean imageUpdate(Image image, int flags, int x, int y,
                           int width, int height)
```

`imageUpdate()` is called by the consumer, as needed, to pass the observer information about the construction of its view of the image. Essentially, any time the image changes, the consumer tells the observer so that the observer can perform any necessary actions, like repainting. `image` holds a reference to the `Image` object the consumer is processing. `flags` is an integer whose bits specify what information about the image is now available. The values of the flags are defined as `static` identifiers in the `ImageObserver` interface, as shown in Table 14.1.

Table 14.1: ImageObserver Information Flags

Flag	Description
`HEIGHT`	The height of the image is ready
`WIDTH`	The width of the image is ready
`FRAMEBITS`	A frame is complete
`SOMEBITS`	An arbitrary number of pixels have arrived
`ALLBITS`	The image is complete
`ABORT`	The image loading has been aborted
`ERROR`	An error occurred during image processing; attempts to display the image will fail

The flags determine which of the other parameters, `x`, `y`, `width`, and `height`, hold valid data and what they mean. To test whether a particular flag in the `flags` integer is set, we have to resort to some binary shenanigans. The following class, `MyObserver`, implements the `ImageObserver` interface and prints its information as it's called:

```
import java.awt.*;
import java.awt.image.*;

class MyObserver implements ImageObserver {

    public boolean imageUpdate( Image image, int flags, int x, int y,
                                int width, int height) {

        if ( (flags & HEIGHT) !=0 )
            System.out.println("Image height = " + height );

        if ( (flags & WIDTH ) !=0 )
            System.out.println("Image width = " + width );

        if ( (flags & FRAMEBITS) != 0 )
            System.out.println("Another frame finished.");

        if ( (flags & SOMEBITS) != 0 )
            System.out.println("Image section :"
                         + new Rectangle( x, y, width, height ) );
        if ( (flags & ALLBITS) != 0 ) {
            System.out.println("Image finished!");
            return false;
        }

        if ( (flags & ABORT) != 0 )  {
            System.out.println("Image load aborted...");
            return false;
        }

        return true;
    }
}
```

The `imageUpdate()` method of `MyObserver` is called by the consumer periodically, and prints simple status messages about the construction of the image. Notice that `width` and `height` play a dual role. If `SOMEBITS` is set, they represent the size of the chunk of the image that has just been delivered. If `HEIGHT` or `WIDTH` is set, however, they represent the overall image dimensions. Just for amusement, we have used the `java.awt.Rectangle` class to help us print the bounds of rectangular region.

`imageUpdate()` returns a `boolean` value indicating whether or not it's interested in future updates. If the image is finished or aborted, `imageUpdate()` returns `false` to indicate it isn't interested in further updates. Otherwise, it returns `true`.

The following example uses `MyObserver` to print information about an image as AWT loads it:

```
import java.awt.*;

public class ObserveImage extends java.applet.Applet {
    Image img;
    public void init() {
        img = getImage( getClass().getResource(getParameter("img")) );
        MyObserver mo = new MyObserver();
        img.getWidth( mo );
        img.getHeight( mo );
        prepareImage( img, mo );
    }
}
```

After requesting the `Image` object with `getImage()`, we perform three operations on it to kick-start the loading process. `getWidth()` and `getHeight()` ask for the image's width and height. If the image hasn't been loaded yet, or its size can't be determined until loading is finished, our observer will be called when the data is ready. `prepareImage()` asks that the image be readied for display on the component. It's a general mechanism for getting AWT started loading, converting, and possibly scaling the image. If the image hasn't been otherwise prepared or displayed, this happens asynchronously, and our image observer will be notified as the data is constructed.

You may be wondering where the image consumer is, since we never see a call to `imageUpdate()`. That's a good question, but for now I'd like you to take it on faith that the consumer exists. As you'll see later, image consumers are rather mysterious objects that tend to hide beneath the surface of image-processing applications. In this case, the consumer is hiding deep inside the implementation of `Applet`.

You should be able to see how we could implement all sorts of sophisticated image loading and tracking schemes. The two most obvious strategies, however, are to draw an image progressively, as it's constructed, or to wait until it's complete and draw it in its entirety. We have already seen that the `Component` class implements the first scheme. Another class, `java.awt.MediaTracker`, is a general utility that tracks the loading of a number of images or other media types for us. We'll look at it next.

Using a MediaTracker

`java.awt.MediaTracker` is a utility class that simplifies life if we have to wait for one or more images to be loaded before they're displayed. A `MediaTracker` monitors the preparation of an image or a group of images and lets us check on them periodically, or wait until they are completed. `MediaTracker` uses the `ImageObserver` interface internally to receive image updates.

The following applet, `LoadMe`, uses a `MediaTracker` to wait while an image is prepared. It shows a "Loading..." message while it's waiting. (If you are retrieving the image from a local disk or very fast network, this might go by quickly, so pay attention.)

```
import java.awt.*;
public class TrackImage extends java.applet.Applet implements Runnable {
    Image img;
    final int MAIN_IMAGE = 0;
    MediaTracker tracker;
    boolean show = false;
    Thread runme;
    String message = "Loading...";
    public void init() {
        img = getImage( getClass().getResource(getParameter("img")) );
        tracker = new MediaTracker(this);
        tracker.addImage( img, MAIN_IMAGE );
    }
    public void start() {
        if ( !tracker.checkID( MAIN_IMAGE ) ) {
            runme = new Thread( this );
            runme.start();
        }
    }
    public void stop() {
        runme.stop();
        runme = null;
    }
    public void run() {
        repaint();
        try {
            tracker.waitForID( MAIN_IMAGE );
        } catch( InterruptedException e) { }
        if ( tracker.isErrorID( MAIN_IMAGE ) )
            message= "Error";
        else
            show = true;
        repaint();
    }
    public void paint( Graphics g ) {
        if ( show )
            g.drawImage( img, 0, 0, this );
        else {
            g.drawRect( 0, 0, getSize().width-1, getSize().height-1);
            g.drawString( message, 20, 20 );
        }
    }
}
```

From its `init()` method, `LoadMe` requests its image and creates a `MediaTracker` to manage it. Later, after the applet is started, `LoadMe` fires up a thread to wait while the image is loaded. Note that we don't do this in `init()` because it would be rude to do anything time-consuming there; it would take up time in an AWT thread that we don't own. In this case, waiting in `init()` would be especially bad because `paint()` would never get called and our "loading" message wouldn't be displayed; the applet would just hang until the image loaded. It's often better to create a new thread for initialization and display a startup message in the interim.

When we construct a `MediaTracker`, we give it a reference to our component (`this`). After creating a `MediaTracker`, we assign it images to manage. Each image is associated with an integer identifier we'll use later for checking on its status. Multiple images can be associated with the same identifier, letting us manage them as a group. The value of the identifier is also used to prioritize loading when waiting on multiple sets of images; lower IDs have higher priority. In this case, we want to manage only a single image, so we created one identifier called `MAIN_IMAGE` and passed it as the ID for our image in the call to `addImage()`.

In our applet's `start()` method, we call the `MediaTracker`'s `checkID()` routine with the ID of the image to see if it's already been loaded. If it hasn't, the applet fires up a new thread to fetch it. The thread executes the `run()` method, which simply calls the `MediaTracker` `waitforID()` routine and blocks on the image, waiting for it to finish loading. The `show` flag tells `paint()` whether to display our status message or the actual image. We do a `repaint()` immediately upon entering `run()` to display the "Loading..." status, and again upon exiting to change the display. We test for errors during image preparation with `isErrorID()` and change the status message if we find one.

This may seem like a lot of work to go through, just to put up a status message while loading a single image. `MediaTracker` is more valuable when we are working with many images that have to be available before we can begin parts of our application. It saves us from implementing a custom `ImageObserver` for every application. In the future, `MediaTracker` should also be able to track the status of audio clips and other kinds of media (as its name suggests).

Producing Image Data

What if we want to make our own image data? To be an image producer, we have to implement the five methods defined in the `ImageProducer` interface:

- `addConsumer()`
- `startProduction()`
- `isConsumer()`
- `removeConsumer()`
- `requestTopDownLeftRightResend()`

Four methods of `ImageProducer` simply deal with the process of registering consumers. `addConsumer()` takes an `ImageConsumer` as an argument and adds it to the list of consumers. Our producer can then start sending image data to the consumer whenever it's ready. `startProduction()` is identical to `addConsumer()`, except that it asks the producer to start sending data as soon as possible. The difference might be that a given producer would send the current frame of data or initiate construction of a frame immediately, rather than waiting until its next cycle. `isConsumer()` tests whether a particular consumer is already registered, and `removeConsumer()` removes a consumer from the list. We'll see shortly that we can perform these kinds of operations easily with a `Vector`; see Chapter 7, *Basic Utility Classes* for a complete discussion of `Vector` objects.

An `ImageProducer` also needs to know how to use the `ImageConsumer` interface of its clients. The final method of the `ImageProducer` interface, `requestTopDownLeftRightResend()`, asks that the image data be resent to the consumer, in order, from beginning to end. In general, a producer can generate pixel data and send it to the consumer in any order that it likes. The `setPixels()` method of the `ImageConsumer` interface takes parameters telling the consumer what part of the image is being delivered on each call. A call to `requestTopDownLeftRightResend()` asks the producer to send the pixel data again, in order. A consumer might do this so that it can use a higher quality conversion algorithm that relies on receiving the pixel data in sequence. It's important to note that the producer is allowed to ignore this request; it doesn't have to be able to send the data in sequence.

Color models

Everybody wants to work with color in their application, but using color raises problems. The most important problem is simply how to represent a color. There are many different ways to encode color information: red, green, blue (RGB) values; hue, saturation, value (HSV); hue, lightness, saturation (HLS); and more. In addition, you can provide full color information for each pixel, or you can just specify an index into a color table (palette) for each pixel. The way you represent a color is called a *color model*. AWT provides tools for two broad groups of color models: *direct* and *indexed*.

As you might expect, you need to specify a color model in order to generate pixel data; the `abstract` class `java.awt.image.ColorModel` represents a color model. A `ColorModel` is one of the arguments to the `setPixels()` method an image producer calls to deliver pixels to a consumer. What you probably wouldn't expect is that you can use a different color model every time you call `setPixels()`. Exactly why you'd do this is another matter. Most of the time, you'll want to work with a single color model; that model will probably be the default direct color model. But the additional flexibility is there if you need it.

By default, the core AWT components use a direct color model called ARGB. The A stands for "alpha,", which is the historical name for transparency. RGB refers to the red, green, and blue color components that are combined to produce a single, composite color. In the default ARGB model, each pixel is represented by a 32-bit integer that is interpreted as four 8-bit fields; in order, the fields represent the transparency (A), red, green, and blue components of the color, as shown in Figure 14.3.

Figure 14.3: ARGB color encoding

[Graphic: Figure 14-3]

To create an instance of the default ARGB model, call the `static getRGBdefault()` method in `ColorModel`. This method returns a `DirectColorModel` object; `DirectColorModel` is a subclass of `ColorModel`. You can also create other direct color models by calling a `DirectColorModel` constructor, but you shouldn't need to unless you have a fairly exotic application.

In an indexed color model, each pixel is represented by a smaller amount of information: an index into a table of real color values. For some applications, generating data with an indexed model may be more convenient. If you have an 8-bit display or smaller, using an indexed model may be more efficient, since your hardware is internally using an indexed color model of some form.

While AWT provides `IndexedColorModel` objects, we won't cover them in this book. It's sufficient to work with the `DirectColorModel`. Even if you have an 8-bit display, the Java implementation on your platform should accommodate the hardware you have and, if necessary, dither colors to fit your display. Java also produces transparency on systems that don't natively support it by dithering colors.

Creating an image

Let's take a look at producing some image data. A picture may be worth a thousand words, but fortunately, we can generate a picture in significantly fewer than a thousand words of Java. If we're interested in producing a static image with just one frame, we can use a utility class that acts as an `ImageProducer` for us.

`java.awt.image.MemoryImageSource` is a simple utility class that implements the `ImageProducer` interface; we give it pixel data in an array and it sends that data to an image consumer. A `MemoryImageSource` can be constructed for a given color model, with various options to specify the type and positioning of its data. We'll use the simplest form, which assumes an ARGB color model.

The following applet, `ColorPan`, creates an image from an array of integers holding ARGB pixel values:

```
import java.awt.*;
import java.awt.image.*;
public class ColorPan extends java.applet.Applet {
    Image img;
    int width, height;
    int [] pixData;
    public void init() {
        width = getSize().width;
        height = getSize().height;
        pixData = new int [width * height];
        int i=0;
        for (int y = 0; y < height; y++) {
            int red = (y * 255) / (height - 1);
            for (int x = 0; x < width; x++) {
                int green = (x * 255) / (width - 1);
                int blue = 128;
                int alpha = 255;
                pixData[i++] = (alpha << 24) | (red << 16) |
                               (green << 8 ) | blue;
            }
        }
    }
    public void paint( Graphics g ) {
        if ( img == null )
            img = createImage( new MemoryImageSource(width, height,
                                          pixData, 0, width));
        g.drawImage( img, 0, 0, this );
    }
}
```

Give it a try. The size of the image is determined by the size of the applet when it starts up. What you should get is a very colorful box that pans from deep blue at the upper left-hand corner to bright yellow at the bottom right with green and red at the other extremes. (You'll have to imagine something more colorful than the black and white image above.)

We create the pixel data for our image in the `init()`method, and then use `MemoryImageSource` to create and display the image in `paint()`. The variable `pixData` is a one-dimensional array of integers that holds 32-bit ARGB pixel values. In `init()` we loop over every pixel in the image and assign it an ARGB value. The alpha (transparency) component is always 255, which means the image is opaque. The blue component is always 128, half its maximum intensity. The red component varies from 0 to 255 along the y axis; likewise, the green component varies from 0 to 255 along the x axis. The line below combines these components into an ARGB value:

```
pixData[i++] = (alpha << 24) | (red << 16) | (green << 8 ) | blue;
```

The bitwise left-shift operator (`<<`) should be familiar to C programmers. It simply shoves the bits over by the specified number of positions. The alpha value takes the top byte of the integer, followed by the red, green, and blue values.

When we construct the `MemoryImageSource` as a producer for this data, we give it five parameters: the width and height of the image to construct (in pixels), the `pixData` array, an offset into that array, and the width of each scan line (in pixels). We'll start with the first element (offset 0) of `pixData`; the width of each scan line and the width of the image are the same. The array `pixData` has `width * height` elements, which means it has one element for each pixel.

We create the actual image once, in `paint()`, using the `createImage()` method that our applet inherits from `Component`. In the double-buffering and off-screen drawing examples, we used `createImage()` to give us an empty off-screen image buffer. Here we use `createImage()` to generate an image from a specified `ImageProducer`. `createImage()` creates the `Image` object and receives pixel data from the producer to construct the image. Note that there's nothing particularly special about `MemoryImageSource`; we could use any object that implements the image-producer interface inside of `createImage()`, including one we wrote ourselves. Once we have the image, we can draw it on the display with the familiar `drawImage()` method.

You can use `MemoryImageSource` to produce complex, pixel-by-pixel graphics or render images from arbitrary sources. It produces static images, though; when it reaches the end of its pixel data, its job is done. To generate a stream of image data or update pixel values, we need a more persistent image producer.

Updating a MemoryImageSource

`MemoryImageSource` can also be used to generate a sequence of images, or to update an image dynamically. In Java 1.1, this is probably the easiest way to build your own low-level animation software. This example simulates the static on a television screen. It generates successive frames of random black and white pixels and displays each frame when it is complete. The figure below shows one frame of random static, followed by the code:

[Graphic: Figure from the text]

```
import java.awt.*;
import java.awt.image.*;
public class StaticGenerator
        extends java.applet.Applet
        implements Runnable {
    int arrayLength, pixels[];
    MemoryImageSource source;
    Image image;
    int width, height;
    public void init() {
        width = getSize().width; height = getSize().height;
        arrayLength = width * height;
        pixels = new int [arrayLength];
        source = new MemoryImageSource(width, height, pixels, 0, width);
        source.setAnimated(true);
        image = createImage(source);
        new Thread(this).start();
    }
    public void run() {
        while (true) {
            try {
                Thread.sleep(1000/24);
            } catch( InterruptedException e ) { /* die */ }
            for (int x = 0; x < width; x++)
                for (int y = 0; y < height; y++)  {
                    boolean rand = Math.random() > 0.5;
                    pixels[y*width+x] =
                        rand ? Color.black.getRGB() : Color.white.getRGB();
                }
            // Push out the new data
            source.newPixels(0, 0, width, height);
        }
    }
    public void paint( Graphics g ) {
        g.drawImage(image, 0, 0, this);
    }
}
```

The `init()` method sets up the `MemoryImageSource` that produces the sequence of images. It first computes the size of the array needed to hold the image data. It then creates a `MemoryImageSource` object that produces images the width and height of the display, using the default color model (the constructor we use assumes that we want the default). We start taking pixels from the beginning of the pixel array, and scan lines in the array have the same width as the image. Once we have created the `MemoryImageSource`, we call its `setAnimated()` method to tell it that we will be generating an image sequence. Then we use the source to create an `Image` that will display our sequence.

We next start a thread that generates the pixel data. For every element in the array, we get a random number, and set the pixel to black if the random number is greater than 0.5. Because `pixels` is an `int` array, we can't assign `Color` objects to it directly; we use `getRGB()` to extract the color components from the black and white `Color` constants. When we have filled the entire array with data, we call the `newPixels()` method, which delivers the new data to the image.

That's about all there is. We provide a very uninteresting `paint()` method, that just calls `drawImage()` to put the current state of the image on the screen. Whenever `paint()` is called, we see the latest collection of static. The image observer, which is the `Applet` itself, schedules a call to `paint()` whenever anything interesting has happened to the image. It's worth noting how simple it is to create this animation. Once we have the `MemoryImageSource`, we use it to create an image that we treat like any other image. The code that generates the image sequence can be arbitrarily complex--certainly in any reasonable example, it would be more complex than our (admittedly cheesy) static. But that complexity never infects the simple task of getting the image on the screen and updating it.

Image Producers and Consumers

In this section we'll create an image producer that generates a stream of image frames rather than just a static image. Unfortunately, it would take too many lines of code to generate anything really interesting, so we'll stick with a simple modification of our `ColorPan` example. After all, figuring out what to display is your job; I'm primarily concerned with giving you the necessary tools. After this, you should have the needed tools to implement more interesting applications.

A word of advice: if you find yourself writing image producers, you're probably making your life excessively difficult. Most situations can be handled by the dynamic `MemoryImageSource` technique that we just demonstrated. Before going to the trouble of writing an image producer, convince yourself that there isn't a simpler solution. Even if you never write an image producer yourself, it's good (like Motherhood and Apple Pie) to understand how Java's image rendering tools work.

Image consumers

First, we have to know a little more about the image consumers we'll be feeding. An image consumer implements the seven methods that are defined in the `ImageConsumer` interface. Two of these methods are overloaded versions of the `setPixels()` method that accept the actual pixel data for a region of the image. They are identical except that one takes the pixel data as an array of integers, and the other uses an array of bytes. (An array of bytes is natural when you're using an indexed color model because each pixel is specified by an index into a color array.) A call to `setPixels()` looks something like the following:

```
setPixels(x, y, width, height, colorModel, pixels, offset, scanLength);
```

`pixels` is the one-dimensional array of bytes or integers that holds the pixel data. Often, you deliver only part of the image with each call to `setPixels()`. The `x`, `y`, `width`, and `height` values define the rectangle of the image for which pixels are being delivered. `x` and `y` specify the upper left-hand corner of the chunk you're delivering, relative to the upper left-hand corner of the image as a whole. `width` specifies the width in pixels of the chunk; `height` specifies the number of scan lines in the chunk. `offset` specifies the point in `pixels` at which the data being delivered in this call to `setPixels()` starts.

Finally, scanLength indicates the width of the entire image, which is not necessarily the same as width. The pixels array must be large enough to accommodate width*length+offset elements; if it's larger, any leftover data is ignored.

We haven't said anything yet about the colorModel argument to setPixels(). In our previous example, we drew our image using the default ARGB color model for pixel values; the version of the MemoryImageSource constructor that we used supplied the default color model for us. In this example, we also stick with the default model, but this time we have to specify it explicitly. The remaining five methods of the ImageConsumer interface accept general attributes and framing information about the image:

- setHints()
- setDimensions()
- setProperties()
- setColorModel()
- imageComplete()

Before delivering any data to a consumer, the producer should call the consumer's setHints() method to pass it information about how pixels will be delivered. Hints are specified in the form of flags defined in the ImageConsumer interface. The flags are described in Table 14.2. The consumer uses these hints to optimize the way it builds the image; it's also free to ignore them.

Table 14.2: ImageConsumer setHints() Flags

Flag	**Description**
RANDOMPIXELORDER	The pixels are delivered in random order
TOPDOWNLEFTRIGHT	The pixels are delivered from top to bottom, left to right
COMPLETESCANLINES	Each call to setPixels() delivers one or more complete scan lines
SINGLEPASS	Each pixel is delivered only once
SINGLEFRAME	The pixels define a single, static image

setDimensions() is called to pass the width and height of the image when they are known.

setProperties() is used to pass a hashtable of image properties, stored by name. This method isn't particularly useful without some prior agreement between the producer and consumer about what properties are meaningful. For example, image formats such as GIF and TIFF can include additional information about the image. These image attributes could be delivered to the consumer in the hashtable.

setColorModel() is called to tell the consumer which color model will be used to process most of the pixel data. However, remember that each call to setPixels() also specifies a ColorModel for its group of pixels. The color model specified in setColorModel() is really only a hint that the consumer can use for optimization. You're not required to use this color model to deliver all (or for that matter, any) of the pixels in the image.

The producer calls the consumer's imageComplete() method when it has completely delivered the image or a frame of an image sequence. If the consumer doesn't wish to receive further frames of the image, it should unregister itself from the producer at this point. The producer passes a status flag formed from the flags shown in Table 14.3.

Table 14.3: ImageConsumer imageComplete() Flags

Flag	Description
STATICIMAGEDONE	A single static image is complete
SINGLEFRAMEDONE	One frame of an image sequence is complete
IMAGEERROR	An error occurred while generating the image

As you can see, the ImageProducer and ImageConsumer interfaces provide a very flexible mechanism for distributing image data. Now let's look at a simple producer.

A sequence of images

The following class, ImageSequence, shows how to implement an ImageProducer that generates a sequence of images. The images are a lot like the ColorPan image we generated a few pages back, except that the blue component of each pixel changes with every frame. This image producer doesn't do anything you couldn't do with a MemoryImageSource. It reads ARGB data from an array, and consults the object that creates the array to give it an opportunity to update the data between each frame.

This is a complex example, so before diving into the code, let's take a broad look at the pieces. The ImageSequence class is an image producer; it generates data and sends it to image consumers to be displayed. To make our design more modular, we define an interface called FrameARGBData that describes how our rendering code provides each frame of ARGB pixel data to our producer. To do the computation and provide the raw bits, we create a class called ColorPanCycle that implements FrameARGBData. This means that ImageSequence doesn't care specifically where the data comes from; if we wanted to draw different images, we could just drop in another class, provided that the new class implements FrameARGBData. Finally, we create an applet called UpdatingImage that includes two image consumers to display the data.

Here's the ImageSequence class:

```
import java.awt.image.*;
import java.util.*;
public class ImageSequence extends Thread implements ImageProducer {
    int width, height, delay;
    ColorModel model = ColorModel.getRGBdefault();
    FrameARGBData frameData;
    private Vector consumers = new Vector();
    public void run() {
        while ( frameData != null ) {
            frameData.nextFrame();
            sendFrame();
            try {
                sleep( delay );
            } catch ( InterruptedException e ) {}
        }
    }
    public ImageSequence(FrameARGBData src, int maxFPS ) {
        frameData = src;
        width = frameData.size().width;
        height = frameData.size().height;
        delay = 1000/maxFPS;
        setPriority( MIN_PRIORITY + 1 );
    }
    public synchronized void addConsumer(ImageConsumer c) {
        if ( isConsumer( c ) )
            return;
        consumers.addElement( c );
        c.setHints(ImageConsumer.TOPDOWNLEFTRIGHT |
                 ImageConsumer.SINGLEPASS );
        c.setDimensions( width, height );
        c.setProperties( new Hashtable() );
        c.setColorModel( model );
    }
    public synchronized boolean isConsumer(ImageConsumer c) {
        return ( consumers.contains( c ) );
    }
    public synchronized void removeConsumer(ImageConsumer c) {
        consumers.removeElement( c );
    }
    public void startProduction(ImageConsumer ic) {
        addConsumer(ic);
    }
    public void requestTopDownLeftRightResend(ImageConsumer ic) { }
    private void sendFrame() {
        for ( Enumeration e = consumers.elements(); e.hasMoreElements();  ) {
            ImageConsumer c = (ImageConsumer)e.nextElement();
            c.setPixels(0, 0, width, height, model, frameData.getPixels(), 0, width);
            c.imageComplete(ImageConsumer.SINGLEFRAMEDONE);
        }
    }
}
```

The bulk of the code in `ImageSequence` creates the skeleton we need for implementing the `ImageProducer` interface. `ImageSequence` is actually a simple subclass of `Thread` whose `run()` method loops, generating and sending a frame of data on each iteration. The `ImageSequence` constructor takes two items: a `FrameARGBData` object that updates the array of pixel data for each frame, and an integer that specifies the maximum number of frames per second to generate. We give the thread a low priority (`MIN_PRIORITY+1`) so that it can't run away with all of our CPU time.

Our `FrameARGBData` object implements the following interface:

```
interface FrameARGBData {
    java.awt.Dimension size();
    int [] getPixels();
    void nextFrame();
}
```

In `ImageSequence`'s `run()` method, we call `nextFrame()` to compute the array of pixels for each frame. After computing the pixels, we call our own `sendFrame()` method to deliver the data to the consumers. `sendFrame()` calls `getPixels()` to retrieve the updated array of pixel data from the `FrameARGBData` object. `sendFrame()` then sends the new data to all of the consumers by invoking each of their `setPixels()` methods and signaling the end of the frame with `imageComplete()`. Note that `sendFrame()` can handle multiple consumers; it iterates through a `Vector` of image consumers. In a more realistic implementation, we would also check for errors and notify the consumers if any occurred.

The business of managing the `Vector` of consumers is handled by `addConsumer()` and the other methods in the

ImageProducer interface. addConsumer() adds an item to consumers. A Vector is a perfect tool for this task, since it's an automatically extendable array, with methods for finding out how many elements it has, whether or not a given element is already a member, and so on.

addConsumer() also gives the consumer hints about how the data will be delivered by calling setHints(). This image provider always works from top to bottom and left to right, and makes only one pass through the data. addConsumer() next gives the consumer an empty hashtable of image properties. Finally, it reports that most of the pixels will use the default ARGB color model (we initialized the variable model to ColorModel.getRGBDefault()). In this example, we always start sending image data on the next frame, so startProduction() simply calls addConsumer().

We've discussed the mechanism for communications between the consumer and producer, but I haven't yet told you where the data comes from. We have a FrameARGBData interface that defines how to retrieve the data, but we don't yet have an object that implements the interface. The following class, ColorPanCycle, implements FrameARGBData; we'll use it to generate our pixels:

```
import java.awt.*;
class ColorPanCycle implements FrameARGBData {
    int frame = 0, width, height;
    private int [] pixels;
    ColorPanCycle ( int w, int h ) {
        width = w;
        height = h;
        pixels = new int [ width * height ];
        nextFrame();
    }
    public synchronized int [] getPixels() {
        return pixels;
    }
    public synchronized void nextFrame() {
        int index = 0;
        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                int red = (y * 255) / (height - 1);
                int green = (x * 255) / (width - 1);
                int blue = (frame * 10) & 0xff;
                pixels[index++] =
                    (255 << 24) | (red << 16) | (green << 8) | blue;
            }
        }
        frame++;
    }
    public Dimension size() {
        return new Dimension ( width, height );
    }
}
```

ColorPanCycle is like our previous ColorPan example, except that it adjusts each pixel's blue component each time nextFrame() is called. This should produce a color cycling effect; as time goes on, the image becomes more blue.

Now let's put the pieces together by writing an applet that displays a sequence of changing images: UpdatingImage. In fact, we'll do better than displaying one sequence. To prove that ImageSequence really can deal with multiple consumers, UpdatingImage creates two components that display different views of the image. Once the mechanism has been set up, it's surprising how little code you need to add additional displays.

```
import java.awt.*;
import java.awt.image.*;
public class UpdatingImage extends java.applet.Applet {
    ImageSequence seq;
    public void init() {
        seq = new ImageSequence( new ColorPanCycle(100, 100), 10);
        setLayout( null );
        add( new ImageCanvas( seq, 50, 50 ) );
        add( new ImageCanvas( seq, 100, 100 ) );
        seq.start();
    }
    public void stop() {
        if ( seq != null ) {
            seq.stop();
            seq = null;
        }
    }
}
class ImageCanvas extends Canvas {
    Image img;
    ImageProducer source;
    ImageCanvas ( ImageProducer p, int w, int h ) {
        source = p;
        setSize( w, h );
    }
    public void update( Graphics g ) {
        paint(g);
    }
    public void paint( Graphics g ) {
        if ( img == null )
            img = createImage( source );
        g.drawImage( img, 0, 0, getSize().width, getSize().height, this );
    }
}
```

UpdatingImage constructs a new ImageSequence producer with an instance of our ColorPanCycle object as its frame source. It then creates two ImageCanvas components that create and display the two views of our animation. ImageCanvas is a subclass of Canvas; it takes an ImageProducer and a width and height in its constructor and creates and displays an appropriately scaled version of the image in its paint() method. UpdatingImage places the smaller view on top of the larger one for a sort of "picture in picture" effect.

If you've followed the example to this point, you're probably wondering where in the heck is the image consumer. After all, we spent a lot of time writing methods in ImageSequence for the consumer to call. If you look back at the code, you'll see that an ImageSequence object gets passed to the ImageCanvas constructor, and that this object is used as an argument to createImage(). But nobody appears to call addConsumer(). And the image producer calls setPixels() and other consumer methods; but it always digs a consumer out of its Vector of registered consumers, so we never see where these consumers come from.

In UpdatingImage, the image consumer is behind the scenes, hidden deep inside the Canvas--in fact, inside the Canvas' peer. The call to createImage() tells its component (i.e., our canvas) to become an image consumer. Something deep inside the component is calling addConsumer() behind our backs and registering a mysterious consumer, and that consumer is the one the producer uses in calls to setPixels() and other methods. We haven't implemented any ImageConsumer objects in this book because, as you might imagine, most image consumers are implemented in native code, since they need to display things on the screen. There are others though; the java.awt.image.PixelGrabber class is a consumer that returns the pixel data as a byte array. You might use it to save an image. You can make your own consumer do anything you like with pixel data from a producer. But in reality, you rarely need to write an image consumer yourself. Let them stay hidden; take it on faith that they exist.

Now for the next question: How does the screen get updated? Even though we are updating the consumer with new data, the new image will not appear on the display unless the applet repaints it periodically. By now, this part of the machinery should be familiar: what we need is an image observer. Remember that all components are image observers (i.e., the class Component implements ImageObserver). The call to drawImage() specifies our ImageCanvas as its image observer. The default Component class-image-observer functionality then repaints our image whenever new pixel data arrives.

In this example, we haven't bothered to stop and start our applet properly; it continues running and wasting CPU time even when it's invisible. There are two strategies for stopping and restarting our thread. We can destroy the thread and create a new one, which would require recreating our ImageCanvas objects, or we could suspend and resume the active thread. As discussed in Chapter 6, *Threads*, neither option is particularly difficult.

Filtering Image Data

As I said above, you rarely need to write an image consumer. However, there is one kind of image consumer that's worth knowing about. In this final section on images, we'll build a simple image filter. An image filter is simply a class that performs some work on image data before passing the data to another consumer.

The ColorSep applet acquires an image; uses an image filter to separate the image into red, green, and blue components; and displays the three resulting images. With this applet and a few million dollars, you could build your own color separation plant.

```
import java.awt.*;
import java.awt.image.*;
public class ColorSep extends java.applet.Applet {
    Image img, redImg, greenImg, blueImg;
    public void init() {
        img = getImage( getClass().getResource( getParameter("img")) );
        redImg = createImage(new FilteredImageSource(img.getSource(),
                                        new ColorMaskFilter( Color.red )));
        greenImg = createImage(new FilteredImageSource(img.getSource(),
                                        new ColorMaskFilter( Color.green )));
        blueImg = createImage(new FilteredImageSource(img.getSource(),
                                        new ColorMaskFilter( Color.blue )));
    }
    public void paint( Graphics g ) {
        int width = getSize().width, height = getSize().height;
        g.drawImage( redImg, 0, 0, width/3, height, this );
        g.drawImage( greenImg, width/3, 0, width/3, height, this );
        g.drawImage( blueImg, 2*width/3, 0, width/3, height, this );
    }
}
class ColorMaskFilter extends RGBImageFilter {
    Color color;
    ColorMaskFilter( Color mask ) {
        color = mask;
        canFilterIndexColorModel = true;
    }
    public int filterRGB(int x, int y, int pixel ) {
        return
            255 << 24 |
            (((pixel & 0xff0000) >> 16) * color.getRed()/255) << 16 |
            (((pixel & 0xff00) >> 8) * color.getGreen()/255) << 8 |
            (pixel & 0xff) * color.getBlue()/255 ;
    }
}
```

The FilteredImageSource and RGBImageFilter classes form the basis for building and using image filters. A FilteredImageSource is an image producer (like MemoryImageSource) that is constructed from an image and an ImageFilter object. It fetches pixel data from the image and feeds it through the image filter before passing the data along. Because FilteredImageSource is an image producer, we can use it in our calls to createImage().

But where's the consumer? FilteredImageSource obviously consumes image data as well as producing it. The image consumer is still mostly hidden, but is peeking out from under its rock. Our class ColorMaskFilter extends RGBImageFilter, which in turn extends ImageFilter. And ImageFilter is (finally!) an image consumer. Of course, we still don't see the calls to addConsumer(), and we don't see an implementation of setPixels(); they're hidden in the ImageFilter sources and inherited by ColorMaskFilter.

So what does ColorMaskFilter actually do? Not much. ColorMaskFilter is a simple subclass of RGBImageFilter that implements one method, filterRGB(), through which all of the pixel data are fed. Its constructor saves a mask value we use for filtering. The filterRGB() method accepts a pixel value, along with its x and y coordinates, and returns the filtered version of the pixel. In ColorMaskFilter, we simply multiply the color components by the mask color to get the proper effect. A more complex filter, however, might use the coordinates to change its behavior based on the pixel's position.

One final detail: the constructor for ColorMaskFilter sets the flag canFilterIndexColorModel. This flag is inherited from RGBImageFilter. It means our filter doesn't depend on the pixel's position. In turn, this means it can filter the colors in a color table. If we were using an indexed color model, filtering the color table would be much faster than filtering the individual pixels.

PREVIOUS | HOME | NEXT
Drawing Techniques | BOOK INDEX | Working with Audio

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Glossary

abstract

> The `abstract` keyword is used to declare abstract methods and classes. An abstract method has no implementation defined; it is declared with arguments and a return type as usual, but the body enclosed in curly braces is replaced with a semicolon. The implementation of an abstract method is provided by a subclass of the class in which it is defined. If an abstract method appears in a class, the class is also abstract.

API (Application Programming Interface)

> An API consists of the functions and variables programmers are use in their applications. The Java API consists of all `public` and `protected` methods of all `public` classes in the `java.applet`, `java.awt`, `java.awt.image`, `java.awt.peer`, `java.io`, `java.lang`, `java.net`, and `java.util` packages.

applet

> An embedded Java application that runs in the context of an applet viewer, such as a Web browser.

<applet> tag

> HTML tag that specifies an applet run within a Web document.

applet viewer

> An application that implements the additional structure needed to run and display Java applets. An applet viewer can be a Web browser like HotJava or Netscape Navigator, or a separate program like Sun's *appletviewer*.

application

A Java program that runs standalone; i.e., it doesn't require an applet viewer.

AWT (Abstract Windowing Toolkit)

Java's platform-independent windowing, graphics, and user-interface toolkit.

boolean

A primitive Java data type that contains a truth value. The two possible values of a `boolean` variable are `true` and `false`.

byte

A primitive Java data type that's an 8-bit two's-complement signed number (in all implementations).

callback

A behavior that is defined by one object and then later invoked by another object when a particular event occurs.

cast

A technique that explicitly converts one data type to another.

catch

The `catch` statement introduces an exception-handling block of code following a `try` statement. The `catch` keyword is followed by an exception type and argument name in parentheses, and a block of code within curly braces.

char

A primitive Java data type; a variable of type `char` holds a single 16-bit Unicode character.

class

a) An encapsulated collection of data and methods to operate on the data. A class may be instantiated to produce an object that's an instance of the class.

b) The `class` keyword is used to declare a class, thereby defining a new object type. Its syntax is similar to the `struct` keyword in C.

class loader

An object in the Java security model that is responsible for loading Java binary classes from the network into the local interpreter. A class loader keeps its classes in a separate namespace, so that loaded classes cannot interact with system classes and breach system security.

class method

A method declared `static`. Methods of this type are not passed implicit `this` references and may refer only to class variables and invoke other class methods of the current class. A class method may be invoked through the class name, rather than through an instance of the class.

class path

The directory path specifying the location of compiled Java class files on the local system.

class variable

A variable declared `static`. Variables of this type are associated with the class, rather than with a particular instance of the class. There is only one copy of a static variable, regardless of the number of instances of the class that are created.

client

The application that initiates a conversation as part of a networked client/server application. See server.

compilation unit

The source code for a Java class. A compilation unit normally contains a single class definition, and in most current development environments is just a file with a *.java* extension.

compiler

A program that translates source code into executable code.

component

> Any of the GUI primitives implemented in the `java.awt` package as subclasses of `Component`. The classes `Button`, `Choice`, and `TextField` (among many others) are components.

composition

> Using objects as part of another, more complex object. When you compose a new object, you create complex behavior by delegating tasks to the internal objects. Composition is different from inheritance, which defines a new object by changing or refining the behavior of an old object. See inheritance.

constructor

> A method that is invoked automatically when a new instance of a class is created. Constructors are used to initialize the variables of the newly created object. The constructor method has the same name as the class.

container

> One of the `java.awt` classes that "contain" GUI components. Components in a container appear within the boundaries of the container. The classes `Dialog`, `Frame`, `Panel`, and `Window` are containers.

content handler

> A class that recognizes the content type of particular data, parses it, and converts it to an appropriate object.

datagram

> A packet of data sent to a receiving computer without warning, error checking, or other control information.

data hiding

> See encapsulation.

double

A Java primitive data type;a `double` value is a 64-bit (double-precision) floating-point number.

encapsulation

An object-oriented programming technique that makes an object's data `private` or `protected` (i.e., hidden) and allows programmers to access and manipulate that data only through method calls. Done well, encapsulation reduces bugs and promotes reusability and modularity of classes. This technique is also known as data hiding.

event

A user's action, such as a mouse click or key press.

exception

A signal that some unexpected condition has occurred in the program. In Java, exceptions are objects that are subclasses of `Exception` or `Error` (which themselves are subclasses of `Throwable`). Exceptions in Java are "raised" with the `throw` keyword and received with the `catch` keyword. See `throw`, `throws`, and `catch`.

extends

The `extends` keyword is used in a `class` declaration to specify the superclass of the class being defined. The class being defined has access to all the `public` and `protected` variables and methods of the superclass (or, if the class being defined is in the same package, it has access to all non-`private` variables and methods). If a class definition omits the `extends` clause, its superclass is taken to be `java.lang.Object`.

final

The `final` keyword is a modifier that may be applied to classes, methods, and variables. It has a similar, but not identical meaning in each case. When `final` is applied to a class, it means that the class may never be subclassed. `java.lang.System` is an example of a `final` class. When `final` is applied to a variable, the variable is a constant; i.e., it can't be modified.

finalize

`finalize` is not actually a Java keyword, but a reserved method name. The finalizer is called when an object is no longer being used (i.e., when there are no further references to it), but before the object's memory is actually reclaimed by the system. A finalizer should perform cleanup tasks and free system resources not handled by Java's garbage-collection system.

finally

This keyword introduces the `finally` block of a `try`/`catch`/`finally` construct. `catch` and `finally` blocks provide exception handling and routine cleanup for code in a `try` block. The `finally` block is optional, and appears after the `try` block, and after zero or more `catch` blocks. The code in a `finally` block is executed once, regardless of how the code in the `try` block executes. In normal execution, control reaches the end of the `try` block and proceeds to the `finally` block, which generally performs any necessary cleanup.

float

A Java primitive data type; a `float` value is a 32-bit (single-precision) floating-point number represented in IEEE 754 format.

garbage collection

The process of reclaiming the memory of objects no longer in use. An object is no longer in use when there are no references to it from other objects in the system and no references in any local variables on the method call stack.

GC

An abbreviation for garbage collection or garbage collector (or occasionally "graphics context").

graphics context

A drawable surface represented by the `java.awt.Graphics` class. A graphics context contains contextual information about the drawing area and provides methods for performing drawing operations in it.

GUI (graphical user interface)

A GUI is a user interface constructed from graphical push buttons, text fields, pull-down menus, dialog boxes, and other standard interface components. In Java, GUIs are implemented with the classes in the `java.awt` package.

hashcode

An arbitrary-looking identifying number used as a kind of signature for an object. A hashcode stores an object in a hashtable. See hashtable.

hashtable

> An object that is like a dictionary or an associative array. A hashtable stores and retrieves elements using key values called hashcodes. See hashcode.

hostname

> The name given to an individual computer attached to the Internet.

HotJava

> A WWW browser written in Java that is capable of downloading and running Java applets.

ImageConsumer

> An interface for receiving image data from an image source. Image consumers are usually implemented by the `awt.peer` interface, so they are largely invisible to programmers.

ImageObserver

> An interface in the `java.awt.image` package that receives information about the status of an image being constructed by a particular `ImageConsumer`.

ImageProducer

> An interface in the `java.awt.image` package that represents an image source (i.e., a source of pixel data).

implements

> The `implements` keyword is used in class declarations to indicate that the class implements the named interface or interfaces. The `implements` clause is optional in class declarations; if it appears, it must follow the `extends` clause (if any). If an `implements` clause appears in the declaration of a non-`abstract` class, every method from each specified interface must be implemented by the class or by one of its superclasses.

import

> The `import` statement makes Java classes available to the current class under an abbreviated name. (Java classes are always available by their fully qualified name, assuming the appropriate class file can be found relative to the `CLASSPATH` environment variable and that the class file is

readable. `import` doesn't make the class available; it just saves typing and makes your code more legible). Any number of `import` statements may appear in a Java program. They must appear, however, after the optional `package` statement at the top of the file, and before the first class or interface definition in the file.

inheritance

An important feature of object-oriented programming that involves defining a new object by changing or refining the behavior of an existing object. That is, an object implicitly contains all the non-`private` variables of its superclass and can invoke all the non-`private` methods of its superclass. Java supports single inheritance of classes and multiple inheritance of interfaces.

instance

An object. When a class is instantiated to produce an object, we say the object is an instance of the class.

instance method

A non-`static` method of a class. Such a method is passed an implicit `this` reference to the object that invoked it. See also class method and `static`.

instanceof

`instanceof` is a Java operator that returns `true` if the object on its left-hand side is an instance of the class (or implements the interface) specified on its right-hand side. `instanceof` returns `false` if the object is not an instance of the specified class or does not implement the specified interface. It also returns `false` if the specified object is `null`.

instance variable

A non-`static` variable of a class. Copies of such variables occur in every instance of the created class. See also class variable and `static`.

int

A primitive Java data type that's a 32-bit two's-complement signed number (in all implementations).

interface

The `interface` keyword is used to declare an interface. More generally, an interface defines a list of methods that enables a class to implement the interface itself.

interpreter

The module that decodes and executes Java bytecode.

ISO8859-1

An 8-bit character encoding standardized by the ISO. This encoding is also known as Latin-1 and contains characters from the Latin alphabet suitable for English and most languages of western Europe.

ISO10646

A 4-byte character encoding that includes all of the world's national standard character encodings. Also known as UCS. The 2-byte Unicode character set maps to the range 0x00000000 to 0x0000FFFF of ISO 10646.

Java WorkShop

Sun's Web browser-based tool written in Java for the development of Java applications.

JDK (Java Development Kit)

A package of software distributed by Sun Microsystems for Java developers. It includes the Java interpreter, Java classes, and Java development tools: compiler, debugger, disassembler, appletviewer, stub file generator, and documentation generator.

JavaScript

A language for creating dynamic Web pages developed by Netscape. From a programmer's point of view, it's unrelated to Java, although some of its capabilities are similar. Internally, there may be a relationship, but even that is unclear.

layout manager

An object that controls the arrangement of components within the display area of a container. The `java.awt` package contains a number of layout managers that provide different layout styles.

Latin-1

A nickname for ISO8859-1.

local variable

A variable that is declared inside a single method. A local variable can be seen only by code within that method.

long

A primitive Java data type that's a 64-bit two's-complement signed number (in all implementations).

method

The object-oriented programming term for a function or procedure.

method overloading

Providing definitions of more than one method with the same name but with different argument lists or return values. When an overloaded method is called, the compiler determines which one is intended by examining the supplied argument types.

method overriding

Defining a method that exactly matches (i.e., same name, same argument types, and same return type) a method defined in a superclass. When an overridden method is invoked, the interpreter uses "dynamic method lookup" to determine which method definition is applicable to the current object.

modifier

A keyword placed before a class, variable, or method that alters the item's accessibility, behavior, or semantics. See `abstract`, `final`, `native`, `private`, `private protected`, `protected`, `public`, `static`, and `synchronized`.

Model/View/Controller (MVC) framework

A user-interface design that originated in Smalltalk. In MVC, the data for a display item is called the "model." A "view" displays a particular representation of the model, and a "controller" provides user interaction with both. Java incorporates many MVC concepts.

NaN (not-a-number)

This is a special value of the `double` and `float` data types that represents an undefined result of a mathematical operation, such as zero divided by zero.

native

`native` is a modifier that may be applied to method declarations. It indicates that the method is implemented (elsewhere) in C, or in some other platform-dependent fashion. A `native` method should have a semicolon instead of a body. A `native` method cannot be `abstract`, but all other method modifiers may be used with `native` methods.

native method

A method that is implemented in a native language on a host platform, rather than being implemented in Java. Native methods provide access to such resources as the network, the windowing system, and the host filesystem.

new

`new` is a unary operator that creates a new object or array (or raises an `OutOfMemoryException` if there is not enough memory available).

null

`null` is a special value that indicates a variable doesn't refer to any object. The value `null` may be assigned to any class or interface variable. It cannot be cast to any integral type, and should not be considered equal to zero, as in C.

object

An instance of a class. A class models a group of things; an object models a particular member of that group.

package

The `package` statement specifies which package the code in the file is part of. Java code that is part of a particular package has access to all classes (`public` and non-`public`) in the package, and all non-`private` methods and fields in all those classes. When Java code is part of a named package, the compiled class file must be placed at the appropriate position in the `CLASSPATH` directory hierarchy before it can be accessed by the Java interpreter or other utilities. If the `package` statement is omitted from a file, the code in that file is part of an unnamed default package. This is convenient for small test programs, or during development because it means the code can be interpreted from the current directory.

<param> tag

HTML tag used within `<applet>` ... `</applet>` to specify a named parameter and string value to an applet within a Web page.

peer

The actual implementation of a GUI component on a specific platform. Peer components reside within a `Toolkit` object. See `Toolkit`.

primitive type

One of the Java data types: `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double`. Primitive types are manipulated, assigned, and passed to methods "by value" (i.e., the actual bytes of the data are copied). See also reference type.

private

The `private` keyword is a visibility modifier that can be applied to method and field variables of classes. A `private` field is not visible outside its class definition.

private protected

When the `private` and `protected` visibility modifiers are both applied to a variable or method in a class, they indicate the field is visible only within the class itself and within subclasses of the class. Note that subclasses can access only `private protected` fields within themselves or within other objects that are subclasses; they cannot access those fields within instances of the superclass.

protected

The `protected` keyword is a visibility modifier that can be applied to method and field variables of classes. A `protected` field is visible only within its class, within subclasses, or within the package of which its class is a part. Note that subclasses in different packages can access only `protected` fields within themselves or within other objects that are subclasses; they cannot access protected fields within instances of the superclass.

protocol handler

> Software that describes and enables the use of a new protocol. A protocol handler consists of two classes: a `StreamHandler` and a `URLConnection`.

public

> The `public` keyword is a visibility modifier that can be applied to classes and interfaces and to the method and field variables of classes and interfaces. A `public` class or interface is visible everywhere. A non-`public` class or interface is visible only within its package. A `public` method or variable is visible everywhere its class is visible. When none of the `private`, `protected` or `public` modifiers is specified, a field is visible only within the package of which its class is a part.

reference type

> Any object or array. Reference types are manipulated, assigned, and passed to methods "by reference." In other words, the underlying value is not copied; only a reference to it is. See also primitive type.

root

> The base of a hierarchy, such as a root class, whose descendants are subclasses. The `java.lang.Object` class serves as the root of the Java class hierarchy.

SecurityManager

> The Java class that defines the methods the system calls to check whether a certain operation is permitted in the current environment.

server

> The application that accepts a request for a conversation as part of a networked client/server application. See client.

shadow

> To declare a variable with the same name as a variable defined in a superclass. We say the variable "shadows" the superclass's variable. Use the `super` keyword to refer to the shadowed variable, or refer to it by casting the object to the type of the superclass.

short

A primitive Java data type that's a 16-bit two's-complement signed number (in all implementations).

socket

An interface that listens for connections from clients on a data port and connects the client data stream with the receiving application.

static

The `static` keyword is a modifier applied to method and variable declarations within a class. A `static` variable is also known as a class variable as opposed to non-`static` instance variables. While each instance of a class has a full set of its own instance variables, there is only one copy of each `static` class variable, regardless of the number of instances of the class (perhaps zero) that are created. `static` variables may be accessed by class name or through an instance. Non-`static` variables can be accessed only through an instance.

stream

A flow of data, or a channel of communication. All fundamental I/O in Java is based on streams.

String

A class used to represent textual information. The `String` class includes many methods for operating on string objects. Java overloads the + operator for string concatenation.

subclass

A class that extends another. The subclass inherits the `public` and `protected` methods and variables of its superclass. See `extends`.

super

The keyword `super` refers to the same value as `this`: the instance of the class for which the current method (these keywords are valid only within non-`static` methods) was invoked. While the type of `this` is the type of the class in which the method appears, the type of `super` is the type of the superclass of the class in which the method appears. `super` is usually used to refer to superclass variables shadowed by variables in the current class. Using `super` in this way is equivalent to casting `this` to the type of the superclass.

superclass

A class extended by some other class. The superclass's `public` and `protected` methods and variables are available to the subclass. See `extends`.

synchronized

The `synchronized` keyword is used in two related ways in Java: as a modifier and as a statement. First, it is a modifier applied to class or instance methods. It indicates that the method modifies the internal state of the class or the internal state of an instance of the class in a way that is not thread-safe. Before running a `synchronized` class method, Java obtains a lock on the class, to ensure that no other threads can modify the class concurrently. Before running a `synchronized` instance method, Java obtains a lock on the instance that invoked the method, ensuring that no other threads can modify the object at the same time.

Java also supports a `synchronized` statement that serves to specify a "critical section" of code. The `synchronized` keyword is followed by an expression in parentheses, and a statement or block of statements. The expression must evaluate to an object or array. Java obtains a lock on the specified object or array before executing the statements.

TCP

Transmission Control Protocol. A connection-oriented, reliable protocol. One of the protocols on which the Internet is based.

this

Within an instance method or constructor of a class, `this` refers to "this object"--the instance currently being operated on. It is useful to refer to an instance variable of the class that has been shadowed by a local variable or method argument. It is also useful to pass the current object as an argument to static methods or methods of other classes.

There is one additional use of `this`: when it appears as the first statement in a constructor method, it refers to one of the other constructors of the class.

thread

A single, independent stream of execution within a program. Since Java is a "multithreaded" programming language, more than one thread may be running within the Java interpreter at a time. Threads in Java are represented and controlled through the `Thread` object.

throw

The `throw` statement signals that an exceptional condition has occurred by throwing a specified exception object. This statement stops program execution and resumes it at the nearest containing `catch` statement that can handle the specified exception object. Note that the `throw` keyword must be followed by an exception object, not an exception class.

throws

The `throws` keyword is used in a method declaration to list the exceptions the method can throw. Any exceptions a method can raise that are not subclasses of `Error` or `RuntimeException` must either be caught within the method or declared in the method's `throws` clause.

Toolkit

The property of the Java API that defines the look and feel of the user interface on a specific platform.

try

The `try` keyword indicates a block of code to which subsequent `catch` and `finally` clauses apply. The `try` statement itself performs no special action. See the entries for `catch` and `finally` for more information on the `try`/`catch`/`finally` construct.

UCS (universal character set)

A synonym for ISO10646.

UDP

User Datagram Protocol. A connectionless unreliable protocol. UDP describes a network data connection based on datagrams with little packet control.

Unicode

A 16-bit character encoding that includes all of the world's commonly used alphabets and ideographic character sets in a "unified" form (i.e., a form from which duplications among national standards have been removed). ASCII and Latin-1 characters may be trivially mapped to Unicode characters. Java uses Unicode for its `char` and `String` types.

UTF-8 (UCS transformation format 8-bit form)

An encoding for Unicode characters (and more generally, UCS characters) commonly used for transmission and storage. It is a multibyte format in which different characters require different numbers of bytes to be represented.

vector

A dynamic array of elements.

verifier

A theorem prover that steps through the Java byte-code before it is run and makes sure that it is well-behaved. The byte-code verifier is the first line of defense in Java's security model.

PREVIOUS
Working with Audio

HOME
BOOK INDEX

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | R | S | T | U | V | W | X | Y | Z

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | R | S | T | U | V | W | X | Y | Z

HOME

O'Reilly Copyright Statement

The electronic versions of *Java in a Nutshell*, *Java Language Reference*, *Java AWT Reference*, *Java Fundamental Classes Reference*, and *Exploring Java* are copyright © 1996, 1997, 1998 by O'Reilly & Associates, Inc.; all rights reserved.

This CD-ROM is intended for use by one individual. As such, you may make copies for your own personal use. However, you may not provide copies to others, or make this reference library availible over a LAN or other network. You may not reprint, offer for sale, or otherwise re-use material from this book without the explicit written permission of O'Reilly & Associates, Inc. If you would like to discuss licensing for multiple users, contact service@oreilly.com.

You can purchase print editions of these books directly from O'Reilly & Associates, Inc. (see http://www.oreilly.com/order/) or from bookstores that carry O'Reilly & Associates books.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly & Associates, Inc., was aware of the trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Library Home | Java in a Nutshell | Java Language Reference | Java AWT Reference | Java Fundamental Classes Reference | Exploring Java

This product is printed with 100% recycled electrons.

1. Introduction

Contents:

The phenomenon that is Java continues to capture new supporters every day. What began as a programming environment for writing fancy animation applets that could be embedded in web browsers is growing up to be a sophisticated platform for delivering all kinds of portable, distributed applications. If you are already an experienced Java programmer, you know just how powerful the portability of Java is. If you are just now discovering Java, you'll be happy to know that the days of porting applications are over. Once you write a Java application, it can run on UNIX workstations, PCs, and Macintosh computers, as well as on many other supported platforms.

This book is a complete programmer's reference to the "fundamental classes" in the Java programming environment. The fundamental classes in the Java Development Kit (JDK) provide a powerful set of tools for creating portable applications; they are an important component of the toolbox used by every Java programmer. This reference covers the classes in the `java.lang`, `java.io`, `java.net`, `java.util`, `java.lang.reflect`, `java.math`, `java.text`, and `java.util.zip` packages. This chapter offers an overview of the fundamental classes in each of these packages.

This reference assumes you are already familiar with the Java language and class libraries. If you aren't, *Exploring Java*, by Pat Niemeyer and Josh Peck, provides a general introduction, and other books in the O'Reilly Java series provide detailed references and tutorials on specific topics. Note that the material herein does not cover the classes that comprise the Abstract Window Toolkit (AWT): the AWT is covered by a companion volume, the *Java AWT Reference*, by John Zukowski. In addition, this book does not cover any of the new "enterprise" APIs in the core 1.1 JDK, such as the classes in the `java.rmi`, `java.sql`, and `java.security` packages. These packages will be covered by forthcoming books on distributed computing and database programming. See the Preface for a complete list of titles in the O'Reilly Java series.

You should be aware that this book covers two versions of Java: 1.0.2 and 1.1. Version 1.1 of the Java Development Kit (JDK) was released in February 1997. This release includes many improvements and additions to the fundamental Java classes; it represents a major step forward in the evolution of Java. Although Java 1.1 has a number of great new features, you may not want to switch to the new version right away, especially if you are writing mostly Java applets. You'll need to keep an eye on the state of Java support in browsers to help you decide when to switch to Java 1.1. Of course, if you are writing Java applications, you can take the plunge today.

This chapter points out new features of Java 1.1 as they come up. However, there is one "feature" that deserves mention that doesn't fit naturally into an overview. As of Java 1.1, classes, methods, and constructors available in Java 1.0.2 can be deprecated in favor of new classes, methods, and constructors in Java 1.1. The Java 1.1 compiler issues a warning whenever you use a deprecated entity.

1.1 The java.lang Package

The `java.lang` package contains classes and interfaces essential to the Java language. For example, the `Object` class is the ultimate superclass of all other classes in Java. `Object` defines some basic methods for thread synchronization that are inherited by all Java classes. In addition, `Object` defines basic methods for equality testing, hashcode generation, and string conversion that can be overridden by subclasses when appropriate.

The `java.lang` package also contains the `Thread` class, which controls the operation of each thread in a multithreaded application. A `Thread` object can be used to start, stop, and suspend a thread. A `Thread` must be associated with an object that implements the `Runnable` interface; the `run()` method of this interface specifies what the thread actually does. See Chapter 3, *Threads*, for a more detailed explanation of how threads work in Java.

The `Throwable` class is the superclass of all error and exception classes in Java, so it defines the basic functionality of all such classes. The `java.lang` package also defines the standard error and exception classes in Java. The error and exception hierarchies are rooted at the `Error` and `Exception` subclasses of `Throwable`. See Chapter 4, *Exception Handling*, for more information about the exception-handling mechanism.

The `Boolean`, `Character`, `Byte`, `Double`, `Float`, `Integer`, `Long`, and `Short` classes encapsulate the Java primitive data types. `Byte` and `Short` are new in Java 1.1, as is the `Void` class.

All of these classes are necessary to support the new Reflection API and class literals in Java 1.1 The `Class` class also has a number of new methods in Java 1.1 to support reflection.

All strings in Java are represented by `String` objects. These objects are immutable. The `StringBuffer` class in `java.lang` can be used to work with mutable text strings. Chapter 2, *Strings and Related Classes*, offers a more detailed description of working with strings in Java.

See Chapter 12, *The java.lang Package*, for complete reference material on all of the classes in the `java.lang` package.

PREVIOUS HOME NEXT
Acknowledgments BOOK INDEX The java.lang.reflect Package

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

JAVA
Fundamental Classes Reference

PREVIOUS Chapter 2 NEXT

2. Strings and Related Classes

Contents:

As with most programming languages, strings are used extensively throughout Java, so the Java API has quite a bit of functionality to help you manipulate strings. This chapter describes the following classes:

- The `java.lang.String` class represents all textual strings in Java. A `String` object is immutable; once you create a `String` object, there is no way to change the sequence of characters it represents or the length of the string.
- The `java.lang.StringBuffer` class represents a variable-length, mutable sequence of characters. With a `StringBuffer` object, you can insert characters anywhere in the sequence and add characters to the end of the sequence.
- The `java.util.StringTokenizer` class provides support for parsing a string into a sequence of words, or tokens.

2.1 String

You can create a `String` object in Java simply by assigning a string literal to a `String` variable:

```
String quote = "To be or not to be";
```

All string literals are compiled into `String` objects. Although the Java compiler does not generally treat expressions involving object references as compile-time constants, references to `String` objects created from string literals are treated as compile-time constants.

Of course, there are many other ways to create a `String` object. The `String` class has a number of constructors that let you create a `String` from an array of bytes, an array of characters, another `String` object, or a `StringBuffer` object.

If you are a C or C++ programmer, you may be wondering if `String` objects are null-terminated. The answer is no, and, in fact, the question is irrelevant. The `String` class actually uses a character array internally. Since arrays in Java are actual objects that know their own length, a `String` object also knows its length and does not require a special terminator. Use the `length()` method to get the length of a `String` object.

Although `String` objects are immutable, the `String` class does provide a number of useful methods for working with strings. Any operation that would otherwise change the characters or the length of the string returns a new `String` object that copies the necessary portions of the original `String`.

The following methods access the contents of a `String` object:

- `substring()` creates a new `String` object that contains a sub-sequence of the sequence of characters represented by a `String` object.

- `charAt()` returns the character at a given position in a `String` object.

- `getChars()` and `getBytes()` return a range of characters in a `char` array or a `byte` array.

- `toCharArray()` returns the entire contents of a `String` object as a `char` array.

You can compare the contents of `String` objects with the following methods:

- `equals()` returns `true` if two `String` objects have the exact same contents, while `equalsIgnoreCase()` returns `true` if two objects have the same contents ignoring differences between upper- and lowercase versions of the same character.

- `regionMatches()` determines if two sub-strings contain the same sequence of characters.

- `startsWith()` and `endsWith()` determine if a `String` object begins or ends with a particular sequence of characters.

- `compareTo()` determines if the contents of one `String` object are less than, equal to, or greater than the contents of another `String` object.

Use the following methods to search for characters in a string:

- `indexOf()` searches forward through a string for a given character or string.
- `lastIndexOf()` searches backwards through a string for a given character or string.

The following methods manipulate the contents of a string and return a new, related string:

- `concat()` returns a new `String` object that is the concatenation of two `String` objects.
- `replace()` returns a new `String` object that contains the same sequence of characters as the original string, but with a given character replaced by another given character.
- `toLowerCase()` and `toUpperCase()` return new `String` objects that contain the same sequence of characters as the original string, but converted to lower- or uppercase.
- `trim()` returns a new `String` object that contains the same character sequence as the original string, but with leading and trailing white space and control characters removed.

The `String` class also defines a number of `static` methods named `valueOf()` that return string representations of primitive Java data types and objects. The `Object` class defines a `toString()` method, and, since `Object` is the ultimate superclass of every other class, every class inherits a basic `toString()` method. Any class that has a string representation should override the `toString()` method to produce the appropriate string.

PREVIOUS HOME NEXT
The java.util.zip Package BOOK INDEX StringBuffer

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

3. Threads

Contents:

Threads provide a way for a Java program to do multiple tasks concurrently. A thread is essentially a flow of control in a program and is similar to the more familiar concept of a process. An operating system that can run more than one program at the same time uses processes to keep track of the various programs that it is running. However, processes generally do not share any state, while multiple threads within the same application share much of the same state. In particular, all of the threads in an application run in the same address space, sharing all resources except the stack. In concrete terms, this means that threads share field variables, but not local variables.

When multiple processes share a single processor, there are times when the operating system must stop the processor from running one process and start it running another process. The operating system must execute a sequence of events called a *context switch* to transfer control from one process to another. When a context switch occurs, the operating system has to save a lot of information for the process that is being paused and load the comparable information for the process being resumed. A context switch between two processes can require the execution of thousands of machine instructions. The Java virtual machine is responsible for handling context switches between threads in a Java program. Because threads share much of the same state, a context switch between two threads typically requires the execution of less than 100 machine instructions.

There are a number of situations where it makes sense to use threads in a Java program. Some programs must be able to engage in multiple activities and still be able to respond to additional input from the user. For example, a web browser should be able to respond to user input while fetching an image or playing a sound. Because threads can be suspended and resumed, they can make it easier to control multiple activities, even if the activities do not need to be concurrent. If a program models real world objects that display independent, autonomous behavior, it makes sense to use a separate thread for each object. Threads can also implement asynchronous methods, so that a calling method does not have to wait for the method it calls to complete before continuing with its own activity.

Java applets make considerable use of threads. For example, an animation is generally implemented with a separate thread. If an applet has to download extensive information, such as an image or a sound, to initialize itself, the initialization can take a long time. This initialization can be done in a separate thread to prevent the initialization from interfering with the display of the applet. If an applet needs to process messages from the network, that work generally is done in a separate thread so that the applet can continue painting itself on the screen and responding to mouse and keyboard events. In addition, if each message is processed separately, the applet uses a separate thread for each message.

For all of the reasons there are to use threads, there are also some compelling reasons not to use them. If a program uses inherently sequential logic, where one operation starts another operation and then must wait for the other operation to complete before continuing, one thread can implement the entire sequence. Using multiple threads in such a case results in a more complex program with no accompanying benefits. There is considerable overhead in creating and starting a thread, so if an operation involves only a few primitive statements, it is faster to handle it with a single thread. This can even be true when the operation is conceptually asynchronous. When multiple threads share objects, the objects must use synchronization mechanisms to coordinate thread access and maintain consistent state. Synchronization mechanisms add complexity to a program, can be difficult to tune for optimal performance, and can be a source of bugs.

3.1 Using Thread Objects

The `Thread` class in the `java.lang` package creates and controls threads in Java programs. The execution of Java code is always under the control of a `Thread` object. The `Thread` class provides a `static` method called `currentThread()` that provides a reference to the `Thread` object that controls the current thread of execution.

Associating a Method with a Thread

The first thing you need to do to make a `Thread` object useful is to associate it with a method you want it to run. Java provides two ways of associating a method with a `Thread`:

- Declare a subclass of `Thread` that defines a `run()` method.

- Pass a reference to an object that implements the `Runnable` interface to a `Thread` constructor.

For example, if you need to load the contents of a URL as part of an applet's initialization, but the applet can provide other functionality before the content is loaded, you might want to load the content in a separate thread. Here is a class that does just that:

```
import java.net.URL;
```

```
class UrlData extends Thread   {
    private Object data;
    private URL url
    public UrlData(String urlName) throws MalformedURLException {
        url = new URL(urlName);
        start();
    }
    public void run(){
        try {
            data = url.getContent();
        } catch (java.io.IOException  e) {
        }
    }
    public Object getUrlData(){
        return data;
    }
}
```

The `UrlData` class is declared as a subclass of `Thread` so that it can get the contents of the URL in a separate thread. The constructor creates a `java.net.URL` object to fetch the contents of the URL, and then calls the `start()` method to start the thread. Once the thread is started, the constructor returns; it does not wait for the contents of the URL to be fetched. The `run()` method is executed after the thread is started; it does the real work of fetching the data. The `getUrlData()` method is an access method that returns the value of the `data` variable. The value of this variable is `null` until the contents of the URL have been fetched, at which time it contains a reference to the actual data.

Subclassing the `Thread` class is convenient when the method you want to run in a separate thread does not need to belong to a particular class. Sometimes, however, you need the method to be part of a particular class that is a subclass of a class other than `Thread`. Say, for example, you want a graphical object that is displayed in a window to alternate its background color between red and blue once a second. The object that implements this behavior needs to be a subclass of the `java.awt.Canvas` class. However, at the same time, you need a separate thread to alternate the color of the object once a second.

In this situation, you want to tell a `Thread` object to run code in another object that is not a subclass of the `Thread` class. You can accomplish this by passing a reference to an object that implements the `Runnable` interface to the constructor of the `Thread` class. The `Runnable` interface requires that an object has a `public` method called `run()` that takes no arguments. When a `Runnable` object is passed to the constructor of the `Thread` class, it creates a `Thread` object that calls the `Runnable` object's `run()` method when the thread is started. The following example shows part of the code that implements an object that alternates its background color between red and blue once a second:

```
class AutoColorChange extends java.awt.Canvas implements Runnable {
    private Thread myThread;
    AutoColorChange () {
        myThread = new Thread(this);
        myThread.start();
        ...
    }
    public void run() {
        while (true) {
            setBackground(java.awt.Color.red);
            repaint();
            try {
                myThread.sleep(1000);
            } catch (InterruptedException e) {}
            setBackground(java.awt.Color.blue);
            repaint();
            try {
                myThread.sleep(1000);
            } catch (InterruptedException e) {}
        }
    }
}
```

The `AutoChangeColor` class extends `java.awt.Canvas`, alternating the background color between red and blue once a second. The constructor creates a new `Thread` by passing the current object to the `Thread` constructor, which tells the `Thread` to call the `run()` method in the `AutoChangeColor` class. The constructor then starts the new thread by calling its `start()` method, so that the color change happens asynchronously of whatever else is going on. The class has an instance variable called `myThread` that contains a reference to the `Thread` object, so that can control the thread. The `run()` method takes care of changing the background color, using the `sleep()` method of the `Thread` class to temporarily suspend the thread and calling `repaint()` to redisplay the object after each color change.

Controlling a Thread

As shown in the previous section, you start a `Thread` by calling its `start()` method. Before the `start()` method is called, the `isAlive()` method of the `Thread` object always returns `false`. When the `start()` method is called, the `Thread` object becomes associated with a scheduled thread in the underlying environment. After the `start()` method has returned, the `isAlive()` method always returns `true`. The `Thread` is now scheduled to run until it dies, unless it is suspended or in another unrunnable state.

It is actually possible for `isAlive()` to return `true` before `start()` returns, but not before start() is called. This can happen because the start() method can return either before the started Thread begins to run or after it begins to run. In other words, the method that called start() and the new thread are now running concurrently. On a multiprocessor system, the start() method can even return at the same time the started Thread begins to run.

Thread objects have a parent-child relationship. The first thread created in a Java environment does not have a parent Thread. However, after the first Thread object is created, the Thread object that controls the thread used to create another Thread object is considered to be the parent of the newly created Thread. This parent-child relationship is used to supply some default values when a Thread object is created, but it has no further significance after a Thread has been created.

Stopping a thread

A thread dies when one of the following things happens:

- The run() method called by the Thread returns.
- An exception is thrown that causes the run() method to be exited.
- The stop() method of the Thread is called.

The stop() method of the Thread class works by throwing a ThreadDeath object in the run() method of the thread. Normally, you should not catch ThreadDeath objects in a try statement. If you need to catch ThreadDeath objects to detect that a Thread is about to die, the try statement that catches ThreadDeath objects should rethrow them.

When an object (ThreadDeath or otherwise) is thrown out of the run() method for the Thread, the uncaughtException() method of the ThreadGroup for that Thread is called. If the thrown object is an instance of the ThreadDeath class, the thread dies, and the thrown object is ignored. Otherwise, if the thrown object is of any other class, uncaughtException() calls the thrown object's printStackTrace() method, the thread dies, and the thrown object is ignored. In either case, if there are other nondaemon threads running in the system, the current program continues to run.

Interrupting a thread

There are a number of methods in the Java API, such as wait() and join(), that are declared as throwing an InterruptedException. What these methods have in common is that they temporarily suspend the execution of a thread. In Java 1.1, if a thread is waiting for one of these methods to return and another thread calls interrupt() on the waiting thread, the method that is waiting throws an InterruptedException.

The interrupt() method sets an internal flag in a Thread object. Before the interrupt()

method is called, the `isInterrupted()` method of the `Thread` object always returns `false`. After the `interrupt()` method is called, `isInterrupted()` returns `true`.

Prior to version 1.1, the methods in the Java API that are declared as throwing an `InterruptedException` do not actually do so. However, the `isInterrupted()` method does function as described above. Thus, if the code in the `run()` method for a thread periodically calls `isInterrupted()`, the thread can respond to a call to `interrupt()` by shutting down in an orderly fashion.

Thread priority

One of the attributes that controls the behavior of a thread is its priority. Although Java does not guarantee much about how threads are scheduled, it does guarantee that a thread with a priority that is higher than that of another thread will be scheduled to run at least as often, and possibly more often, than the thread with the lower priority. The priority of a thread is set when the `Thread` object is created, by passing an argument to the constructor that creates the `Thread` object. If an explicit priority is not specified, the `Thread` inherits the priority of its parent `Thread` object.

You can query the priority of a `Thread` object by calling its `getPriority()` method. Similarly, you can set the priority of a `Thread` using its `setPriority()` method. The priority you specify must be greater than or equal to `Thread.MIN_PRIORITY` and less than or equal to `Thread.MAX_PRIORITY`.

Before actually setting the priority of a `Thread` object, the `setPriority()` method checks the maximum allowable priority for the `ThreadGroup` that contains the `Thread` by calling `getMaxPriority()` on the `ThreadGroup`. If the call to `setPriority()` tries to set the priority to a value that is higher than the maximum allowable priority for the `ThreadGroup`, the priority is instead set to the maximum priority. It is possible for the current priority of a `Thread` to be greater than the maximum allowable priority for the `ThreadGroup`. In this case, an attempt to raise the priority of the `Thread` results in its priority being lowered to the maximum priority.

Daemon threads

A daemon thread is a thread that runs continuously to perform a service, without having any connection with the overall state of the program. For example, the thread that runs the garbage collector in Java is a daemon thread. The thread that processes mouse events for a Java program is also a daemon thread. In general, threads that run application code are not daemon threads, and threads that run system code are daemon threads. If a thread dies and there are no other threads except daemon threads alive, the Java virtual machine stops.

A `Thread` object has a `boolean` attribute that specifies whether or not a thread is a daemon thread. The daemon attribute of a thread is set when the `Thread` object is created, by passing an argument to the constructor that creates the `Thread` object. If the daemon attribute is not explicitly specified, the `Thread` inherits the daemon attribute of its parent `Thread` object.

The daemon attribute is queried using the `isDaemon()` method; it is set using the `setDaemon()` method.

Yielding

When a thread has nothing to do, it can call the `yield()` method of its `Thread` object. This method tells the scheduler to run a different thread. The value of calling `yield()` depends largely on whether the scheduling mechanism for the platform on which the program is running is preemptive or nonpreemptive.

By choosing a maximum length of time a thread can continuously, a *preemptive* scheduling mechanism guarantees that no single thread uses more than its fair share of the processor. If a thread runs for that amount of time without yielding control to another thread, the scheduler preempts the thread and causes it to stop running so that another thread can run.

A *nonpreemptive* scheduling mechanism cannot preempt threads. A nonpreemptive scheduler relies on the individual threads to yield control of the processor frequently, so that it can provide reasonable performance. A thread explicitly yields control by calling the `Thread` object's `yield()` method. More often, however, a thread implicitly yields control when it is forced to wait for something to happen elsewhere.

Calling a `Thread` object's `yield()` method during a lengthy computation can be quite valuable on a platform that uses a nonpreemptive scheduling mechanism, as it allows other threads to run. Otherwise, the lengthy computation can prevent other threads from running. On a platform that uses a preemptive scheduling mechanism, calling `yield()` does not usually make any noticeable difference in the responsiveness of threads.

Regardless of the scheduling algorithm that is being used, you should not make any assumptions about when a thread will be scheduled to run again after it has called `yield()`. If you want to prevent a thread from being scheduled to run until a specified amount of time has elapsed, you should call the `sleep()` method of the `Thread` object. The `sleep()` method takes an argument that specifies a minimum number of milliseconds that must elapse before the thread can be scheduled to run again.

Controlling groups of threads

Sometimes is it necessary to control multiple threads at the same time. Java provides the `ThreadGroup` class for this purpose. Every `Thread` object belongs to a `ThreadGroup` object. By passing an argument to the constructor that creates the `Thread` object, the `ThreadGroup` of a thread can be set when the `Thread` object is created. If an explicit `ThreadGroup` is not specified, the `Thread` belongs to the same ThreadGroup as its parent Thread object.

PREVIOUS
StringTokenizer

HOME
BOOK INDEX

NEXT
Synchronizing Multiple Threads

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

4. Exception Handling

Contents:

Exception handling is a mechanism that allows Java programs to handle various exceptional conditions, such as semantic violations of the language and program-defined errors, in a robust way. When an exceptional condition occurs, an *exception* is thrown. If the Java virtual machine or run-time environment detects a semantic violation, the virtual machine or run-time environment implicitly throws an exception. Alternately, a program can throw an exception explicitly using the `throw` statement. After an exception is thrown, control is transferred from the current point of execution to an appropriate `catch` clause of an enclosing `try` statement. The `catch` clause is called an exception handler because it handles the exception by taking whatever actions are necessary to recover from it.

4.1 Handling Exceptions

The `try` statement provides Java's exception-handling mechanism. A `try` statement contains a block of code to be executed. Putting a block in a `try` statement indicates that any exceptions or other abnormal exits in the block are going to be handled appropriately. A `try` statement can have any number of optional `catch` clauses that act as exception handlers for the `try` block. A `try` statement can also have a `finally` clause. The `finally` block is always executed before control leaves the `try` statement; it cleans up after the `try` block. Note that a `try` statement must have either a `catch` clause or a `finally` clause.

Here is an example of a `try` statement that includes a `catch` clause and a `finally` clause:

```
try {
    out.write(b);
} catch (IOException e) {
    System.out.println("Output Error");
} finally {
    out.close();
}
```

If `out.write()` throws an `IOException`, the exception is caught by the `catch` clause. Regardless of whether `out.write()` returns normally or throws an exception, the `finally` block is executed, which ensures that `out.close()` is always called.

A `try` statement executes the block that follows the keyword `try`. If an exception is thrown from within the `try` block and the `try` statement has any `catch` clauses, those clauses are searched, in order, for one that can handle the exception. If a `catch` clause handles an exception, that `catch` block is executed.

However, if the `try` statement does not have any `catch` clauses that can handle the exception (or does not have any `catch` clauses at all), the exception propagates up through enclosing statements in the current method. If the current method does not contain a `try` statement that can handle the exception, the exception propagates up to the invoking method. If this method does not contain an appropriate `try` statement, the exception propagates up again, and so on. Finally, if no `try` statement is found to handle the exception, the currently running thread terminates.

A `catch` clause is declared with a parameter that specifies the type of exception it can handle. The parameter in a `catch` clause must be of type `Throwable` or one of its subclasses. When an exception occurs, the `catch` clauses are searched for the first one with a parameter that matches the type of the exception thrown or is a superclass of the thrown exception. When the appropriate `catch` block is executed, the actual exception object is passed as an argument to the `catch` block. The code within a `catch` block should do whatever is necessary to handle the exception condition.

The `finally` clause of a `try` statement is always executed, no matter how control leaves the `try` statement. Thus it is a good place to handle clean-up operations, such as closing files, freeing resources, and closing network connections.

6. I/O

Contents:

The `java.io` package contains the fundamental classes for performing input and output operations in Java. These I/O classes can be divided into four basic groups:

- Classes for reading input from a stream.
- Classes for writing output to a stream.
- Classes for manipulating files.
- Classes for serializing objects.

All fundamental I/O in Java is based on *streams*. A stream represents a flow of data, or a channel of communication. Conceptually, there is a reading process at one end of the stream and a writing process at the other end. Java 1.0 supported only byte streams, which meant that Unicode characters were not always handled correctly. As of Java 1.1, there are classes in `java.io` for both byte streams and character streams. The character stream classes, which are called *readers* and *writers*, handle Unicode characters appropriately.

The rest of this chapter describes the classes in `java.io` that read from and write to streams, as well as the classes that manipulate files. The classes for serializing objects are described in Chapter 7, *Object Serialization*.

6.1 Input Streams and Readers

The `InputStream` class is an `abstract` class that defines methods to read sequentially from a stream of bytes. Java provides subclasses of the `InputStream` class for reading from files, `StringBuffer` objects, and byte arrays, among other things. Other subclasses of `InputStream` can be chained together to provide additional logic, such as keeping track of the current line number or combining multiple input sources into one logical input stream. It is also easy to define a subclass of `InputStream` that reads from any other kind of data source.

In Java 1.1, the `Reader` class is an `abstract` class that defines methods to read sequentially from a stream of characters. Many of the byte-oriented `InputStream` subclasses have character-based `Reader` counterparts. Thus, there are subclasses of `Reader` for reading from files, character arrays, and `String` objects.

InputStream

The `InputStream` class is the `abstract` superclass of all other byte input stream classes. It defines three `read()` methods for reading from a raw stream of bytes:

```
read()
read(byte[] b)
read(byte[] b, int off, int len)
```

If there is no data available to read, these methods block until input is available. The class also defines an `available()` method that returns the number of bytes that can be read without blocking and a `skip()` method that skips ahead a specified number of bytes. The `InputStream` class defines a mechanism for marking a position in the stream and returning to it later, via the `mark()` and `reset()` methods. The `markSupported()` method returns `true` in subclasses that support these methods.

Because the `InputStream` class is `abstract`, you cannot create a "pure" `InputStream`. However, the various subclasses of `InputStream` can be used interchangeably. For example, methods often take an `InputStream` as a parameter. Such a method accepts any subclass of `InputStream` as an argument.

`InputStream` is designed so that `read(byte[])` and `read(byte[], int, int)` both call `read()`. Thus, when you subclass `InputStream`, you only need to define the `read()` method. However, for efficiency's sake, you should also override `read(byte[], int, int)` with a method that can read a block of data more efficiently than reading each byte separately.

Reader

The `Reader` class is the `abstract` superclass of all other character input stream classes. It defines nearly the same methods as `InputStream`, except that the `read()` methods deal with characters instead of bytes:

```
read()
read(char[] cbuf)
read(char[] cbuf, int off, int len)
```

The `available()` method of `InputStream` has been replaced by the `ready()` method of `Reader`, which simply returns a flag that indicates whether or not the stream must block to read the next character.

`Reader` is designed so that `read()` and `read(char[])` both call `read(char[], int, int)`. Thus, when you subclass `Reader`, you only need to define the `read(char[], int, int)` method. Note that this design is different from, and more efficient than, that of `InputStream`.

InputStreamReader

The InputStreamReader class serves as a bridge between InputStream objects and Reader objects. Although an InputStreamReader acts like a character stream, it gets its input from an underlying byte stream and uses a character encoding scheme to translate bytes into characters. When you create an InputStreamReader, specify the underlying InputStream and, optionally, the name of an encoding scheme. For example, the following code fragment creates an InputStreamReader that reads characters from a file that is encoded using the ISO 8859-5 encoding:

```
String fileName = "encodedfile.txt"; String encodingName = "8859_5";
InputStreamReader in;
try {
   x FileInputStream fileIn = new FileInputStream(fileName);
    in = new InputStreamReader(fileIn, encodingName);
} catch (UnsupportedEncodingException e1) {
    System.out.println(encodingName + " is not a supported encoding scheme.");
} catch (IOException e2) {
    System.out.println("The file " + fileName + " could not be opened.");
}
```

FileInputStream and FileReader

The FileInputStream class is a subclass of InputStream that allows a stream of bytes to be read from a file. The FileInputStream class has no explicit open method. Instead, the file is implicitly opened, if appropriate, when the FileInputStream is created. There are three ways to create a FileInputStream:

- You can create a FileInputStream by passing the name of a file to be read:

  ```
  FileInputStream f1 = new FileInputStream("foo.txt");
  ```

- You can create a FileInputStream with a File object:

  ```
  File f = new File("foo.txt");
  FileInputStream f2 = new FileInputStream(f);
  ```

- You can create a FileInputStream with a FileDescriptor object. A FileDescriptor object encapsulates the native operating system's representation of an open file. You can get a FileDescriptor from a RandomAccessFile by calling its getFD() method. You create a FileInputStream that reads from the open file associated with a RandomAccessFile as follows:

  ```
  RandomAccessFile raf;
  raf = new RandomAccessFile("z.txt","r");
  FileInputStream f3 = new FileInputStream(raf.getFD());
  ```

  The FileReader class is a subclass of Reader that reads a stream of characters from a file. The bytes in the file are converted to characters using the default character encoding scheme. If you do not want to use the default encoding scheme, you need to wrap an InputStreamReader around a FileInputStream, as shown above. You can create a FileReader from a filename, a File object, or a FileDescriptor object, as described above for `FileInputStream`.

StringReader and StringBufferInputStream

The `StringReader` class is a subclass of `Reader` that gets its input from a `String` object. The `StringReader` class supports mark-and-reset functionality via the `mark()` and `reset()` methods. The following example shows the use of `StringReader`:

```
StringReader sr = new StringReader("abcdefg");
try {
    char[] buffer = new char[3];
    sr.read(buffer);
    System.out.println(buffer);
} catch (IOException e) {
    System.out.println("There was an error while reading.");
}
```

This code fragment produces the following output:

```
abc
```

The `StringBufferInputStream` class is the byte-based relative of `StringReader`. The entire class is deprecated as of Java 1.1 because it does not properly convert the characters of the string to a byte stream; it simply chops off the high eight bits of each character. Although the `markSupported()` method of `StringBufferInputStream` returns `false`, the `reset()` method causes the next read operation to read from the beginning of the `String`.

CharArrayReader and ByteArrayInputStream

The `CharArrayReader` class is a subclass of `Reader` that reads a stream of characters from an array of characters. The `CharArrayReader` class supports mark-and-reset functionality via the `mark()` and `reset()` methods. You can create a `CharArrayReader` by passing a reference to a `char` array to a constructor like this:

```
char[] c;
...
CharArrayReader r;
r = new CharArrayReader(c);
```

You can also create a `CharArrayReader` that only reads from part of an array of characters by passing an offset and a length to the constructor. For example, to create a `CharArrayReader` that reads elements 5 through 24 of a `char` array you would write:

```
r = new CharArrayReader(c, 5, 20);
```

The `ByteArrayInputStream` class is just like `CharArrayReader`, except that it deals with bytes instead of characters. In Java 1.0, `ByteArrayInputStream` did not fully support `mark()` and `reset()`; in Java 1.1

these methods are completely supported.

PipedInputStream and PipedReader

The PipedInputStream class is a subclass of InputStream that facilitates communication between threads. Because it reads bytes written by a connected PipedOutputStream, a PipedInputStream must be connected to a PipedOutputStream to be useful. There are a few ways to connect a PipedInputStream to a PipedOutputStream. You can first create the PipedOutputStream and pass it to the PipedInputStream constructor like this:

```
PipedOutputStream po = new PipedOutputStream();
PipedInputStream pi = new PipedInputStream(po);
```

You can also create the PipedInputStream first and pass it to the PipedOutputStream constructor like this:

```
PipedInputStream pi = new PipedInputStream();
PipedOutputStream po = new PipedOutputStream(pi);
```

The PipedInputStream and PipedOutputStream classes each have a connect() method you can use to explicitly connect a PipedInputStream and a PipedOutputStream as follows:

```
PipedInputStream pi = new PipedInputStream();
PipedOutputStream po = new PipedOutputStream();
pi.connect(po);
```

Or you can use connect() as follows:

```
PipedInputStream pi = new PipedInputStream();
PipedOutputStream po = new PipedOutputStream();
po.connect(pi);
```

Multiple PipedOutputStream objects can be connected to a single PipedInputStream at one time, but the results are unpredictable. If you connect a PipedOutputStream to an already connected PipedInputStream, any unread bytes from the previously connected PipedOutputStream are lost. Once the two PipedOutputStream objects are connected, the PipedInputStream reads bytes written by either PipedOutputStream in the order that it receives them. The scheduling of different threads may vary from one execution of the program to the next, so the order in which the PipedInputStream receives bytes from multiple PipedOutputStream objects can be inconsistent.

The PipedReader class is the character-based equivalent of PipedInputStream. It works in the same way, except that a PipedReader is connected to a PipedWriter to complete the pipe, using either the appropriate constructor or the connect() method.

FilterInputStream and FilterReader

The FilterInputStream class is a wrapper class for InputStream objects. Conceptually, an object that belongs to a subclass of FilterInputStream is wrapped around another InputStream object. The constructor for this class requires an InputStream. The constructor sets the object's in instance variable to reference the specified InputStream, so from that point on, the FilterInputStream is associated with the given InputStream. All of the methods in FilterInputStream work by calling the corresponding methods in the underlying InputStream. Because the close() method of a FilterInputStream calls the close() method of the InputStream that it wraps, you do not need to explicitly close the underlying InputStream.

A FilterInputStream does not add any functionality to the object that it wraps, so by itself it is not very useful. However, subclasses of the FilterInputStream class do add functionality to the objects that they wrap in two ways:

- Some subclasses add logic to the InputStream methods. For example, the InflaterInputStream class in the java.util.zip package decompresses data automatically in the read() methods.

- Some subclasses add new methods. An example is DataInputStream, which provides methods for reading primitive Java data types from the stream.

The FilterReader class is the character-based equivalent of FilterInputStream. A FilterReader is wrapped around an underlying Reader object; the methods of FilterReader call the corresponding methods of the underlying Reader. However, unlike FilterInputStream, FilterReader is an abstract class, so you cannot instantiate it directly.

DataInputStream

The DataInputStream class is a subclass of FilterInputStream that provides methods for reading a variety of data types. The DataInputStream class implements the DataInput interface, so it defines methods for reading all of the primitive Java data types.

You create a DataInputStream by passing a reference to an underlying InputStream to the constructor. Here is an example that creates a DataInputStream and uses it to read an int that represents the length of an array and then to read the array of long values:

```
long[] readLongArray(InputStream in) throws IOException {
    DataInputStream din = new DataInputStream(in);
    int count = din.readInt();
    long[] a = new long[count];
    for (int i = 0; i < count; i++) {
        a[i] = din.readLong();
    }
    return a;
}
```

BufferedReader and BufferedInputStream

The BufferedReader class is a subclass of Reader that buffers input from an underlying Reader. A BufferedReader object reads enough characters from its underlying Reader to fill a relatively large buffer, and then it satisfies read operations by supplying characters that are already in the buffer. If most read operations read just a few characters, using a BufferedReader can improve performance because it reduces the number of read operations that the program asks the operating system to perform. There is generally a measurable overhead associated with each call to the operating system, so reducing the number of calls into the operating system improves performance. The BufferedReader class supports mark-and-reset functionality via the mark() and reset() methods.

Here is an example that shows how to create a BufferedReader to improve the efficiency of reading from a file:

```
try {
    FileReader fileIn = new FileReader("data.dat");
    BufferedReader in = new BufferedReader(fileIn);
    // read from the file
} catch (IOException e) {
    System.out.println(e);
}
```

The BufferedInputStream class is the byte-based counterpart of BufferedReader. It works in the same way as BufferedReader, except that it buffers input from an underlying InputStream.

LineNumberReader and LineNumberInputStream

The LineNumberReader class is a subclass of BufferedReader. Its read() methods contain additional logic to count end-of-line characters and thereby maintain a line number. Since different platforms use different characters to represent the end of a line, LineNumberReader takes a flexible approach and recognizes "\n", "\r", or "\r\n" as the end of a line. Regardless of the end-of-line character it reads, LineNumberReader returns only "\n" from its read() methods.

You can create a LineNumberReader by passing its constructor a Reader. The following example prints out the first five lines of a file, with each line prefixed by its number. If you try this example, you'll see that the line numbers begin at 0 by default:

```
try {
    FileReader fileIn = new FileReader("text.txt");
    LineNumberReader in = new LineNumberReader(fileIn);
    for (int i = 0; i < 5; i++)
        System.out.println(in.getLineNumber() + " " + in.readLine());
}catch (IOException e) {
    System.out.println(e);
}
```

The LineNumberReader class has two methods pertaining to line numbers. The getLineNumber() method returns the current line number. If you want to change the current line number of a LineNumberReader, use setLineNumber(). This method does not affect the stream position; it merely sets the value of the line number.

The `LineNumberInputStream` is the byte-based equivalent of `LineNumberReader`. The entire class is deprecated in Java 1.1 because it does not convert bytes to characters properly. Apart from the conversion problem, `LineNumberInputStream` works the same as `LineNumberReader`, except that it takes its input from an `InputStream` instead of a `Reader`.

SequenceInputStream

The `SequenceInputStream` class is used to sequence together multiple `InputStream` objects. Consider this example:

```
FileInputStream f1 = new FileInputStream("data1.dat");
FileInputStream f2 = new FileInputStream("data2.dat");
SequenceInputStream s = new SequenceInputStream(f1, f2);
```

This example creates a `SequenceInputStream` that reads all of the bytes from `f1` and then reads all of the bytes from `f2` before reporting that it has encountered the end of the stream. You can also cascade `SequenceInputStream` object themselves, to allow more than two input streams to be read as if they were one. You would write it like this:

```
FileInputStream f3 = new FileInputStream("data3.dat");
SequenceInputStream s2 = new SequenceInputStream(s, f3);
```

The `SequenceInputStream` class has one other constructor that may be more appropriate for wrapping more than two `InputStream` objects together. It takes an `Enumeration` of `InputStream` objects as its argument. The following example shows how to create a `SequenceInputStream` in this manner:

```
Vector v = new Vector();
v.add(new FileInputStream("data1.dat"));
v.add(new FileInputStream("data2.dat"));
v.add(new FileInputStream("data3.dat"));
Enumeration e = v.elements();
SequenceInputStream s = new SequenceInputStream(e);
```

PushbackInputStream and PushbackReader

The `PushbackInputStream` class is a `FilterInputStream` that allows data to be pushed back into the input stream and reread by the next read operation. This functionality is useful for implementing things like parsers that need to read data and then return it to the input stream. The Java 1.0 version of `PushbackInputStream` supported only a one-byte pushback buffer; in Java 1.1 this class has been enhanced to support a larger pushback buffer.

To create a `PushbackInputStream`, pass an `InputStream` to its constructor like this:

```
FileInputStream ef = new FileInputStream("expr.txt");
PushbackInputStream pb = new PushbackInputStream(ef);
```

This constructor creates a `PushbackInputStream` that uses a default one-byte pushback buffer. When you have data that you want to push back into the input stream to be read by the next read operation, you pass the data to one of the `unread()` methods.

The `PushbackReader` class is the character-based equivalent of `PushbackInputStream`. In the following example, we create a `PushbackReader` with a pushback buffer of 48 characters:

```
FileReader fileIn = new FileReader("expr.txt");
PushbackReader in = new PushbackReader(fileIn, 48);
```

Here is an example that shows the use of a `PushbackReader`:

```
public String readDigits(PushbackReader pb) {
    char c;
    StringBuffer buffer = new StringBuffer();
    try {
        while (true) {
            c = (char)pb.read();
            if (!Character.isDigit(c))
                break;
            buffer.append(c);
        }
        if (c != -1)
            pb.unread(c);
    }catch (IOException e) {}
    return buffer.toString();
}
```

The above example shows a method that reads characters corresponding to digits from a `PushbackReader`. When it reads a character that is not a digit, it calls the `unread()` method so that the nondigit can be read by the next read operation. It then returns a string that contains the digits that were read.

PREVIOUS	HOME	NEXT
Hashtables	BOOK INDEX	Output Streams and Writers

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

PREVIOUS Chapter 7 NEXT

7. Object Serialization

Contents:

The object serialization mechanism in Java 1.1 provides a way for objects to be written as a stream of bytes and then later recreated from that stream of bytes. This facility supports a variety of interesting applications. For example, object serialization provides persistent storage for objects, whereby objects are stored in a file for later use. Also, a copy of an object can be sent through a socket to another Java program. Object serialization forms the basis for the remote method invocation mechanism in Java that facilitates distributed programs. Object serialization is supported by a number of new classes in the `java.io` package in Java 1.1.

7.1 Object Serialization Basics

If a class is designed to work with object serialization, reading and writing instances of that class is quite simple. The process of writing an object to a byte stream is called *serialization*. For example, here is how you can write a `Color` object to a file:

```
FileOutputStream out = new FileOutputStream("tmp");
ObjectOutput objOut = new ObjectOutputStream(out);
objOut.writeObject(Color.red);
```

All you need to do is create an `ObjectOutputStream` around another output stream and then pass the object to be written to the `writeObject()` method. If you are writing objects to a socket or any other destination that is time-sensitive, you should call the `flush()` method after you are finished passing objects to the `ObjectOutputStream`.

The process of reading an object from byte stream is called *deserialization*. Here is how you can read that Color object from its file:

```
FileInputStream in = new FileInputStream("tmp");
ObjectInputStream objIn = new ObjectInputStream(in);
Color c = (Color)objIn.readObject();
```

Here all you need to do is create an ObjectInputStream object around another input stream and call its readObject() method.

PREVIOUS | HOME | NEXT
File Manipulation | BOOK INDEX | Writing Classes to Work with Serialization

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

8. Networking

Contents:

The `java.net` package provides two basic mechanisms for accessing data and other resources over a network. The fundamental mechanism is called a socket. A socket allows programs to exchange groups of bytes called packets. There are a number of classes in `java.net` that support sockets, including `Socket`, `ServerSocket`, `DatagramSocket`, `DatagramPacket`, and `MulticastSocket`. The `java.net` package also includes a `URL` class that provides a higher-level mechanism for accessing and processing data over a network.

8.1 Sockets

A socket is a mechanism that allows programs to send packets of bytes to each other. The programs do not need to be running on the same machine, but if they are running on different machines, they do need to be connected to a network that allows the machines to exchange data. Java's socket implementation is based on the socket library that was originally part of BSD UNIX. Programmers who are familiar with UNIX sockets or the Microsoft WinSock library should be able to see the similarities in the Java implementation.

When a program creates a socket, an identifying number called a port number is associated with the socket. Depending on how the socket is used, the port number is either specified by the program or assigned by the operating system. When a socket sends a packet, the packet is accompanied by two pieces of information that specify the destination of the packet:

- A network address that specifies the system that should receive the packet.
- A port number that tells the receiving system to which socket to deliver the data.

Sockets typically work in pairs, where one socket acts as a client and the other functions as a server. A server socket specifies the port number for the network communication and then listens for data that is sent to it by client sockets. The port numbers for server sockets are well-known numbers that are known to client programs.

For example, an FTP server uses a socket that listens at port 21. If a client program wants to communicate with an FTP server, it knows to contact a socket that listens at port 21.

The operating system normally specifies port numbers for client sockets because the choice of a port number is not usually important. When a client socket sends a packet to a server socket, the packet is accompanied by the port number of the client socket and the client's network address. The server is then able to use that information to respond to the client.

When using sockets, you have to decide which type of protocol that you want it to use to transport packets over the network: a connection-oriented protocol or a connectionless protocol. With a connection-oriented protocol, a client socket establishes a connection to a server socket when it is created. Once the connection has been established, a connection-oriented protocol ensures that data is delivered reliably, which means:

- For every packet that is sent, the packet is delivered. Every time a socket sends a packet, it expects to receive an acknowledgement that the packet has been received successfully. If the socket does not receive that acknowledgement within the time it expects to receive it, the socket sends the packet again. The socket keeps trying until transmission is successful, or it decides that delivery has become impossible.

- Packets are read from the receiving socket in the same order that they were sent. Because of the way that networks work, packets may arrive at the receiving socket in a different order than they were sent. A reliable, connection-oriented protocol allows the receiving socket to reorder the packets it receives, so that they can be read by the receiving program in the same order that they were sent.

A connectionless protocol allows a best-effort delivery of packets. It does not guarantee that packets are delivered or that packets are read by the receiving program in the same order they were sent. A connectionless protocol trades these deficiencies for performance advantages over connection-oriented protocols. Here are two types of situations in which connectionless protocols are frequently preferred over connection-oriented protocols:

- When only a single packet needs to be sent and guaranteed delivery is not crucial, a connectionless protocol eliminates the overhead involved in creating and destroying a connection. For comparison purposes, the connection-oriented TCP/IP protocol uses seven packets to send a single packet, while the connectionless UDP/IP protocol uses only one. A protocol for getting the current time typically uses a connectionless protocol to request the current time from the server and to return the time to the requester.

- For extremely time-sensitive applications, such as sending audio in real time, the guarantee of reliable transmission is not an advantage and may be a disadvantage. Pausing until a missing piece of data is received can cause noticeable clicks or pauses in the audio. Techniques for sending audio over a network that use a connectionless protocol have been developed and they work noticeably better. For example, RealAudio uses a protocol that runs on top of a connectionless protocol to transmit sound over a network.

Table 8.1 shows the roles of the various socket classes in the `java.net` package.

Table 8.1: Socket Classes in java.net

	Client	**Server**
Connection-oriented Protocol	`Socket`	`ServerSocket`
Connectionless Protocol	`DatagramSocket`	`DatagramSocket`

As of Java 1.1, the `java.net` package also contains a `MulticastSocket` class that supports connectionless, multicast data communication.

Sockets for Connection-Oriented Protocols

When you are writing code that implements the server side of a connection-oriented protocol, your code typically follows this pattern:

- Create a `ServerSocket` object to accept connections.
- When the `ServerSocket` accepts a connection, it creates a `Socket` object that encapsulates the connection.
- The `Socket` is asked to create `InputStream` and `OutputStream` objects that read and write bytes to and from the connection.
- The `ServerSocket` can optionally create a new thread for each connection, so that the server can listen for new connections while it is communicating with clients.

The code that implements the client side of a connection-oriented protocol is quite simple. It creates a `Socket` object that opens a connection with a server, and then it uses that `Socket` object to communicate with the server.

Now let's look at an example. The example consists of a pair of programs that allows a client to get the contents of a file from a server. The client requests the contents of a file by opening a connection to the server and sending it the name of a file followed by a newline character. If the server is able to read the named file, it responds by sending the string `"Good:\n"` followed by the contents of the file. If the server is not able to read the named file, it responds by sending the string `"Bad:"` followed by the name of the file and a newline character. After the server has sent its response, it closes the connection.

Here's the program that implements the server side of this file transfer:

```
public class FileServer extends Thread {
    public static void main(String[] argv) {
        ServerSocket s;
        try {
```

```
        s = new ServerSocket(1234, 10);
    }catch (IOException e) {
        System.err.println("Unable to create socket");
        e.printStackTrace();
        return;
    }
    try {
        while (true) {
            new FileServer(s.accept());
        }
    }catch (IOException e) {
    }
}
private Socket socket;
FileServer(Socket s) {
    socket = s;
    start();
}
public void run() {
    InputStream in;
    String fileName = "";
    PrintStream out = null;
    FileInputStream f;
    try {
        in = socket.getInputStream();
        out = new PrintStream(socket.getOutputStream());
        fileName = new DataInputStream(in).readLine();
        f = new FileInputStream(fileName);
    }catch (IOException e) {
        if (out != null)
          out.print("Bad:"+fileName+"\n");
        out.close();
        try {
            socket.close();
        }catch (IOException ie) {
        }
        return;
    }
    out.print("Good:\n");
    // send contents of file to client.
    byte[] buffer = new byte[4096];
    try {
        int len;
        while ((len = f.read(buffer)) > 0) {
            out.write(buffer, 0, len);
        }// while
        }catch (IOException e) {
        }finally {
            try {
                in.close();
                out.close();
                socket.close();
            }catch (IOException e) {
            }
        }
    }
}
```

The `FileServer` class implements the server side of the file transfer; it is a subclass of `Thread` to make it easier to write code that can handle multiple connections at the same time. The `main()` method provides the top-level logic for the program. The first thing that `main()` does is to create a `ServerSocket` object to listen for connections. The constructor for `ServerSocket` takes two parameters: the port number for the socket and a value that specifies the maximum length of the pending connections queue. The operating system can accept connections on behalf of the socket when the server program is busy doing something other than accepting connections. If the second parameter is greater than zero, the operating system can accept up to that many connections on behalf of the socket and store them in a queue. If the second parameter is zero, however, the operating system does not accept any connections on behalf of the server program. The remainder of the `main()` method accepts a connection, creates a new instance of the `FileServer` class to process the connection, and then waits for the next connection.

Each `FileServer` object is responsible for handling a connection accepted by its `main()` method. A `FileServer` object uses a `private` variable, `socket`, to refer to the `Socket` object that allows it to communicate with the client program on the other end of the connection. The constructor for `FileServer` sets its `socket` variable to refer to the `Socket` object that is passed to it by the `main()` method and then calls its `start()` method. The `FileServer` class inherits the `start()` method from the `Thread` class; the `start()` method starts a new thread that calls the `run()` method. Because the rest of the connection processing is done asynchronously in a separate thread, the constructor can return immediately. This allows the `main()` method to accept another connection right away, instead of having to wait for this connection to be fully processed before accepting another.

The `run()` method uses the `in` and `out` variables to refer to `InputStream` and `PrintStream` objects that read from and write to the connection associated with the `Socket` object, respectively. These streams are created by calling the `getInputStream()` and `getOutputStream()` methods of the `Socket` object. The `run()` method then reads the name of the file that the client program wants to receive and creates a `FileInputStream` to read that file. If any of the methods called up to this point have detected a problem, they throw some kind of `IOException`. In this case, the server sends a response to the client that consists of the string `"Bad:"` followed by the filename and then closes the socket and returns, which kills the thread.

If everything up to this point has been fine, the server sends the string `"Good:"` and then copies the contents of the file to the socket. The copying is done by repeatedly filling a buffer with bytes from the file and writing the buffer to the socket. When the contents of the file are exhausted, the streams and the socket are closed, the `run()` method returns, and the thread dies.

Now let's take a look at the client part of this program:

```
public class FileClient {
    private static boolean usageOk(String[] argv) {
        if (argv.length != 2) {
            String msg = "usage is: " + "FileClient server-name file-name";
            System.out.println(msg);
            return false;
        }
        return true;
    }
    public static void main(String[] argv) {
        int exitCode = 0;
        if (!usageOk(argv))
          return;
        Socket s = null;
        try {
            s = new Socket(argv[0], 1234);
        }catch (IOException e) {
            String msg = "Unable to connect to server";
            System.err.println(msg);
            e.printStackTrace();
            System.exit(1);
        }
        InputStream in = null;
        try {
            OutputStream out = s.getOutputStream();
            new PrintStream(out).print(argv[1]+"\n");
            in = s.getInputStream();
            DataInputStream din = new DataInputStream(in);
            String serverStatus = din.readLine();
            if (serverStatus.startsWith("Bad")) {
                exitCode = 1;
            int ch;
            while((ch = in.read()) >= 0) {
                System.out.write((char)ch);
            }// while
        }catch (IOException e) {
        }finally {
            try {
                s.close();
            }catch (IOException e) {
            }
        }
    }
}
```

The `usageOk()` method is simply a utility method that verifies that the correct number of arguments have been passed to the client application. It outputs a help message if the number of arguments is not what is expected. It is generally a good idea to include a method like this in a Java application that uses command-line parameters.

The `main()` method does the real work of `FileClient`. After it verifies that it has the correct number of parameters, it attempts to create a socket connected to the server program running on the specified host and listening for connections on port number 1234. The socket that it creates is encapsulated by a `Socket` object. The constructor for the `Socket` object takes two arguments: the name of the machine the server program is running on and the port number. After the socket is successfully opened, the client sends the specified filename, followed by a new line character, to the server. The client then gets an `InputStream` from the socket to read what the server is sending and reads the success/failure code that the server sends back. If the request is a success, the client reads the contents of the requested file.

Note that the `finally` clause at the end closes the socket. If the program did not explicitly close the socket, it would be closed automatically when the program terminates. However, it is a good programming practice to explicitly close a socket when you are done with it.

Sockets for Connectionless Protocols

Communicating with a connectionless protocol is simpler than using a connection-oriented protocol, as both the client and the server use `DatagramSocket` objects. The code for the server-side program has the following pattern:

- Create a `DatagramSocket` object associated with a specified port number.
- Create a `DatagramPacket` object and ask the `DatagramSocket` to put the next piece of data it receives in the `DatagramPacket`.

On the client-side, the order is simply reversed:

- Create a `DatagramPacket` object associated with a piece of data, a destination network address, and a port number.
- Ask a `DatagramSocket` object to send the data associated with the `DatagramPacket` to the destination associated with the `DatagramSocket`.

Let's look at an example that shows how this pattern can be coded into a server that provides the current time and a client that requests the current time. Here's the code for the server class:

```
public class TimeServer {
    static DatagramSocket socket;
    public static void main(String[] argv) {
        try {
            socket = new DatagramSocket(7654);
        }catch (SocketException e) {
            System.err.println("Unable to create socket");
            e.printStackTrace();
            System.exit(1);
        }
        DatagramPacket datagram;
        datagram = new DatagramPacket(new byte[1], 1);
        while (true) {
            try {
                socket.receive(datagram);
                respond(datagram);
            }catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
    static void respond(DatagramPacket request) {
        ByteArrayOutputStream bs;
        bs = new ByteArrayOutputStream();
        DataOutputStream ds = new DataOutputStream(bs);
        try {
            ds.writeLong(System.currentTimeMillis());
        }catch (IOException e) {
        }
        DatagramPacket response;
        byte[] data = bs.toByteArray();
        response = new DatagramPacket(data, data.length,
                              request.getAddress(), request.getPort());
        try {
            socket.send(response);
        }catch (IOException e) {
            // Give up, we've done our best.
        }
    }
}
```

The `main()` method of the `TimeServer` class begins by creating a `DatagramSocket` object that uses port number 7654. The `socket` variable refers to this `DatagramSocket`, which is used to communicate with clients. Then the `main()` method creates a `DatagramPacket` object to contain data received by the `DatagramSocket`. The two-argument constructor for `DatagramPacket` creates objects that receive data. The first argument is an array of bytes to contain the data, while the second argument specifies the number of bytes to read. When a `DatagramSocket` is asked to receive a packet into a `DatagramPacket`, only the specified number of bytes are read. Even though the client is not really sending any information to the server, we still create a `DatagramPacket` with a 1-byte buffer. In theory, all that the server needs is an empty packet that specifies the client's network address and port number, but attempting to receive a zero-byte packet does not work. When the `receive()` method of a `DatagramSocket` is called to receive a zero-byte packet, it returns immediately, rather than waiting for a packet to arrive. Finally, the server enters an infinite loop that receives requests from clients using the `receive()` method of the `DatagramSocket`, and sends responses.

The `respond()` method handles sending responses. It starts by writing the current time as a `long` value to an array of bytes. Next, the `respond()` method prepares to send the array of bytes by creating a `DatagramPacket` object that encapsulates the array and the address and port number of the client that requested the time. Notice that the constructor used to create a `DatagramPacket` object for sending a packet takes four arguments: an array of bytes, the number of bytes to send, the client's network address, and the client's port number. The address and port are retrieved from the request `DatagramPacket` with the `getAddress()` and `getPort()` methods. The `respond()` method finishes its work by actually sending the `DatagramPacket` using the `send()` method of the `DatagramSocket`.

Now here's the code for the corresponding client program:

```
public class TimeClient {
    private static boolean usageOk(String[] argv) {
        if (argv.length != 1) {
            String msg = "usage is: " + "TimeClient server-name";
            System.out.println(msg);
            return false;
        }
        return true;
    }
    public static void main(String[] argv) {
        if (!usageOk(argv))
            System.exit(1);
        DatagramSocket socket;
        try {
            socket = new DatagramSocket();
        }catch (SocketException e) {
            System.err.println("Unable to create socket");
            e.printStackTrace();
            System.exit(1);
            return;
        }
        long time;
        try {
            byte[] buf = new byte[1];
            socket.send(new DatagramPacket(buf, 1,
                               InetAddress.getByName(argv[0]), 7654));
            DatagramPacket response = new DatagramPacket(new byte[8],8);
            socket.receive(response);
            ByteArrayInputStream bs;
            bs = new ByteArrayInputStream(response.getData());
            DataInputStream ds = new DataInputStream(bs);
            time = ds.readLong();
        }catch (IOException e) {
            e.printStackTrace();
            System.exit(1);
            return;
        }
        System.out.println(new Date(time));
        socket.close();
    }
}
```

The `main()` method does the real work of `TimeClient`. After it verifies that it has the correct number of parameters with `usageOk()`, it creates a `DatagramSocket` object for communicating with the server. Note that the constructor for this `DatagramSocket` does not specify any parameters; a client `DatagramSocket` is not explicitly connected to a specific port. Then the `main()` method creates a `DatagramPacket` object to contain the request to be sent to the server. Since this `DatagramPacket` is being used to send a packet, the code uses the four-argument constructor that specifies an array of bytes, the number of bytes to send, the specified network address for a time server, and the server's port number. The `DatagramPacket` is then sent to the server with the `send()` method of the `DatagramSocket`.

Now the `main()` method creates another `DatagramPacket` to receive the response from the server. The two-argument constructor is used this time because the object is being created to receive data. After calling the `receive()` method of the `DatagramSocket` to get the response from the server, the `main()` method gets the data from the response `DatagramPacket` by calling `getData()`. The data is wrapped in a `DataInputStream` so that the data can be read as a `long` value. If everything has gone smoothly, the client finishes by printing the current time and closing the socket.

PREVIOUS
Versioning of Classes

HOME
BOOK INDEX

NEXT
URL Objects

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

9. Security

Contents:

Java uses a "sandbox" security model to ensure that applets cannot cause security problems. The idea is that an applet can do whatever it wants within the constraints of its sandbox, but that nothing done inside the sandbox has any consequences outside of the sandbox.

9.1 SecurityManager

Java implements the sandbox model using the `java.lang.SecurityManager` class. An instance of `SecurityManager` is passed to the method `System.setSecurityManager()` to establish the security policy for an application. Before `setSecurityManager()` is called, a Java program can access any resources available on the system. After `setSecurityManager()` is called, however, the `SecurityManager` object is responsible for providing a security policy. Once a security policy has been set by calling `setSecurityManager`, the method cannot be called again. Subsequent calls simply throw a `SecurityException`.

All methods in the Java API that can access resources outside of the Java environment call a `SecurityManager` method to ask permission before doing anything. If the `SecurityManager` method throws a `SecurityException`, the exception is thrown out of the calling method, and access to the resource is denied. The `SecurityManager` class defines a number of methods for asking for permission to access specific resources. Each of these methods has a name that begins with the word "check." Table 9.1 shows the names of the `check` methods provided by the `SecurityManager` class.

Table 9.1: The Check Methods of SecurityManager

Method Name	**Permission**

`checkAccept()`	To accept a network connection
`checkAccess()`	To modify a `Thread` or `ThreadGroup`
`checkAwtEventQueueAccess()`	To access the AWT event queue
`checkConnect()`	To establish a network connection or send a datagram
`checkCreateClassLoader()`	To create a `ClassLoader` object
`checkDelete()`	To delete a file
`checkExec()`	To call an external program
`checkExit()`	To stop the Java virtual machine and exit the Java environment
`checkLink()`	To dynamically link an external library into the Java environment
`checkListen()`	To listen for a network connection
`checkMemberAccess()`	To access the members of a class
`checkMulticast()`	To use a multicast connection
`checkPackageAccess()`	To access the classes in a package
`checkPackageDefinition()`	To define classes in a package
`checkPrintJobAccess()`	To initiate a print job request
`checkPropertiesAccess()`	To get or set the `Properties` object that defines all of the system properties
`checkPropertyAccess()`	To get or set a system property
`checkRead()`	To read from a file or input stream
`checkSecurityAccess()`	To perform a security action
`checkSetFactory()`	To set a factory class that determines classes to be used for managing network connections and their content
`checkSystemClipboardAccess()`	To access the system clipboard
`checkTopLevelWindow()`	To create a top-level window on the screen
`checkWrite()`	To write to a file or output stream

The `SecurityManager` class provides implementations of these methods that always refuse the requested permission. To implement a more permissive security policy, you need to create a subclass of `SecurityManager` that implements that policy.

In Java 1.0, most browsers consider an applet to be trusted or untrusted. An untrusted applet is one that does not come from the local filesystem. An untrusted applet is treated as follows by most popular browsers:

- It can establish network connections to the network address from which it came.
- It can create new windows on the screen. However, a notice is displayed on the bottom of the window that the window was created by an untrusted applet.
- It cannot access any other external resources. In particular, untrusted applets cannot access local files.

As of Java 1.1, an applet can have a digital signature attached to it. When an applet has been signed by a trusted entity, a browser may consider the applet to be trusted and relax its security policy.

PREVIOUS HOME NEXT
URL Objects BOOK INDEX ClassLoader

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

11. The java.io Package

Contents:

InvalidClassException

InvalidObjectException

IOException

LineNumberInputStream

LineNumberReader

NotActiveException

NotSerializableException

ObjectInput

ObjectInputStream

ObjectInputValidation

ObjectOutput

ObjectOutputStream

ObjectStreamClass

ObjectStreamException

OptionalDataException

OutputStream

OutputStreamWriter

PipedInputStream

PipedOutputStream

PipedReader

PipedWriter

PrintStream

PrintWriter

PushbackInputStream

PushbackReader

RandomAccessFile

Reader

SequenceInputStream

Serializable

StreamCorruptedException

StreamTokenizer

StringBufferInputStream

StringReader

StringWriter

SyncFailedException

UnsupportedEncodingException

UTFDataFormatException

WriteAbortedException

Writer

The package `java.io` contains the classes that handle fundamental input and output operations in Java. The I/O classes can be grouped as follows:

- Classes for reading input from a stream of data.
- Classes for writing output to a stream of data.
- Classes that manipulate files on the local filesystem.
- Classes that handle object serialization.

I/O in Java is based on streams. A stream represents a flow of data or a channel of communication. Java 1.0 supports only byte streams. The `InputStream` class is the superclass of all of the Java 1.0 byte input streams, while `OutputStream` is the superclass of all the byte output streams. The drawback to these byte streams is that they do not always handle Unicode characters correctly.

As of Java 1.1, `java.io` contains classes that represent character streams. These character stream classes handle Unicode characters appropriately by using a character encoding to convert bytes to characters and vice versa. The `Reader` class is the superclass of all the Java 1.1 character input streams, while `Writer` is the superclass of all character output streams.

The `InputStreamReader` and `OutputStreamWriter` classes provide a bridge between byte streams and character streams. If you wrap an `InputStreamReader` around an `InputStream` object, the bytes in the byte stream are read and converted to characters using the character encoding scheme specified by the `InputStreamReader`. Likewise, you can wrap an `OutputStreamWriter` around any `OutputStream` object so that you can write characters and have them converted to bytes.

As of Java 1.1, `java.io` also contains classes to support object serialization. Object serialization is the ability to write the complete state of an object to an output stream, and then later recreate that object by reading in the serialized state from an input stream. The `ObjectOutputStream` and `ObjectInputStream` classes handle serializing and deserializing objects, respectively.

The `RandomAccessFile` class is the only class that does not use a stream for reading or writing data. As its name implies, `RandomAccessFile` provides nonsequential access to a file for both reading and writing purposes.

The `File` class represents a file on the local file system. The class provides methods to identify and retrieve information about a file.

Figure 11.1 shows the class hierarchy for the `java.io` package. The `java.io` package defines a number of standard I/O exception classes. These exception classes are all subclasses of `IOException`, as shown in Figure 11.2.

Figure 11.1: The java.io package

[Graphic: Figure 11-1]

Figure 11.2: The exception classes in the java.io package

[Graphic: Figure 11-2]

BufferedInputStream

Name

BufferedInputStream

Synopsis

Class Name:

`java.io.BufferedInputStream`

Superclass:

`java.io.FilterInputStream`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

A BufferedInputStream object provides a more efficient way to read just a few bytes at a time from an InputStream. BufferedInputStream object use a buffer to store input from an associated InputStream. In other words, a large number of bytes are read from the underlying stream and stored in an internal buffer. A BufferedInputStream is more efficient than a regular InputStream because reading data from memory is faster than reading it from a disk or a network. All reading is done directly from the internal buffer; the disk or network needs to be accessed only occasionally to fill up the buffer.

You should wrap a BufferedInputStream around any InputStream whose read() operations may be time consuming or costly, such as a FileInputStream.

BufferedInputStream provides a way to mark a position in the stream and subsequently reset the stream to that position, using mark() and reset().

Class Summary

```
public class java.io.BufferedInputStream extends java.io.FilterInputStream {
  // Variables
  protected byte[] buf;
  protected int count;
  protected int marklimit;
  protected int markpos;
  protected int pos;
  // Constructors
  public BufferedInputStream(InputStream in);
  public BufferedInputStream(InputStream in, int size);
  // Instance Methods
  public synchronized int available();
  public synchronized void mark(int readlimit);
  public boolean markSupported();
  public synchronized int read();
  public synchronized int read(byte[] b, int off, int len);
  public synchronized void reset();
  public synchronized long skip(long n);
}
```

Variables

buf

protected byte[] buf

Description

> The buffer that stores the data from the input stream.

count

protected int count

Description

> A placeholder that marks the end of valid data in the buffer.

marklimit

protected int marklimit

Description

> The maximum number of bytes that can be read after a call to `mark()` before a call to `reset()` fails.

markpos

protected int markpos

Description

> The position of the stream when `mark()` was called. If `mark()` has not been called, this variable is `-1`.

pos

protected int pos

Description

> The current position in the buffer, or in other words, the index of the next character to be read.

Constructors

BufferedInputStream

public BufferedInputStream(InputStream in)

Parameters

`in`

The input stream to buffer.

Description

This constructor creates a `BufferedInputStream` that buffers input from the given `InputStream`, using a buffer with the default size of 2048 bytes.

public BufferedInputStream(InputStream in, int size)

Parameters

`in`

The input stream to buffer.

`size`

The size of buffer to use.

Description

This constructor creates a `BufferedInputStream` that buffers input from the given `InputStream`, using a buffer of the given size.

Instance Methods

available

public synchronized int available() throws IOException

Returns

The number of bytes that can be read without blocking.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterInputStream.available()`

Description

This method returns the number of bytes that can be read without having to wait for more data to become available. The returned value is the sum of the number of bytes remaining in the object's buffer and the number returned as the result of calling the `available()` method of the underlying `InputStream` object.

mark

public synchronized void mark(int readlimit)

Parameters

`readlimit`

The maximum number of bytes that can be read before the saved position becomes invalid.

Overrides

`FilterInputStream.mark()`

Description

This method causes the `BufferedInputStream` to remember its current position. A subsequent call to `reset()` causes the object to return to that saved position, and thus reread a portion of the buffer.

markSupported

public synchronized boolean markSupported()

Returns

The `boolean` value `true`.

Overrides

`FilterInputStream.markSupported()`

Description

This method returns `true` to indicate that this class supports `mark()` and `reset()`.

read

public synchronized int read() throws IOException

Returns

The next byte of data or `-1` if the end of the stream is encountered.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterInputStream.read()`

Description

This method returns the next byte from the buffer. If all the bytes in the buffer have been read, the buffer is filled from the underlying `InputStream` and the next byte is returned. If the buffer does not need to be filled, this method returns immediately. If the buffer needs to be filled, this method blocks until data is available from the underlying `InputStream`, the end of the stream is reached, or an exception is thrown.

public synchronized int read(byte b[], int off, int len) throws IOException

Parameters

`b`

An array of bytes to be filled from the stream.

`off`

An offset into the byte array.

`len`

The number of bytes to read.

Returns

The actual number of bytes read or `-1` if the end of the stream is encountered immediately.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterInputStream.read(byte[], int, int)`

Description

This method copies bytes from the internal buffer into the given array `b`, starting at index `off` and continuing for up to `len` bytes. If there are any bytes in the buffer, this method returns immediately. Otherwise the buffer needs to be filled; this method blocks until the data is available from the underlying `InputStream`, the end of the stream is reached, or an exception is thrown.

reset

public synchronized void reset() throws IOException

Throws

`IOException`

If there was no previous call to this `BufferedInputStream`'s mark method, or the saved position has been invalidated.

Overrides

`FilterInputStream.reset()`

Description

This method sets the position of the `BufferedInputStream` to a position that was saved by a previous call to `mark()`. Subsequent bytes read from this `BufferedInputStream` will begin from the saved position and continue normally.

skip

public synchronized long skip(long n) throws IOException

Parameters

`n`

The number of bytes to skip.

Returns

The actual number of bytes skipped.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterInputStream.skip()`

Description

This method skips `n` bytes of input. If the new position of the stream is still within the data contained in the buffer, the method returns immediately. Otherwise the `skip()` method of the underlying stream is called. A subsequent call to `read()` forces the buffer to be filled.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`close()`	`FilterInputStream`
`equals(Object)`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`

read(byte[])	FilterInputStream	toString()	Object
void wait()	Object	void wait(long)	Object
void wait(long, int)	Object		

See Also

`FilterInputStream`, `InputStream`, `IOException`

12. The java.lang Package

Contents:

InstantiationError
InstantiationException
InternalError
InterruptedException
LinkageError
Long
Math
NegativeArraySizeException
NoClassDefFoundError
NoSuchFieldError
NoSuchFieldException
NoSuchMethodError
NoSuchMethodException
NullPointerException
Number
NumberFormatException
Object
OutOfMemoryError
Process
Runnable
Runtime
RuntimeException
SecurityException
SecurityManager
Short
StackOverflowError
String
StringBuffer
StringIndexOutOfBoundsException
System
Thread
ThreadDeath
ThreadGroup
Throwable
UnknownError
UnsatisfiedLinkError
VerifyError
VirtualMachineError
Void

The package `java.lang` contains classes and interfaces that are essential to the Java language. These include:

- `Object`, the ultimate superclass of all classes in Java.
- `Thread`, the class that controls each thread in a multithreaded program.
- `Throwable`, the superclass of all error and exception classes in Java.
- Classes that encapsulate the primitive data types in Java.
- Classes for accessing system resources and other low-level entities.
- `Math`, a class that provides standard mathematical methods.
- `String`, the class that represents strings.

Because the classes in the `java.lang` package are so essential, the `java.lang` package is implicitly imported by every Java source file. In other words, you can refer to all of the classes and interfaces in `java.lang` using their simple names.

Figure 12.1 shows the class hierarchy for the `java.lang` package.

The possible exceptions in a Java program are organized in a hierarchy of exception classes. The `Throwable` class is at the root of the exception hierarchy. `Throwable` has two immediate subclasses: `Exception` and `Error`. Figure 12.2 shows the standard exception classes defined in the `java.lang` package, while Figure 12.3 shows the standard error classes defined in `java.lang`.

Figure 12.1: The java.lang package

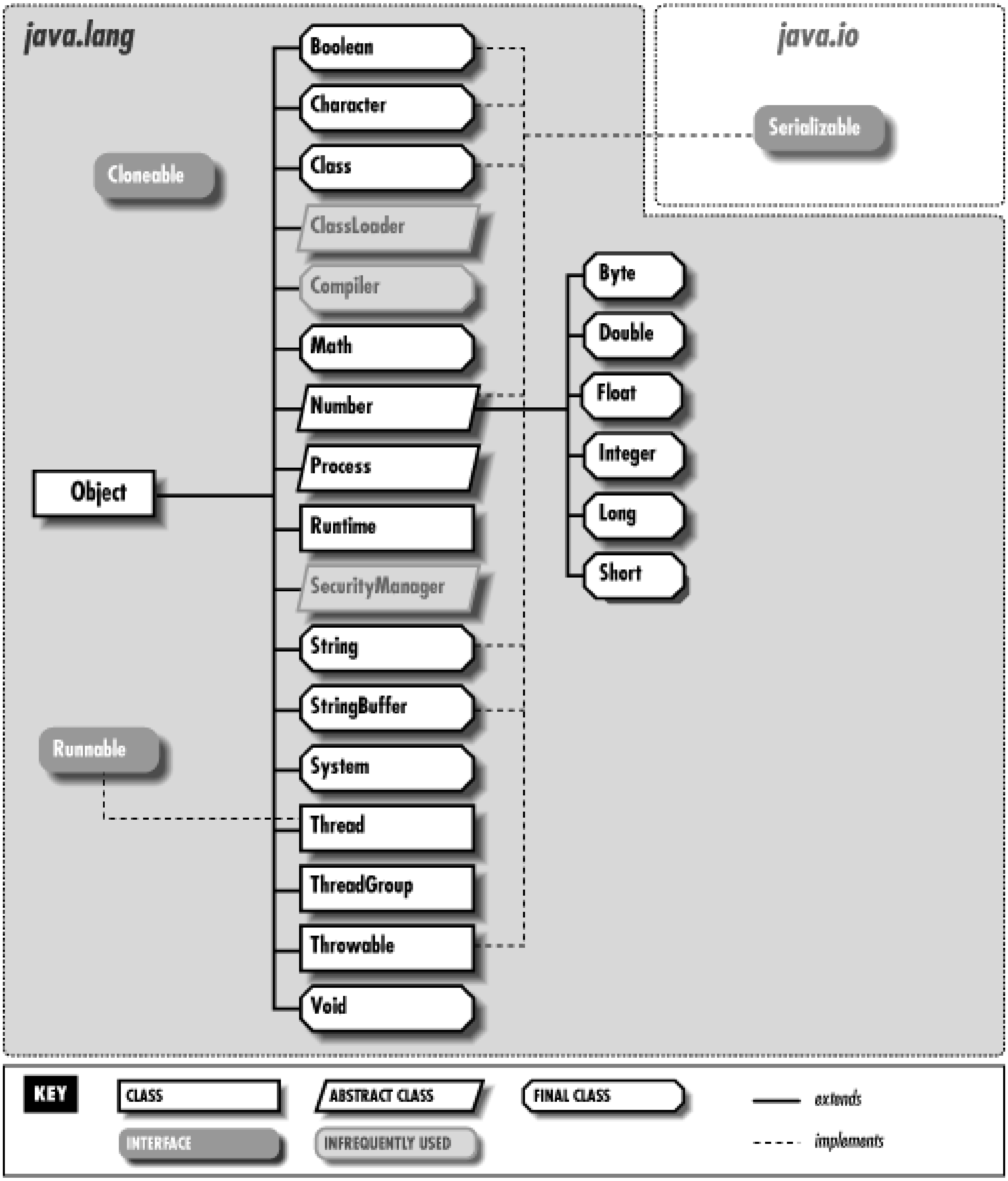


Figure 12.2: The exception classes in the java.lang package

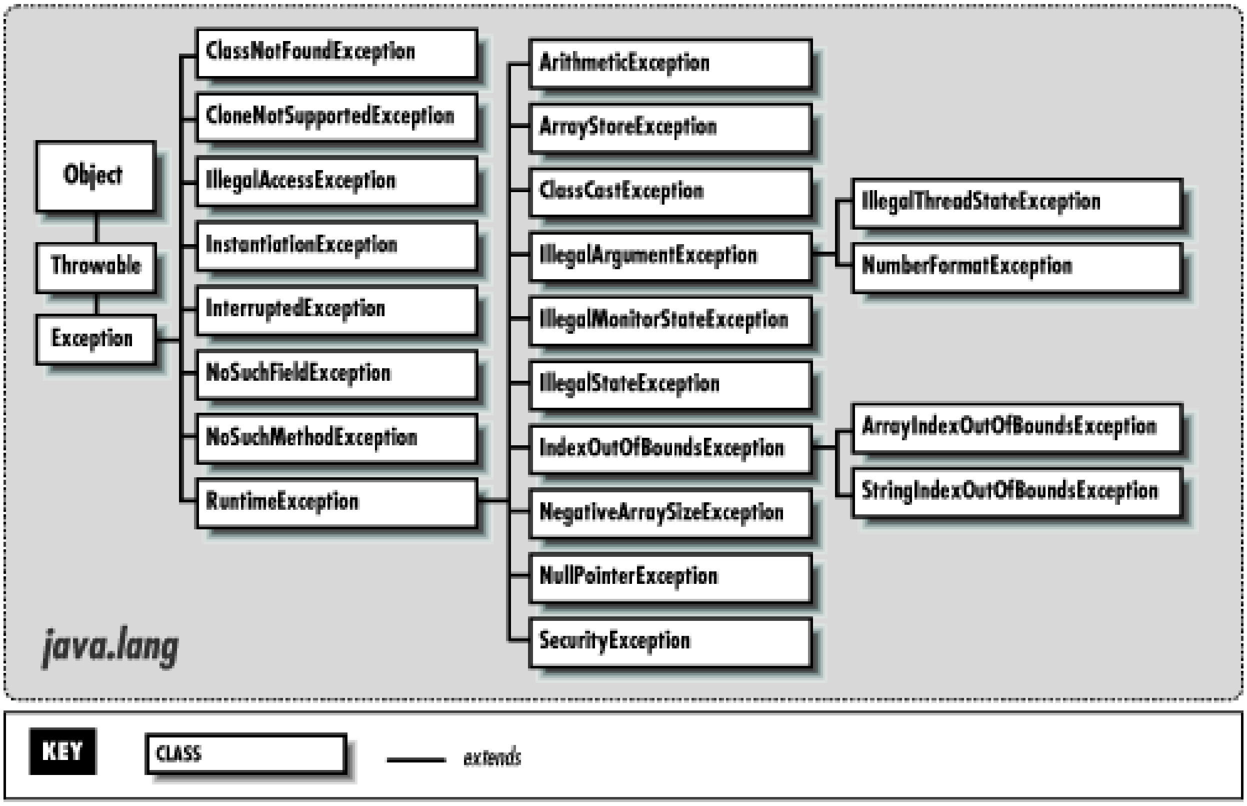
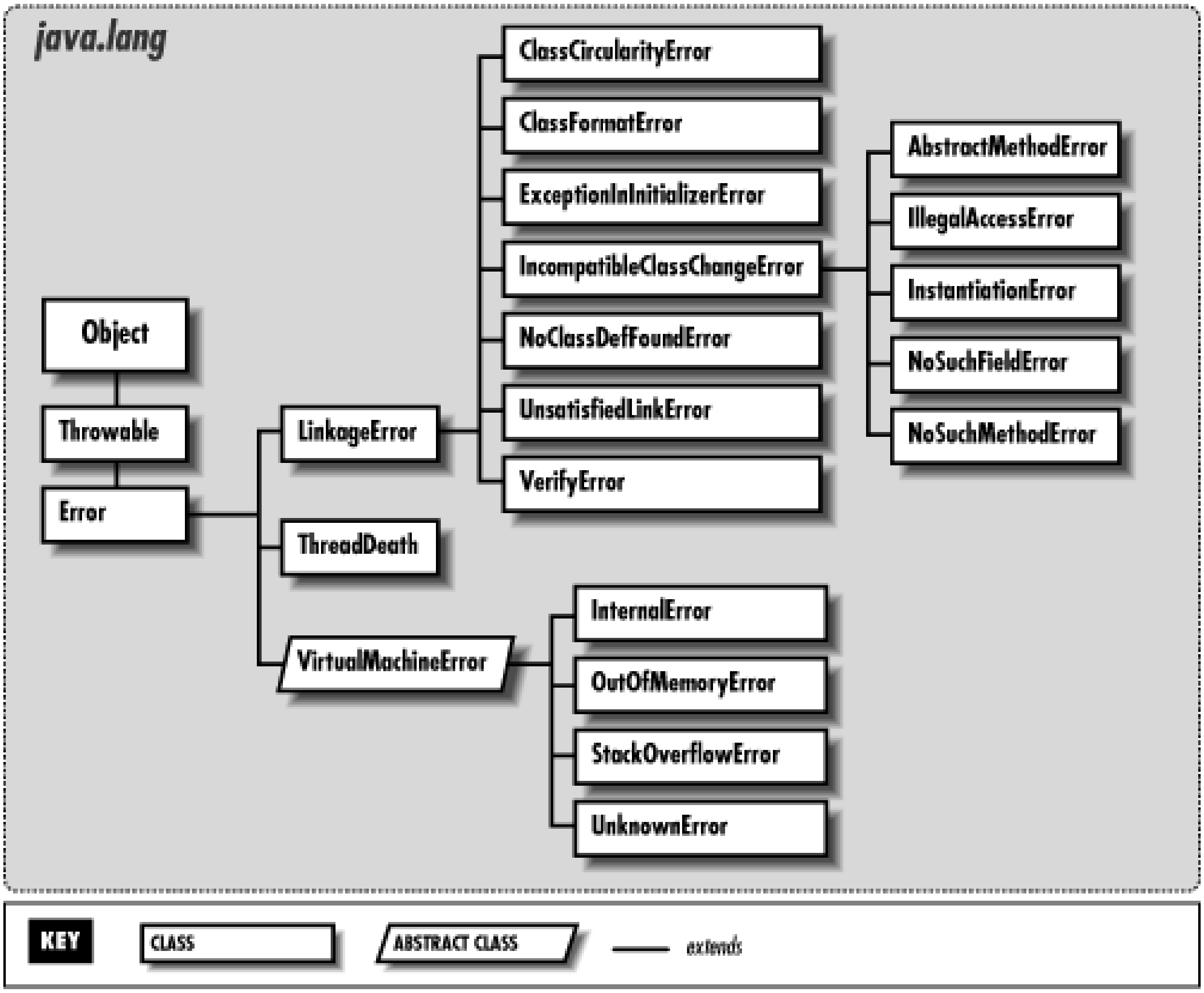


Figure 12.3: The error classes in the java.lang package



AbstractMethodError

Name

AbstractMethodError

Synopsis

Class Name:

`java.lang.AbstractMethodError`

Superclass:

`java.lang.IncompatibleClassChangeError`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

An `AbstractMethodError` is thrown when there is an attempt to invoke an `abstract` method.

Class Summary

```
public class java.lang.AbstractMethodError
             extends java.lang.IncompatibleClassChangeError {
  // Constructors
  public AbstractMethodError();
  public AbstractMethodError(String s);
}
```

Constructors

AbstractMethodError

public AbstractMethodError()

Description

This constructor creates an `AbstractMethodError` with no associated detail message.

public AbstractMethodError(String s)

Parameters

`s`

The detail message.

Description

This constructor creates an `AbstractMethodError` with the specified detail message.

Inherited Methods

Method	**Inherited From**	**Method**	**Inherited From**
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Error`, `IncompatibleClassChangeError`, `Throwable`

PREVIOUS | HOME | NEXT
Writer | BOOK INDEX | ArithmeticException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

13. The java.lang.reflect Package

Contents:

The package `java.lang.reflect` is new as of Java 1.1. It contains classes and interfaces that support the Reflection API. Reflection refers to the ability of a class to reflect upon itself, or look inside of itself, to see what it can do. The Reflection API makes it possible to:

- Discover the variables, methods, and constructors of any class.
- Create an instance of any class using any available constructor of that class, even if the class initiating the creation was not compiled with any information about the class to be instantiated.
- Access the variables of any object, even if the accessing class was not compiled with any information about the class to be accessed.
- Call the methods of any object, even if the calling class was not compiled with any information about the class that contains the methods.
- Create an array of objects that are instances of any class, even if the creating class was not compiled with any information about the class.

These capabilities are implemented by the `java.lang.Class` class and the classes in the `java.lang.reflect` package. Figure 13.1 shows the class hierarchy for the `java.lang.reflect` package.

Figure 13.1: The java.lang.reflect package

[Graphic: Figure 13-1]

Java 1.1 currently uses the Reflection API for two purposes:

- The JavaBeans API supports a mechanism for customizing objects that is based on being able to discover their public variables, methods, and constructors. See the forthcoming *Developing Java Beans* from O'Reilly & Associates for more information about the JavaBeans API.
- The object serialization functionality in `java.io` is built on top of the Reflection API. Object serialization allows arbitrary objects to be written to a stream of bytes and then read back later as objects.

Array

Name

Array

Synopsis

Class Name:

`java.lang.reflect.Array`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

`java.lang.Cloneable`, `java.io.Serializable`

Availability:

New as of JDK 1.1

Description

The `Array` class provides static methods to manipulate arbitrary arrays in Java. There are methods to set and retrieve elements in an array, determine the size of an array, and create a new instance of an array.

The `Array` class is used to create array objects and manipulate their elements. The `Array` class is not used to represent array types. Because arrays in Java are objects, array types are represented by `Class` objects.

Class Summary

```
public final class java.lang.reflect.Array extends java.lang.Object {
  // Class Methods
  public static native Object get(Object array, int index);
  public static native boolean getBoolean(Object array, int index);
  public static native byte getByte(Object array, int index);
  public static native char getChar(Object array, int index);
  public static native double getDouble(Object array, int index);
  public static native float getFloat(Object array, int index);
  public static native int getInt(Object array, int index);
  public static native int getLength(Object array);
  public static native long getLong(Object array, int index);
  public static native short getShort(Object array, int index);
  public static Object newInstance(Class componentType, int length);
  public static Object newInstance(Class componentType, int[] dimensions);
  public static native void set(Object array, int index, Object value);
  public static native void setBoolean(Object array, int index, boolean z);
  public static native void setByte(Object array, int index, byte b);
  public static native void setChar(Object array, int index, char c);
  public static native void setDouble(Object array, int index, double d);
  public static native void setFloat(Object array, int index, float f);
  public static native void setInt(Object array, int index, int i);
  public static native void setLong(Object array, int index, long l);
  public static native void setShort(Object array, int index, short s);
}
```

Class Methods

get

public static native Object get(Object array, int index) throws IllegalArgumentException, ArrayIndexOutOfBoundsException

Parameters

`array`

An array object.

`index`

An index into the array.

Returns

The object at the given index in the specified array.

Throws

`IllegalArgumentException`

If the given object is not an array.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method returns the object at the given index in the array. If the array contains values of a primitive type, the value at the given index is wrapped in an appropriate object, and the object is returned.

getBoolean

public static native boolean getBoolean(Object array, int index) throws IllegalArgumentException, ArrayIndexOutOfBoundsException

Parameters

`array`

An array object.

`index`

An index into the array.

Returns

The `boolean` value at the given index in the specified array.

Throws

`IllegalArgumentException`

If the given object is not an array, or the object at the given index cannot be converted to a `boolean`.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method returns the object at the given index in the array as a `boolean` value.

getByte

```
public static native byte getByte(Object array, int index) throws
IllegalArgumentException, ArrayIndexOutOfBoundsException
```

Parameters

`array`

An array object.

`index`

An index into the array.

Returns

The `byte` value at the given index in the specified array.

Throws

`IllegalArgumentException`

If the given object is not an array, or the object at the given index cannot be converted to a `byte`.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method returns the object at the given index in the array as a `byte` value.

getChar

```
 public static native char getChar(Object array, int index) throws
IllegalArgumentException, ArrayIndexOutOfBoundsException
```

Parameters

`array`

An array object.

`index`

An index into the array.

Returns

The `char` value at the given index in the specified array.

Throws

`IllegalArgumentException`

If the given object is not an array, or the object at the given index cannot be converted to a `char`.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method returns the object at the given index in the array as a `char` value.

getDouble

`public static native double getDouble(Object array, int index) throws IllegalArgumentException, ArrayIndexOutOfBoundsException`

Parameters

`array`

An array object.

`index`

An index into the array.

Returns

The `double` value at the given index in the specified array.

Throws

`IllegalArgumentException`

If the given object is not an array, or the object at the given index cannot be converted to a `double`.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method returns the object at the given index in the array as a `double` value.

getFloat

`public static native float getFloat(Object array, int index) throws`

IllegalArgumentException, ArrayIndexOutOfBoundsException

Parameters

`array`

An array object.

`index`

An index into the array.

Returns

The `float` value at the given index in the specified array.

Throws

`IllegalArgumentException`

If the given object is not an array, or the object at the given index cannot be converted to a `float`.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method returns the object at the given index in the array as a `float` value.

getInt

public static native int getInt(Object array, int index) throws IllegalArgumentException, ArrayIndexOutOfBoundsException

Parameters

`array`

An array object.

`index`

An index into the array.

Returns

The `int` value at the given index in the specified array.

Throws

`IllegalArgumentException`

If the given object is not an array, or the object at the given index cannot be converted to a `int`.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method returns the object at the given index in the array as a `int` value.

getLength

`public static native int getLength(Object array) throws IllegalArgumentException`

Parameters

`array`

An array object.

Returns

The length of the specified array.

Throws

`IllegalArgumentException`

If the given object is not an array.

Description

This method returns the length of the array.

getLong

`public static native long getLong(Object array, long index) throws IllegalArgumentException, ArrayIndexOutOfBoundsException`

Parameters

`array`

An array object.

`index`

An index into the array.

Returns

The `long` value at the given index in the specified array.

Throws

`IllegalArgumentException`

If the given object is not an array, or the object at the given index cannot be converted to a `long`.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method returns the object at the given index in the array as a `long` value.

getShort

`public static native short getShort(Object array, short index) throws IllegalArgumentException, ArrayIndexOutOfBoundsException`

Parameters

`array`

An array object.

`index`

An index into the array.

Returns

The `short` value at the given index in the specified array.

Throws

`IllegalArgumentException`

If the given object is not an array, or the object at the given index cannot be converted to a `short`.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method returns the object at the given index in the array as a `short` value.

newInstance

`public static Object newInstance(Class componentType, int length) throws NegativeArraySizeException`

Parameters

`componentType`

The type of each element in the array.

`length`

The length of the array.

Returns

An array object that contains elements of the given component type and has the specified length.

Throws

`NegativeArraySizeException`

If `length` is negative.

`NullPointerException`

If `componentType` is `null`.

Description

This method creates a single-dimension array with the given length and component type.

`public static Object newInstance(Class componentType, int[] dimensions) throws IllegalArgumentException, NegativeArraySizeException`

Parameters

`componentType`

The type of each element in the array.

`dimensions`

An array that specifies the dimensions of the array to be created.

Returns

An array object that contains elements of the given component type and has the specified number of dimensions.

Throws

`IllegalArgumentException`

If `dimensions` has zero dimensions itself, or if it has too many dimensions (typically 255 array dimensions are supported).

`NegativeArraySizeException`

If `length` is negative.

`NullPointerException`

If `componentType` is `null`.

Description

This method creates a multidimensional array with the given dimensions and component type.

set

`public static native void set(Object array, int index, Object value) throws IllegalArgumentException, ArrayIndexOutOfBoundsException`

Parameters

`array`

An array object.

`index`

An index into the array.

`value`

The new value.

Throws

`IllegalArgumentException`

If the given object is not an array, or if it represents an array of primitive values, and the given value cannot be unwrapped and converted to that primitive type.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method sets the object at the given index in the array to the specified value. If the array contains values of a primitive type, the given value is automatically unwrapped before it is put in the array.

setBoolean

`public static native void setBoolean(Object array, int index, boolean z) throws IllegalArgumentException, ArrayIndexOutOfBoundsException`

Parameters

`array`

An array object.

`index`

An index into the array.

`z`

The new value.

Throws

`IllegalArgumentException`

If the given object is not an array, or if the given value cannot be converted to the component type of the array.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method sets the element at the given index in the array to the given `boolean` value.

setByte

```
public static native void setByte(Object array, int index, byte b) throws
IllegalArgumentException, ArrayIndexOutOfBoundsException
```

Parameters

`array`

An array object.

`index`

An index into the array.

`b`

The new value.

Throws

`IllegalArgumentException`

If the given object is not an array, or if the given value cannot be converted to the component type of the array.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method sets the element at the given index in the array to the given `byte` value.

setChar

`public static native void setChar(Object array, int index, char c) throws IllegalArgumentException, ArrayIndexOutOfBoundsException`

Parameters

`array`

An array object.

`index`

An index into the array.

`c`

The new value.

Throws

`IllegalArgumentException`

If the given object is not an array, or if the given value cannot be converted to the component type of the array.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method sets the element at the given index in the array to the given `char` value.

setDouble

`public static native void setDouble(Object array, int index, double d) throws IllegalArgumentException, ArrayIndexOutOfBoundsException`

Parameters

`array`

An array object.

`index`

An index into the array.

`d`

The new value.

Throws

`IllegalArgumentException`

If the given object is not an array, or if the given value cannot be converted to the component type of the array.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method sets the element at the given index in the array to the given `double` value.

setFloat

```
 public static native void setFloat(Object array, int index, float f) throws
IllegalArgumentException, ArrayIndexOutOfBoundsException
```

Parameters

`array`

An array object.

`index`

An index into the array.

`f`

The new value.

Throws

`IllegalArgumentException`

If the given object is not an array, or if the given value cannot be converted to the component type of the array.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method sets the element at the given index in the array to the given `float` value.

setInt

public static native void setInt(Object array, int index, int i) throws IllegalArgumentException, ArrayIndexOutOfBoundsException

Parameters

`array`

An array object.

`index`

An index into the array.

`i`

The new value.

Throws

`IllegalArgumentException`

If the given object is not an array, or if the given value cannot be converted to the component type of the array.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method sets the element at the given index in the array to the given `int` value.

setLong

public static native void setLong(Object array, int index, long l) throws IllegalArgumentException, ArrayIndexOutOfBoundsException

Parameters

`array`

An array object.

`index`

An index into the array.

`l`

The new value.

Throws

`IllegalArgumentException`

If the given object is not an array, or if the given value cannot be converted to the component type of the array.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method sets the element at the given index in the array to the given `long` value.

setShort

`public static native void setShort(Object array, int index, short s) throws IllegalArgumentException, ArrayIndexOutOfBoundsException`

Parameters

`array`

An array object.

`index`

An index into the array.

`s`

The new value.

Throws

`IllegalArgumentException`

If the given object is not an array, or if the given value cannot be converted to the component type of the array.

`ArrayIndexOutOfBoundsException`

If the given index is invalid.

`NullPointerException`

If `array` is `null`.

Description

This method sets the element at the given index in the array to the given `short` value.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`ArrayIndexOutOfBoundsException`, `Class`, `IllegalArgumentException`, `NegativeArraySizeException`, `NullPointerException`, `Object`

PREVIOUS HOME NEXT
Void BOOK INDEX Constructor

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

14. The java.math Package

Contents:

The package `java.math` is new as of Java 1.1. It contains two classes that support arithmetic on arbitrarily large integers and floating-point numbers. Figure 14.1 shows the class hierarchy for the `java.math` package.

Figure 14.1: The java.math package

[Graphic: Figure 14-1]

BigDecimal

Name

BigDecimal

Synopsis

Class Name:

```
java.math.BigDecimal
```

Superclass:

```
java.lang.Number
```

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `BigDecimal` class represents arbitrary-precision rational numbers. A `BigDecimal` object provides a good way to represent a real number that exceeds the range or precision that can be represented by a `double` value or the rounding that is done on a `double` value is unacceptable.

The representation for a `BigDecimal` consists of an unlimited precision integer value and an integer scale factor. The scale factor indicates a power of 10 that the integer value is implicitly divided by. For example, a `BigDecimal` would represent the value 123.456 with an integer value of 123456 and the scale factor of 3. Note that the scale factor cannot be negative and a `BigDecimal` cannot overflow.

Most of the methods in `BigDecimal` perform mathematical operations or make comparisons with other `BigDecimal` objects. Operations that result in some loss of precision, such as division, require a rounding method to be specified. The `BigDecimal` class defines constants to represent the different rounding methods. The rounding method determines if the digit before a discarded fraction is rounded up or left unchanged.

Class Summary

```
public class java.math.BigDecimal extends java.lang.Number {
  // Constants
  public static final int ROUND_CEILING;
  public static final int ROUND_DOWN;
  public static final int ROUND_FLOOR;
  public static final int ROUND_HALF_DOWN;
  public static final int ROUND_HALF_EVEN;
  public static final int ROUND_HALF_UP;
  public static final int ROUND_UNNECESSARY;
  public static final int ROUND_UP;
  // Constructors
  public BigDecimal(double val);
  public BigDecimal(String val);
  public BigDecimal(BigInteger val);
  public BigDecimal(BigInteger val, int scale);
  // Class Methods
  public static BigDecimal valueOf(long val);
  public static BigDecimal valueOf(long val, int scale);
  // Instance Methods
  public BigDecimal abs();
  public BigDecimal add(BigDecimal val);
  public int compareTo(BigDecimal val);
  public BigDecimal divide(BigDecimal val, int roundingMode);
  public BigDecimal divide(BigDecimal val, int scale, int roundingMode);
  public double doubleValue();
  public boolean equals(Object x);
  public float floatValue();
  public int hashCode();
  public int intValue();
  public long longValue();
  public BigDecimal max(BigDecimal val);
  public BigDecimal min(BigDecimal val);
  public BigDecimal movePointLeft(int n);
  public BigDecimal movePointRight(int n);
  public BigDecimal multiply(BigDecimal val);
  public BigDecimal negate();
  public int scale();
  public BigDecimal setScale(int scale);
  public BigDecimal setScale(int scale, int roundingMode);
  public int signum();
  public BigDecimal subtract(BigDecimal val);
  public BigInteger toBigInteger();
  public String toString();
}
```

Constants

ROUND_CEILING

public static final int ROUND_CEILING

Description

> A rounding method that rounds towards positive infinity. Under this method, the value is rounded to the least integer greater than or equal to its value. For example, 2.5 rounds to 3 and -2.5 rounds to -2.

ROUND_DOWN

public static final int ROUND_DOWN

Description

> A rounding method that rounds towards zero by truncating. For example, 2.5 rounds to 2 and -2.5 rounds to -2.

ROUND_FLOOR

public static final int ROUND_FLOOR

Description

A rounding method that rounds towards negative infinity. Under this method, the value is rounded to the greatest integer less than or equal to its value. For example, 2.5 rounds to 2 and -2.5 rounds to -3.

ROUND_HALF_DOWN

public static final int ROUND_HALF_DOWN

Description

A rounding method that increments the digit prior to a discarded fraction if the fraction is greater than 0.5; otherwise, the digit is left unchanged. For example, 2.5 rounds to 2, 2.51 rounds to 3, -2.5 rounds to -2, and -2.51 rounds to -3.

ROUND_HALF_EVEN

public static final int ROUND_HALF_EVEN

Description

A rounding method that behaves like `ROUND_HALF_UP` if the digit prior to the discarded fraction is odd; otherwise it behaves like `ROUND_HALF_DOWN`. For example, 2.5 rounds to 2, 3.5 rounds to 4, -2.5 rounds to -2, and -3.5 rounds to -4.

ROUND_HALF_UP

public static final int ROUND_HALF_UP

Description

A rounding method that increments the digit prior to a discarded fraction if the fraction is greater than or equal to 0.5; otherwise, the digit is left unchanged. For example, 2.5 rounds to 3, 2.49 rounds to 2, -2.5 rounds to -3, and -2.49 rounds to -2.

ROUND_UNNECESSARY

public static final int ROUND_UNNECESSARY

Description

A constant that specifies that rounding is not necessary. If the result really does require rounding, an `ArithmeticException` is thrown.

ROUND_UP

public static final int ROUND_UP

Description

A rounding method that rounds away from zero by truncating. For example, 2.5 rounds to 3 and -2.5 rounds to -3.

Constructors

BigDecimal

public BigDecimal(double val) throws NumberFormatException

Parameters

`val`

The initial value.

Throws

`NumberFormatException`

If the `double` has any of the special values: `Double.NEGATIVE_INFINITY`, `Double.POSITIVE_INFINITY`, or `Double.NaN`.

Description

This constructor creates a `BigDecimal` with the given initial value. The scale of the `BigDecimal` that is created is the smallest value such that (`10^scale x val`) is an integer.

public BigDecimal(String val) throws NumberFormatException

Parameters

`val`

The initial value.

Throws

`NumberFormatException`

If the string cannot be parsed into a valid `BigDecimal`.

Description

This constructor creates a `BigDecimal` with the initial value specified by the `String`. The string can contain an optional minus sign, followed by zero or more decimal digits, followed by an optional fraction. The fraction must contain a decimal point and zero or more decimal digits. The string must contain as least one digit in the integer or fractional part. The scale of the `BigDecimal` that is created is equal to the number of digits to the right of the decimal point or 0 if there is no decimal point. The mapping from characters to digits is provided by the `Character.digit()` method.

public BigDecimal(BigInteger val)

Parameters

`val`

The initial value.

Description

This constructor creates a `BigDecimal` whose initial value comes from the given `BigInteger`. The scale of the `BigDecimal` that is created is 0.

`public BigDecimal(BigInteger val, int scale) throws NumberFormatException`

Parameters

`val`

The initial value.

`scale`

The initial scale.

Throws

`NumberFormatException`

If `scale` is negative.

Description

This constructor creates a `BigDecimal` from the given parameters. The `scale` parameter specifies how many digits of the supplied `BigInteger` fall to the right of the decimal point.

Class Methods

valueOf

public static BigDecimal valueOf(long val)

Parameters

`val`

The initial value.

Returns

A `BigDecimal` that represents the given value.

Description

This method creates a `BigDecimal` from the given `long` value. The scale of the `BigDecimal` that is created is 0.

`public static BigDecimal valueOf(long val, int scale) throws NumberFormatException`

Parameters

`val`

The initial value.

`scale`

The initial scale.

Returns

A `BigDecimal` that represents the given value and scale.

Throws

`NumberFormatException`

If `scale` is negative.

Description

This method creates a `BigDecimal` from the given parameters. The `scale` parameter specifies how many digits of the supplied `long` fall to the right of the decimal point.

Instance Methods

abs

public BigDecimal abs()

Returns

A `BigDecimal` that contains the absolute value of this number.

Description

This method returns the absolute value of this `BigDecimal`. If this `BigDecimal` is nonnegative, it is returned. Otherwise, a new `BigDecimal` that contains the absolute value of this `BigDecimal` is returned. The scale of the new `BigDecimal` is the same as that of this `BigDecimal`.

add

public BigDecimal add(BigDecimal val)

Parameters

`val`

The number to be added.

Returns

A new `BigDecimal` that contains the sum of this number and the given value.

Description

This method returns the sum of this `BigDecimal` and the given `BigDecimal` as a new `BigDecimal`. The value of the new `BigDecimal` is the sum of the values of the two `BigDecimal` objects being added; the scale is the maximum of their two scales.

compareTo

public int compareTo(BigDecimal val)

Parameters

`val`

The number to be compared.

Returns

`-1` if this number is less than `val`, `0` if this number is equal to `val`, or `1` if this number is greater than `val`.

Description

This method compares this `BigDecimal` to the given `BigDecimal` and returns a value that indicates the result of the comparison. The method considers two `BigDecimal` objects that have the same values but different scales to be equal. This method can be used to implement all six of the standard boolean comparison operators: `==`, `!=`, `<=`, `<`, `>=`, and `>`.

divide

public BigDecimal divide(BigDecimal val, int roundingMode) throws ArithmeticException, IllegalArgumentException

Parameters

`val`

The divisor.

`roundingMode`

The rounding mode.

Returns

A new `BigDecimal` that contains the result (quotient) of dividing this number by the supplied value.

Throws

`ArithmeticException`

If `val` is `0`, or if `ROUND_UNNECESSARY` is specified for the rounding mode but rounding is necessary.

`IllegalArgumentException`

If `roundingMode` is not a valid value.

Description

This method returns the quotient that results from dividing this `BigDecimal` by the given `BigDecimal` and applying the specified rounding mode. The quotient is returned as a new `BigDecimal` that has the same scale as the scale of this `BigDecimal` scale. One of the rounding constants must be specified for the rounding mode.

public BigDecimal divide(BigDecimal val, int scale, int roundingMode) throws ArithmeticException, IllegalArgumentException

Parameters

`val`

The divisor.

`scale`

The scale for the result.

`roundingMode`

The rounding mode.

Returns

A new `BigDecimal` that contains the result (quotient) of dividing this number by the supplied value.

Throws

`ArithmeticException`

If `val` is `0`, if `scale` is less than zero, or if `ROUND_UNNECESSARY` is specified for the rounding mode but rounding is necessary.

`IllegalArgumentException`

If `roundingMode` is not a valid value.

Description

This method returns the quotient that results from dividing dividing this `BigDecimal` by the given `BigDecimal` and applying the specified rounding mode. The quotient is returned as a new `BigDecimal` that has the given scale. One of the rounding constants must be specified for the rounding mode.

doubleValue

public double doubleValue()

Returns

The value of this `BigDecimal` as a `double`.

Overrides

`Number.doubleValue()`

Description

This method returns the value of this `BigDecimal` as a `double`. If the value exceeds the limits of a `double`, `Double.POSITIVE_INFINITY` or `Double.NEGATIVE_INFINITY` is returned.

equals

public boolean equals(Object x)

Parameters

`x`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `x` is an instance of `BigDecimal`, and it represents the same value as this `BigDecimal`. In order to be considered equal using this method, two `BigDecimal` objects must have the same values and scales.

floatValue

public float floatValue()

Returns

The value of this `BigDecimal` as a `float`.

Overrides

`Number.floatValue()`

Description

This method returns the value of this `BigDecimal` as a `float`. If the value exceeds the limits of a `float`, `Float.POSITIVE_INFINITY` or `Float.NEGATIVE_INFINITY` is returned.

hashCode

public int hashCode()

Returns

A hashcode for this object.

Overrides

`Object.hashCode()`

Description

This method returns a hashcode for this `BigDecimal`.

intValue

public int intValue()

Returns

The value of this `BigDecimal` as an `int`.

Overrides

`Number.intValue()`

Description

This method returns the value of this `BigDecimal` as an `int`. If the value exceeds the limits of an `int`, the excessive high-order bits are discarded. Any fractional part of this `BigDecimal` is truncated.

longValue

public long longValue()

Returns

The value of this `BigDecimal` as a `long`.

Overrides

`Number.longValue()`

Description

This method returns the value of this `BigDecimal` as a `long`. If the value exceeds the limits of a `long`, the excessive high-order bits are discarded. Any fractional part of this `BigDecimal` is also truncated.

max

public BigDecimal max(BigDecimal val)

Parameters

`val`

The number to be compared.

Returns

The `BigDecimal` that represents the greater of this number and the given value.

Description

This method returns the greater of this `BigDecimal` and the given `BigDecimal`.

min

public BigDecimal min(BigDecimal val)

Parameters

`val`

The number to be compared.

Returns

The `BigDecimal` that represents the lesser of this number and the given value.

Description

This method returns the lesser of this `BigDecimal` and the given `BigDecimal`.

movePointLeft

public BigDecimal movePointLeft(int n)

Parameters

`n`

The number of digits to move the decimal point to the left.

Returns

A new `BigDecimal` that contains the adjusted number.

Description

This method returns a `BigDecimal` that is computed by shifting the decimal point of this `BigDecimal` left by the given number of digits. If `n` is nonnegative, the value of the new `BigDecimal` is the same as the current value, and the scale is increased by `n`. If `n` is negative, the method call is equivalent to `movePointRight(-n)`.

movePointRight

public BigDecimal movePointRight(int n)

Parameters

`n`

The number of digits to move the decimal point to the right.

Returns

A new `BigDecimal` that contains the adjusted number.

Description

This method returns a `BigDecimal` that is computed by shifting the decimal point of this `BigDecimal` right by the given number of digits. If `n` is nonnegative, the value of the new `BigDecimal` is the same as the current value, and the scale is decreased by `n`. If `n` is negative, the method call is equivalent to `movePointLeft(-n)`.

multiply

public BigDecimal multiply(BigDecimal val)

Parameters

`val`

The number to be multiplied.

Returns

A new `BigDecimal` that contains the product of this number and the given value.

Description

This method multiplies this `BigDecimal` and the given `BigDecimal`, and returns the result as a new `BigDecimal`. The value of the new `BigDecimal` is the product of the values of the two `BigDecimal` objects being added; the scale is the sum of their two scales.

negate

public BigDecimal negate()

Returns

A new `BigDecimal` that contains the negative of this number.

Description

This method returns a new `BigDecimal` that is identical to this `BigDecimal` except that its sign is reversed. The scale of the new `BigDecimal` is the same as the scale of this `BigDecimal`.

scale

public int scale()

Returns

The scale of this number.

Description

This method returns the scale of this `BigDecimal`.

setScale

`public BigDecimal setScale(int scale) throws ArithmeticException, IllegalArgumentException`

Parameters

`scale`

a The new scale.

Returns

A new `BigDecimal` that is identical to this number, except that is has the given scale.

Throws

`ArithmeticException`

If the new number cannot be calculated without rounding.

`IllegalArgumentException`

This exception is never thrown.

Description

This method creates a new `BigDecimal` that has the given scale and a value that is calculated by multiplying or dividing the value of this `BigDecimal` by the appropriate power of 10 to maintain the overall value. The method is typically used to increase the scale of a number, not decrease it. It can decrease the scale, however, if there are enough zeros in the fractional part of the value to allow for rescaling without loss of precision.

Calling this method is equivalent to calling `setScale(scale, BigDecimal.ROUND_UNNECESSARY)`.

`public BigDecimal setScale(int scale, int roundingMode) throws ArithmeticException, IllegalArgumentException`

Parameters

`scale`

The new scale.

`roundingMode`

The rounding mode.

Returns

A new `BigDecimal` that contains this number adjusted to the given scale.

Throws

`ArithmeticException`

If `scale` is less than zero, or if `ROUND_UNNECESSARY` is specified for the rounding mode but rounding is necessary.

`IllegalArgumentException`

If `roundingMode` is not a valid value.

Description

This method creates a new `BigDecimal` that has the given scale and a value that is calculated by multiplying or dividing the value of this `BigDecimal` by the appropriate power of 10 to maintain the overall value. When the scale is reduced, the value must be divided, so precision may be lost. In this case, the specified rounding mode is used.

signum

public int signum()

Returns

`-1` if this number is negative, `0` if this number is zero, or `1` if this number is positive.

Description

This method returns a value that indicates the sign of this `BigDecimal`.

subtract

public BigDecimal subtract(BigDecimal val)

Parameters

`val`

The number to be subtracted.

Returns

A new `BigDecimal` that contains the result of subtracting the given number from this number.

Description

This method subtracts the given `BigDecimal` from this `BigDecimal` and returns the result as a new `BigDecimal`. The value of the new `BigDecimal` is the result of subtracting the value of the given `BigDecimal` from this `BigDecimal`; the scale is the maximum of their two scales.

toBigInteger

public BigInteger toBigInteger()

Returns

The value of this `BigDecimal` as a `BigInteger`.

Description

This method returns the value of this `BigDecimal` as a `BigInteger`. The fractional part of this number is truncated.

toString

public String toString()

Returns

A string representation of this object.

Overrides

`Object.toString()`

Description

This method returns a string representation of this `BigDecimal`. A minus sign represents the sign, and a decimal point is used to represent the scale. The mapping from digits to characters is provided by the `Character.forDigit()` method.

Inherited Methods

Method	Inherited From Method	Inherited From

`byteValue()`	`Number`	`clone()`	`Object`
`getClass()`	`Object`	`finalize()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`shortValue()`	`Number`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`ArithmeticException`, `BigInteger`, `Character`, `Double`, `Float`, `IllegalArgumentException`, `Integer`, `Long`, `Number`, `NumberFormatException`

PREVIOUS HOME NEXT
Modifier BOOK INDEX BigInteger

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

15. The java.net Package

Contents:

The package `java.net` contains classes and interfaces that provide a powerful infrastructure for networking in Java. These include:

- The `URL` class for basic access to Uniform Resource Locators (URLs).

- The URLConnection class, which supports more complex operations on URLs.

- The Socket class for connecting to particular ports on specific Internet hosts and reading and writing data using streams.

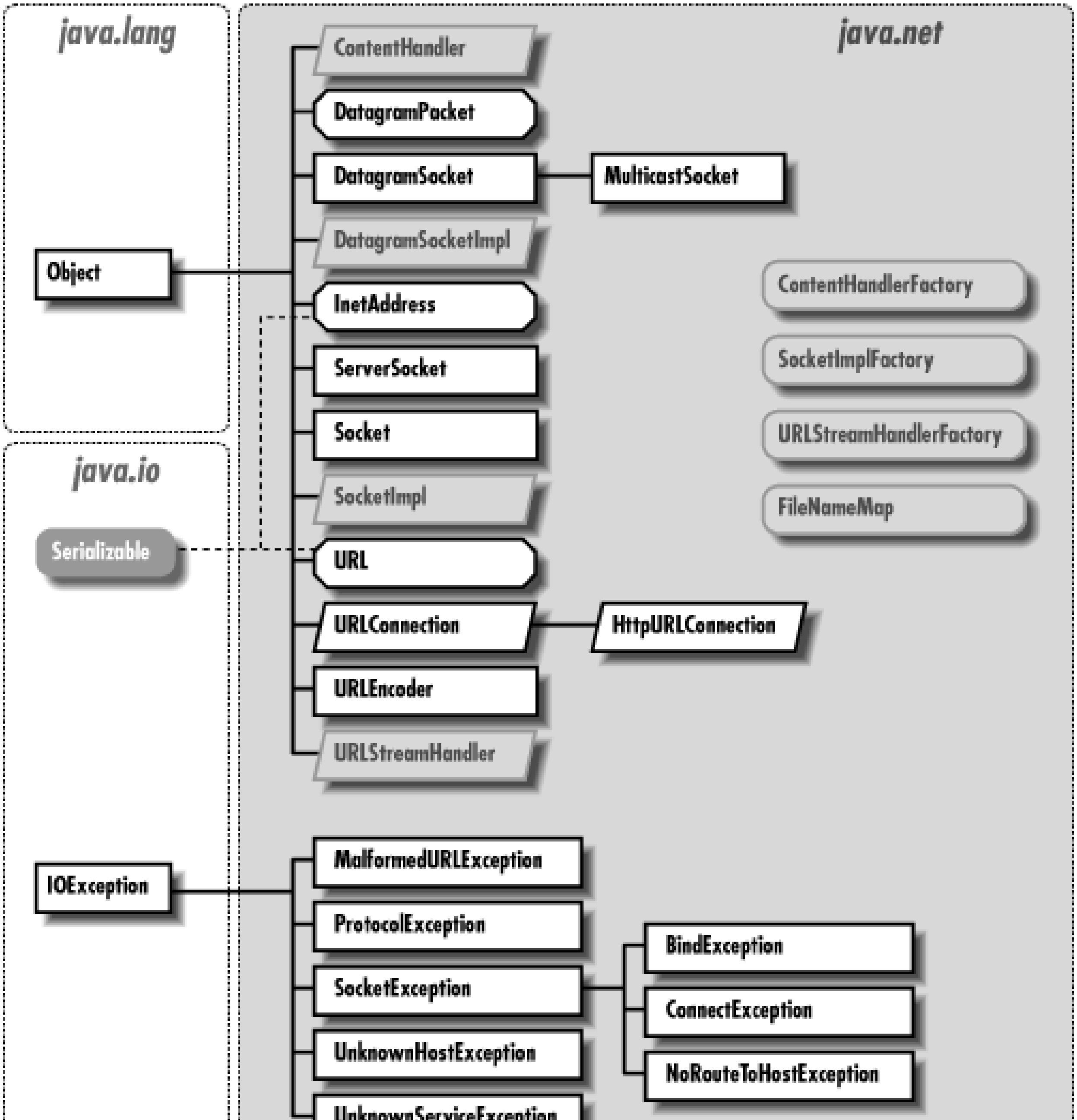
- The ServerSocket class for implementing servers that accept connections from clients.

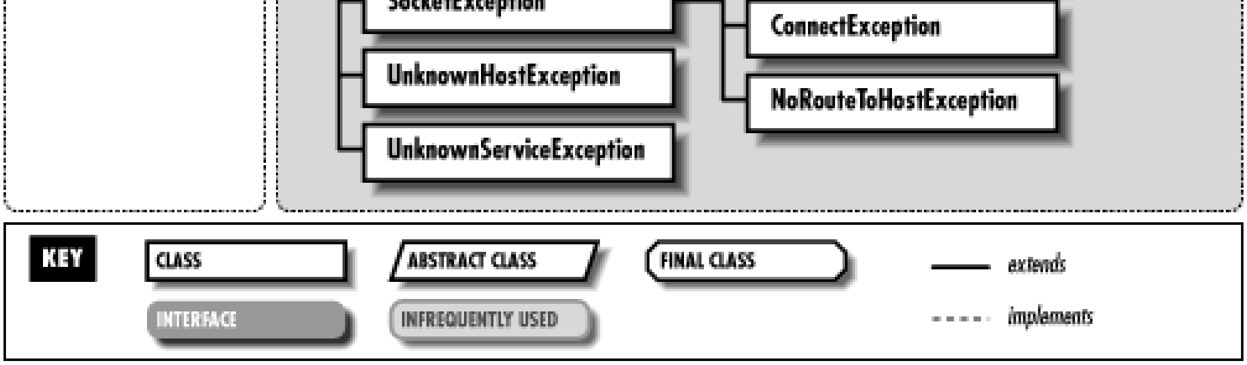
- The DatagramSocket, MulticastSocket, and DatagramPacket classes for implementing low-level networking.

- The InetAddress class, which represents Internet addresses.

Figure 15.1 shows the class hierarchy for the java.net package.

Figure 15.1: The java.net package





BindException

Name

BindException

Synopsis

Class Name:

```
java.net.BindException
```

Superclass:

```
java.net.SocketException
```

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

A `BindException` is thrown when a socket cannot be bound to a local address and port, which can occur if the port is already in use or the address is unavailable.

Class Summary

```
public class java.net.BindException extends java.net.SocketException {
  // Constructors
  public BindException();
  public BindException(String msg);
}
```

Constructors

BindException

public BindException()

Description

> This constructor `create`s a `BindException` with no associated detail message.

public BindException(String msg)

Parameters

> `msg`
>
> > The detail message.

Description

> This constructor creates a `BindException` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`

`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Throwable`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Exception`, `IOException`, `RuntimeException`, `SocketException`

PREVIOUS HOME NEXT
UnknownServiceException BOOK INDEX ConnectException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

16. The java.text Package

Contents:

The package `java.text` is new as of Java 1.1. It contains classes that support the internationalization of Java programs. The internationalization classes can be grouped as follows:

- Classes for formatting string representations of dates, times, numbers, and messages based on the conventions of a locale.
- Classes that collate strings according to the rules of a locale.

- Classes for finding boundaries in text according to the rules of a locale.

Many of the classes in `java.text` rely upon a `java.util.Locale` object to provide information about the locale that is in use.

The `Format` class is the superclass of all of the classes that generate and parse string representations of various types of data. The `DateFormat` class formats and parses dates and times according to the customs and language of a particular locale. Similarly, the `NumberFormat` class formats and parses numbers, including currency values, in a locale-dependent manner.

The `MessageFormat` class can create a textual message from a pattern string, while `ChoiceFormat` maps numerical ranges to strings. By themselves, these classes do not provide different results for different locales. However, they can be used in conjunction with `java.util.ResourceBundle` objects that generate locale-specific pattern strings.

The `Collator` class handles collating strings according to the rules of a particular locale. Different languages have different characters and different rules for sorting those characters; `Collator` and its subclass, `RuleBasedCollator`, are designed to take those differences into account when collating strings. In addition, the `CollationKey` class can be used to optimize the sorting of a large collection of strings.

The `BreakIterator` class finds various boundaries, such as word boundaries and line boundaries, in textual data. Again, `BreakIterator` locates these boundaries according to the rules of a particular locale.

Figure 16.1 shows the class hierarchy for the `java.text` package.

Figure 16.1: The java.text package

[Graphic: Figure 16-1]

BreakIterator

Name

BreakIterator

Synopsis

Class Name:

```
java.text.BreakIterator
```

Superclass:

java.lang.Object

Immediate Subclasses:

None

Interfaces Implemented:

java.lang.Cloneable, java.io.Serializable

Availability:

New as of JDK 1.1

Description

The BreakIterator class is an abstract class that defines methods that find the locations of boundaries in text, such as word boundaries and sentence boundaries. A BreakIterator operates on the object passed to its setText() method; that object must implement the CharacterIterator interface or be a String object. When a String is passed to setText(), the BreakIterator creates an internal StringCharacterIterator to iterate over the String.

When you use a BreakIterator, you call first() to get the location of the first boundary and then repeatedly call next() to iterate through the subsequent boundaries.

The BreakIterator class defines four static factory methods that return instances of BreakIterator that locate various kinds of boundaries. Each of these factory methods selects a concrete subclass of BreakIterator based either on the default locale or a specified locale. You must create a separate instance of BreakIterator to handle each kind of boundary you are trying to locate:

- getWordInstance() returns an iterator that locates word boundaries, which is useful for search-and-replace operations. A word iterator correctly handles punctuation marks.

- getSentenceInstance() returns an iterator that locates sentence boundaries, which is useful for textual selection. A sentence iterator correctly handle punctuation marks.

- getLineInstance() returns an iterator that locates line boundaries, which is useful in line wrapping. A line iterator correctly handles hyphenation and punctuation.

- `getCharacterInstance()` returns an iterator that locates boundaries between characters, which is useful for allowing the cursor to interact with characters appropriately, since some characters are stored as a base character and a diacritical mark, but only represent one display character.

Class Summary

```
public abstract class java.util.BreakIterator extends java.lang.Object
                      implements java.lang.Cloneable,
                                 java.io.Serializable {
  // Constants
  public final static int DONE;
  // Constructors
  protected BreakIterator();
  // Class Methods
  public static synchronized Locale[] getAvailableLocales();
  public static BreakIterator getCharacterInstance();
  public static BreakIterator getCharacterInstance(Locale where);
  public static BreakIterator getLineInstance();
  public static BreakIterator getLineInstance(Locale where);
  public static BreakIterator getSentenceInstance();
  public static BreakIterator getSentenceInstance(Locale where);
  public static BreakIterator getWordInstance();
  public static BreakIterator getWordInstance(Locale where);
  // Instance Methods
  public Object clone();
  public abstract int current();
  public abstract int first();
  public abstract int following(int offset);
  public abstract CharacterIterator getText();
  public abstract int last();
  public abstract int next();
  public abstract int next(int n)
  public abstract int previous();
  public abstract void setText(CharacterIterator newText);
  public void setText(String newText);
}
```

Constants

DONE

public final static int DONE

Description

> A constant that is returned by `next()` or `previous()` if there are no more breaks to be returned.

Constructors

BreakIterator

protected BreakIterator()

Description

> This constructor should be called only from constructors of subclasses.

Class Methods

getAvailableLocales

public static synchronized Locale[] getAvailableLocales()

Returns

> An array of `Locale` objects.

Description

> This method returns an array of the `Locale` objects that can be passed to `getCharacterInstance()`, `getLineInstance()`, `getSentenceInstance()`, or `getWordInstance()`.

getCharacterInstance

public static BreakIterator getCharacterInstance()

Returns

A `BreakIterator` appropriate for the default `Locale`.

Description

This method creates a `BreakIterator` that can locate character boundaries in the default `Locale`.

public static BreakIterator getCharacterInstance(Locale where)

Parameters

`where`

The `Locale` to use.

Returns

A `BreakIterator` appropriate for the given `Locale`.

Description

This method creates a `BreakIterator` that can locate character boundaries in the given `Locale`.

getLineInstance

public static BreakIterator getLineInstance()

Returns

A `BreakIterator` appropriate for the default `Locale`.

Description

This method creates a `BreakIterator` that can locate line boundaries in the default `Locale`.

public static BreakIterator getLineInstance(Locale where)

Parameters

`where`

The `Locale` to use.

Returns

A `BreakIterator` appropriate for the given `Locale`.

Description

This method creates a `BreakIterator` that can locate line boundaries in the given `Locale`.

getSentenceInstance

public static BreakIterator getSentenceInstance()

Returns

A `BreakIterator` appropriate for the default `Locale`.

Description

This method creates a `BreakIterator` that can locate sentence boundaries in the default `Locale`.

public static BreakIterator getSentenceInstance(Locale where)

Parameters

`where`

The `Locale` to use.

Returns

A `BreakIterator` appropriate for the given `Locale`.

Description

This method creates a `BreakIterator` that can locate sentence boundaries in the given `Locale`.

getWordInstance

public static BreakIterator getWordInstance()

Returns

A `BreakIterator` appropriate for the default `Locale`.

Description

This method creates a `BreakIterator` that can locate word boundaries in the default `Locale`.

public static BreakIterator getWordInstance(Locale where)

Parameters

`where`

The `Locale` to use.

Returns

A `BreakIterator` appropriate for the given `Locale`.

Description

This method creates a `BreakIterator` that can locate word boundaries in the given `Locale`.

Instance Methods

clone

public Object clone()

Returns

A copy of this `BreakIterator`.

Overrides

`Object.clone()`

Description

This method creates a copy of this `BreakIterator` and then returns it.

current

public abstract int current()

Returns

The current position of this `BreakIterator`.

Description

This method returns the current position of this `BreakIterator`. The current position is the character index of the most recently returned boundary.

first

public abstract int first()

Returns

The position of the first boundary of this `BreakIterator`.

Description

This method finds the first boundary in this `BreakIterator` and returns its character index. The current position of the iterator is set to this boundary.

following

public abstract int following(int offset)

Parameters

`offset`

An offset into this `BreakIterator`.

Returns

The position of the first boundary after the given offset of this `BreakIterator` or `DONE` if there are no more boundaries.

Throws

`IllegalArgumentException`

If `offset` is not a valid value for the `CharacterIterator` of this `BreakIterator`.

Description

This method finds the first boundary after the given offset in this `BreakIterator` and returns its character index.

getText

public abstract CharacterIterator getText()

Returns

The `CharacterIterator` that this `BreakIterator` uses.

Description

This method returns a `CharacterIterator` that represents the text this `BreakIterator` examines.

last

public abstract int last()

Returns

The position of the last boundary of this `BreakIterator`.

Description

This method finds the last boundary in this `BreakIterator` and returns its character index. The current position of the iterator is set to this boundary.

next

public abstract int next()

Returns

The position of the next boundary of this `BreakIterator` or `DONE` if there are no more boundaries.

Description

This method finds the next boundary in this `BreakIterator` after the current position and returns its character index. The current position of the iterator is set to this boundary.

public abstract int next(int n)

Parameters

`n`

The boundary to return. A positive value moves to a later boundary a negative value moves to a previous boundary; the value 0 does nothing.

Returns

The position of the requested boundary of this `BreakIterator`.

Description

This method finds the `n`th boundary in this `BreakIterator`, starting from the current position, and returns its character index. The current position of the iterator is set to this boundary.

For example, `next(-2)` finds the third previous boundary. Thus `next(1)` is equivalent to `next()`, `next(-1)` is equivalent to `previous()`, and `next(0)` does nothing.

previous

public abstract int previous()

Returns

The position of the previous boundary of this `BreakIterator`.

Description

This method finds the previous boundary in this `BreakIterator`, starting from the current position, and returns its character index. The current position of the iterator is set to this boundary.

setText

public abstract void setText(CharacterIterator newText)

Parameters

`newText`

The `CharacterIterator` that contains the text to be examined.

Description

This method tells this `BreakIterator` to examine the piece of text specified by the `CharacterIterator`. This current position of this `BreakIterator` is set to `first()`.

public void setText(String newText)

Parameters

`newText`

The `String` that contains the text to be examined.

Description

This method tells this `BreakIterator` to examine the piece of text specified by the `String`, using a `StringCharacterIterator` created from the given string. This current position of this `BreakIterator` is set to `first()`.

Inherited Methods

Method	Inherited From	Method	Inherited From
`equals(Object)`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`toString()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`CharacterIterator`, `Locale`, `String`, `StringCharacterIterator`

PREVIOUS | HOME | NEXT
UnknownServiceException | BOOK INDEX | CharacterIterator

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

17. The java.util Package

Contents:

The package `java.util` contains a number of useful classes and interfaces. Although the name of the package might imply that these are utility classes, they are really more important than that. In fact, Java depends directly on several of the classes in this package, and many programs will find these classes indispensable. The classes and interfaces in `java.util` include:

- The `Hashtable` class for implementing hashtables, or associative arrays.
- The `Vector` class, which supports variable-length arrays.
- The `Enumeration` interface for iterating through a collection of elements.
- The `StringTokenizer` class for parsing strings into distinct tokens separated by delimiter characters.
- The `EventObject` class and the `EventListener` interface, which form the basis of the new AWT event model in Java 1.1.
- The `Locale` class in Java 1.1, which represents a particular locale for internationalization purposes.
- The `Calendar` and `TimeZone` classes in Java. These classes interpret the value of a `Date` object in the context of a particular calendar system.
- The `ResourceBundle` class and its subclasses, `ListResourceBundle` and `PropertyResourceBundle`, which represent sets of localized data in Java 1.1.

Figure 17.1 shows the class hierarchy for the `java.util` package.

Figure 17.1: The java.util package

[Graphic: Figure 17-1]

BitSet

Name

Synopsis

Class Name:

`java.util.BitSet`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

`java.lang.Cloneable`, `java.io.Serializable`

Availability:

JDK 1.0 or later

Description

The `BitSet` class implements a set of bits. The set grows in size as needed. Each element of a `BitSet` has a boolean value. When a `BitSet` object is created, all of the bits are set to `false` by default. The bits in a `BitSet` are indexed by nonnegative integers, starting at 0. The size of a `BitSet` is the number of bits that it currently contains. The `BitSet` class provides methods to set, clear, and retrieve the values of the individual bits in a `BitSet`. There are also methods to perform logical AND, OR, and XOR operations.

Class Summary

```
public final class java.util.BitSet extends java.lang.Object
                   implements java.lang.Cloneable, java.io.Serializable {
  // Constructors
  public BitSet();
  public BitSet(int nbits);
  // Instance Methods
  public void and(BitSet set);
  public void clear(int bit);
  public Object clone();
  public boolean equals(Object obj);
  public boolean get(int bit);
  public int hashCode();
  public void or(BitSet set);
  public void set(int bit);
  public int size();
  public String toString();
  public void xor(BitSet set);
}
```

Constructors

BitSet

public BitSet()

Description

> This constructor creates `a BitSet` with a default size of 64 bits. All of the bits in the `BitSet` are initially set to `false`.

public BitSet(int nbits)

Parameters

> `nbits`
>
> > The initial number of bits.

Description

> This constructor creates a `BitSet` with a size of `nbits`. All of the bits in the `BitSet` are initially set to `false`.

Instance Methods

and

public void and(BitSet set)

Parameters

set

The BitSet to AND with this BitSet.

Description

This method computes the logical AND of this BitSet and the specified BitSet and stores the result in this BitSet. In other words, for each bit in this BitSet, the value is set to only true if the bit is already true in this BitSet and the corresponding bit in set is true.

If the size of set is greater than the size of this BitSet, the extra bits in set are ignored. If the size of set is less than the size of this BitSet, the extra bits in this BitSet are set to false.

clear

public void clear(int bit)

Parameters

bit

The index of the bit to clear.

Description

This method sets the bit at the given index to false. If bit is greater than or equal to the number of bits in the BitSet, the size of the BitSet is increased so that it contains bit values. All of the additional bits are set to false.

clone

public Object clone()

Returns

A copy of this BitSet.

Overrides

`Object.clone()`

Description

This method creates a copy of this `BitSet` and returns it. In other words, the returned `BitSet` has the same size as this `BitSet`, and it has the same bits set to `true`.

equals

public boolean equals(Object obj)

Parameters

`obj`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `obj` is an instance of `BitSet` and it contains the same bit values as the object this method is associated with. In other words, this method compares each bit of this `BitSet` with the corresponding bit of `obj`. If any bits do not match, the method returns `false`. If the size of this `BitSet` is different than `obj`, the extra bits in either this `BitSet` or in `obj` must be `false` for this method to return `true`.

get

public boolean get(int bit)

Parameters

`bit`

The index of the bit to retrieve.

Returns

The `boolean` value of the bit at the given index.

Description

This method returns the value of the given bit. If `bit` is greater than or equal to the number of bits in the `BitSet`, the method returns `false`.

hashCode

public int hashCode()

Returns

The hashcode for this `BitSet`.

Overrides

`Object.hashCode()`

Description

This method returns a hashcode for this object.

or

public void or(BitSet set)

Parameters

`set`

The `BitSet` to OR with this `BitSet`.

Description

This method computes the logical OR of this `BitSet` and the specified `BitSet`, and stores the result in this `BitSet`. In other words, for each bit in this `BitSet`, the value is set to `true` if the bit is already `true` in this `BitSet` or the corresponding bit in `set` is `true`.

If the size of `set` is greater than the size of this `BitSet`, this `BitSet` is first increased in size to accommodate the additional bits. All of the additional bits are initially set to `false`.

set

public void set(int bit)

Parameters

`bit`

The index of the bit to set.

Description

This method sets the bit at the given index to `true`. If `bit` is greater than or equal to the number of bits in the `BitSet`, the size of the `BitSet` is increased so that it contains `bit` values. All of the additional bits except the last one are set to `false`.

size

public int size()

Returns

The size of this `BitSet`.

Description

This method returns the size of this `BitSet`, which is the number of bits currently in the set.

toString

public String toString()

Returns

A string representation of this `BitSet`.

Overrides

`Object.toString()`

Description

This method returns a string representation of this `BitSet`. The string lists the indexes of all the bits in the `BitSet` that are `true`.

xor

public void xor(BitSet set)

Parameters

`set`

The `BitSet` to XOR with this `BitSet`.

Description

This method computes the logical XOR (exclusive OR) of this `BitSet` and the specified `BitSet` and stores the result in this `BitSet`. In other words, for each bit in this `BitSet`, the value is set to `true` only if the bit is already `true` in this `BitSet`, and the corresponding bit in `set` is `false`, or if the bit is `false` in this `BitSet` and the corresponding bit in `set` is `true`.

If the size of `set` is greater than the size of this `BitSet`, this `BitSet` is first increased in size to accommodate the additional bits. All of the additional bits are initially set to `false`.

Inherited Methods

Method	Inherited From	Method	Inherited From
`finalize()`	`Object`	`getClass()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Cloneable`, `Serializable`

PREVIOUS HOME NEXT
Vector BOOK INDEX Calendar

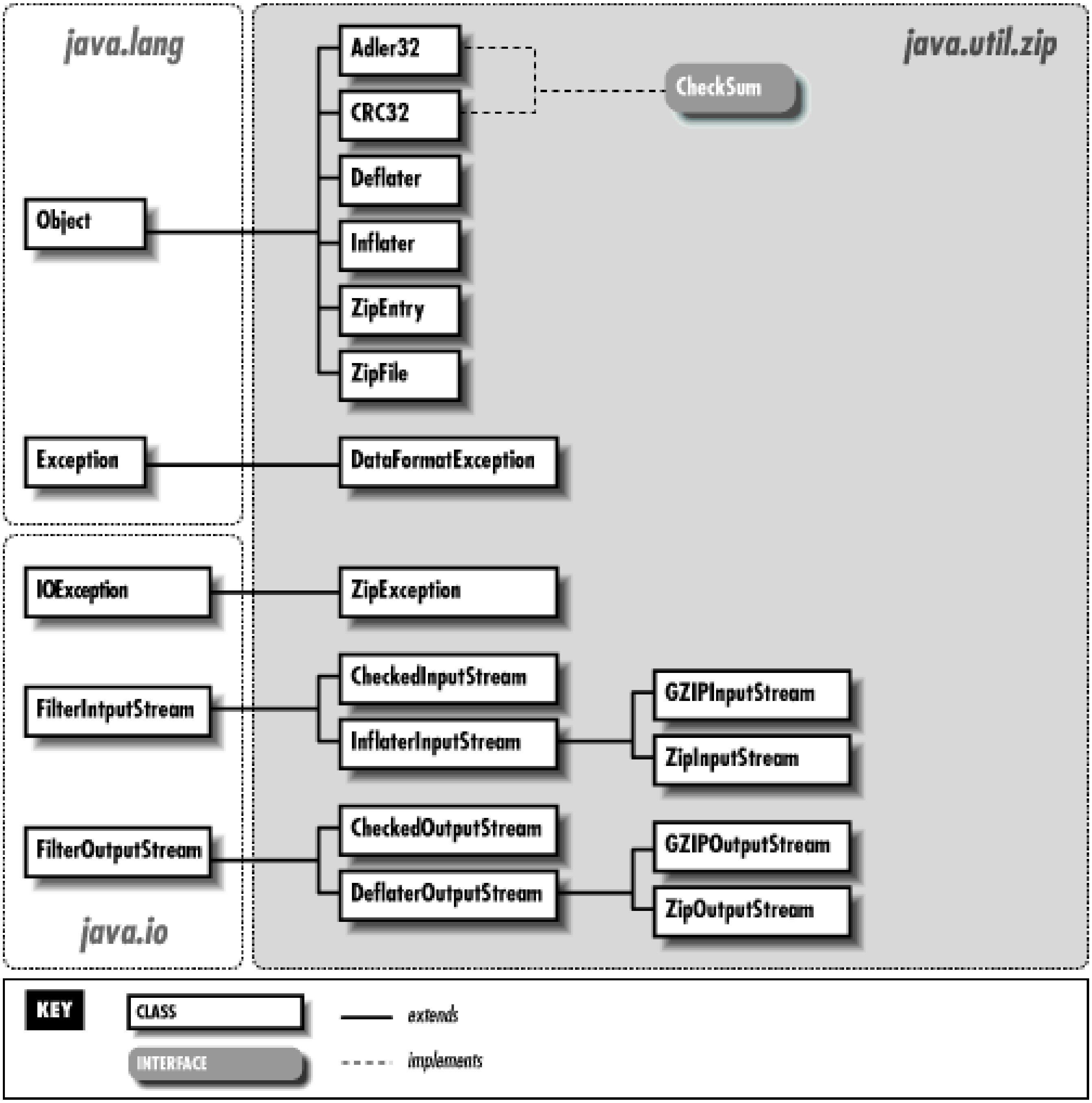
JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

18. The java.util.zip Package

Contents:

The package `java.util.zip` is new as of Java 1.1. It contains classes that provide support for general-purpose data compression and decompression using the ZLIB compression algorithms. The important classes in `java.util.zip` are those that provide the means to read and write data that is compatible with the popular GZIP and ZIP formats: `GZIPInputStream`, `GZIPOutputStream`, `ZipInputStream`, and `ZipOutputStream`. Figure 18.1 shows the class hierarchy for the `java.util.zip` package.

Figure 18.1: The java.text package



It is easy to use the GZIP and ZIP classes because they subclass `java.io.FilterInputStream` and `java.io.FilterOutputStream`. For example, to decompress GZIP data, you simply create a `GZIPInputStream` around the input stream that represents the compressed data. As with any `InputStream`, you could be reading from a file, a socket, or some other data source. You can then read decompressed data by calling the `read()` methods of the `GZIPInputStream`. The following code fragment creates a `GZIPInputStream` that reads data from the file *sample.gz* :

```
FileInputStream inFile;
try {
    inFile = new FileInputStream("sample.gz");
} catch (IOException e) {
    System.out.println("Couldn't open file.");
    return;
}
GZIPInputStream in = new GZIPInputStream(inFile);
// Now use in.read() to get decompressed data.
```

Similarly, you can compress data using the GZIP format by creating a `GZIPOutputStream` around an output stream and using the `write()` methods of `GZIPOutputStream`. The following code fragment creates a `GZIPOutputStream` that writes data to the file *sample.gz* :

```
FileOutputStream outFile;
try {
    outFile = new FileOutputStream("sample.gz");
} catch (IOException e) {
    System.out.println("Couldn't open file.");
    return;
}
GZIPOutputStream out = new GZIPOutputStream(outFile);
// Now use out.write() to write compressed data.
```

A ZIP file, or archive, is not quite as easy to use because it may contain more than one compressed file. A `ZipEntry` object represents each compressed file in the archive. When you are reading from a `ZipInputStream`, you must first call `getNextEntry()` to access an entry, and then you can read decompressed data from the stream, just like with a `GZIPInputStream`. When you are writing data to a `ZipOutputStream`, use `putNextEntry()` before you start writing each entry in the archive. The `ZipFile` class is provided as a convenience for reading an archive; it allows nonsequential access to the entries in a ZIP file.

The remainder of the classes in `java.util.zip` exist to support the GZIP and ZIP classes. The generic `Deflater` and `Inflater` classes implement the ZLIB algorithms; they are used by `DeflaterOutputStream` and `InflaterInputStream` to decompress and compress data. The `Checksum` interface and the classes that implement it, `Adler32` and `CRC32`, define algorithms that generate checksums from stream data. These checksums are used by the `CheckedInputStream` and `CheckedOutputStream` classes.

Adler32

Name

Adler32

Synopsis

Class Name:

`java.util.zip.Adler32`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

`java.util.zip.Checksum`

Availability:

New as of JDK 1.1

Description

The `Adler32` class implements the `Checksum` interface using the Adler-32 algorithm. This algorithm is significantly faster than CRC-32 and almost as reliable.

Class Summary

```
public class java.util.zip.Adler32 extends java.lang.Object
             implements java.util.zip.Checksum {
  // Constructors
  public Adler32();

  // Instance Methods
  public long getValue();
  public void reset();
  public void update(int b);
  public void update(byte[] b);
  public native void update(byte[] b, int off, int len);
}
```

Constructors

Adler32

public Adler32()

Description

> This constructor creates an `Adler32` object.

Instance Methods

getValue

public long getValue()

Returns

> The current checksum value.

Implements

> `Checksum.getValue()`

Description

> This method returns the current value of this checksum.

reset

public void reset()

Implements

> `Checksum.reset()`

Description

This method resets the checksum to its initial value, making it appear as though it has not been updated by any data.

update

public void update(int b)

Parameters

`b`

The value to be added to the data stream for the checksum calculation.

Implements

`Checksum.update(int)`

Description

This method adds the specified value to the data stream and updates the checksum value. The method uses only the lowest eight bits of the given `int`.

public void update(byte[] b)

Parameters

`b`

An array of bytes to be added to the data stream for the checksum calculation.

Description

This method adds the bytes from the specified array to the data stream and updates the checksum value.

public native void update(byte[] b, int off, int len)

Parameters

`b`

An array of bytes to be added to the data stream for the checksum calculation.

`off`

An offset into the byte array.

`len`

The number of bytes to use.

Implements

`Checksum.update(byte[], int, int)`

Description

This method adds `len` bytes from the specified array, starting at `off`, to the data stream and updates the checksum value.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Checksum`, `CRC32`

A. The Unicode 2.0 Character Set

Characters	Description
\u0000 - \u1FFF	Alphabets
\u0020 - \u007F	Basic Latin
\u0080 - \u00FF	Latin-1 supplement
\u0100 - \u017F	Latin extended-A
\u0180 - \u024F	Latin extended-B
\u0250 - \u02AF	IPA extensions
\u02B0 - \u02FF	Spacing modifier letters
\u0300 - \u036F	Combining diacritical marks
\u0370 - \u03FF	Greek
\u0400 - \u04FF	Cyrillic
\u0530 - \u058F	Armenian
\u0590 - \u05FF	Hebrew
\u0600 - \u06FF	Arabic
\u0900 - \u097F	Devanagari
\u0980 - \u09FF	Bengali
\u0A00 - \u0A7F	Gurmukhi
\u0A80 - \u0AFF	Gujarati
\u0B00 - \u0B7F	Oriya
\u0B80 - \u0BFF	Tamil
\u0C00 - \u0C7F	Telugu
\u0C80 - \u0CFF	Kannada
\u0D00 - \u0D7F	Malayalam

\u0E00 - \u0E7F Thai

\u0E80 - \u0EFF Lao

\u0F00 - \u0FBF Tibetan

\u10A0 - \u10FF Georgian

\u1100 - \u11FF Hangul Jamo

\u1E00 - \u1EFF Latin extended additional

\u1F00 - \u1FFF Greek extended

\u2000 - \u2FFF Symbols and punctuation

\u2000 - \u206F General punctuation

\u2070 - \u209F Superscripts and subscripts

\u20A0 - \u20CF Currency symbols

\u20D0 - \u20FF Combining diacritical marks for symbols

\u2100 - \u214F Letterlike symbols

\u2150 - \u218F Number forms

\u2190 - \u21FF Arrows

\u2200 - \u22FF Mathematical operators

\u2300 - \u23FF Miscellaneous technical

\u2400 - \u243F Control pictures

\u2440 - \u245F Optical character recognition

\u2460 - \u24FF Enclosed alphanumerics

\u2500 - \u257F Box drawing

\u2580 - \u259F Block elements

\u25A0 - \u25FF Geometric shapes

\u2600 - \u26FF Miscellaneous symbols

\u2700 - \u27BF Dingbats

\u3000 - \u33FF CJK auxiliary

\u3000 - \u303F CJK symbols and punctuation

\u3040 - \u309F Hiragana

\u30A0 - \u30FF Katakana

\u3100 - \u312F Bopomofo

\u3130 - \u318F Hangul compatibility Jamo

\u3190 - \u319F Kanbun

\u3200 - \u32FF Enclosed CJK letters and months

\u3300 - \u33FF CJK compatibility

\u4E00 - \u9FFF	CJK unified ideographs: Han characters used in China, Japan, Korea, Taiwan, and Vietnam
\uAC00 - \uD7A3	Hangul syllables
\uD800 - \uDFFF	Surrogates
\uD800 - \uDB7F	High surrogates
\uDB80 - \uDBFF	High private use surrogates
\uDC00 - \uDFFF	Low surrogates
\uE000 - \uF8FF	Private use
\uF900 - \uFFFF	Miscellaneous
\uF900 - \uFAFF	CJK compatibility ideographs
\uFB00 - \uFB4F	Alphabetic presentation forms
\uFB50 - \uFDFF	Arabic presentation forms-A
\uFE20 - \uFE2F	Combing half marks
\uFE30 - \uFE4F	CJK compatibility forms
\uFE50 - \uFE6F	Small form variants
\uFE70 - \uFEFE	Arabic presentation forms-B
\uFEFF	Specials
\uFF00 - \uFFEF	Halfwidth and fullwidth forms
\uFFF0 - \uFFFF	Specials

Symbols and Numbers

Preface

Contents:

The Abstract Window Tookit (AWT) provides the user interface for Java programs. Unless you want to construct your own GUI or use a crude text-only interface, the AWT provides the tools you will use to communicate with the user. Although we are beginning to see some other APIs for building user interfaces, like Netscape's IFC (Internet Foundation Classes), those alternative APIs will not be in widespread use for some time, and some will be platform specific. Likewise, we are beginning to see automated tools for building GUIs in Java; Sun's JavaBeans effort promises to make such tools much more widespread. (In fact, the biggest changes in Java 1.1 prepare the way for using the various AWT components as JavaBeans.) However, even with automated tools and JavaBeans in the future, an in-depth knowledge of AWT is essential for the practicing Java programmer.

The major problem facing Java developers these days is that AWT is a moving target. Java 1.0.2 is being replaced by Java 1.1, with many significant new features. Java 1.1 was released on February 18, 1997, but it isn't clear how long it will take for 1.1 to be accepted in the market. The problem facing developers is not just learning about the new features and changes in Java 1.1, but also knowing when they can afford to use these new features in their code. In practice, this boils down to one question: when will Netscape Navigator support Java 1.1? Rumor has it that the answer is "as soon as possible"--and we all hope this rumor is correct. But given the realities of maintaining a very complex piece of software, and the fact that Netscape is currently in the beta process for Navigator 4.0, there's a possibility that "as soon as possible" and "soon" aren't the same thing. In other words, you should expect Java 1.0.2 to stick around for a while, especially since Web users won't all replace their browsers as soon as Navigator has 1.1 support.

This state of affairs raises obvious problems for my book. Nothing would have made me happier than to write a book that covered AWT 1.1 only. It would be significantly shorter, for one thing, and I wouldn't have to spend so much effort pointing out which features are present in which release. But that's not the current reality. For the time being, programmers still need to know about 1.0.2. Therefore, this book covers both releases thoroughly. There are many examples using 1.0.2; many more examples that require 1.1; and more examples showing you how to update 1.0.2 code to use 1.1's features.

Sun has done a good job of maintaining compatibility between versions: 1.0 code runs under Java 1.1, with very few exceptions. All of the 1.0 examples in this book have been tested under Java 1.1. However, Java 1.1--and particularly, AWT 1.1--offer many advantages over older releases. If nothing else, I hope this book convinces you that you should be looking forward to the day when you can forget about writing code for Java 1.0.2.

New Features of AWT in Java 1.1

Having spent all this time talking about 1.0.2 and 1.1 and the transitional state we're currently in and having alluded briefly to the advantages of Java 1.1, you deserve a brief summary of what has changed. Of course, you'll find the details in the book.

Improved event handling

> Java 1.1 provides a completely new event model. Instead of propagating events to all objects that might possibly have an interest, objects in Java 1.1 register their interest in particular kinds of events and get only the events they're interested in hearing. The old event model is still supported, but the new model is much more efficient.
>
> The new event model is also important in the context of JavaBeans. The old events were pretty much specific to AWT. The new model has been designed as a general purpose feature for communication between software components. Unfortunately, how to use events in this more general sense is beyond the scope of this book, but you should be aware that it's possible.

New components and containers

> Java 1.1 provides one new component, the `PopupMenu`, and one new container, the `ScrollPane`. Pop-up menus are a staple of modern user interfaces; providing them fixes a serious omission. `ScrollPane` makes it trivial to implement scrolling; in Java 1.0, you had to do scrolling "by hand." In Java 1.1, you also get menu shortcuts (i.e., the ability to select menu items using the keyboard), another standard feature of modern user interfaces.

Java 1.1 also introduces a `LightweightPeer`, which means that it is possible to create "lightweight components." To do so, you subclass `Component` or `Container` directly; this wasn't possible in earlier releases. For simple operations, lightweight components are much more efficient than full-fledged components.

Clipboards

Java 1.1 lets you read from and write to the system clipboard and create private clipboards for use by your programs. The clipboard facility is a down payment on a larger data transfer facility, which will support drag and drop. (No promises about when drag and drop will appear.)

Printing

Java 1.1 gives components the ability to print.

The rest

There are many other new features, including more flexible use of cursors; the ability to use system color schemes, and thus make your program look like other software in the run-time environment; more image filters to play with; and the ability to prescale an image.

Deprecated Methods and JavaBeans

One of the biggest changes in Java 1.1 doesn't concern the feature set at all. This was the addition of many new methods that differ from a method of Java 1.0 in name only. There are hundreds of these, particularly in AWT. The new method names show an important future direction for the AWT package (in fact, all of Java). The new names obey the naming conventions used by JavaBeans, which means that all AWT classes are potentially Beans. These conventions make it possible for an application builder to analyze what a component does based on its public methods. For example, the method `setFont()` changes the value of the component's `Font` property. In turn, this means that you will eventually be able to build user interfaces and, in some cases, entire applications, inside some other tool, without writing any Java code at all. An application builder will be able to find out what it needs to know about any component by looking at the component itself, and letting you customize the component and its interactions with others.

Comments in the JDK source code indicate that the older method names have been "deprecated," which means that you should consider the old names obsolete and avoid using them; they could disappear in a future release.

Reworking AWT to comply with JavaBeans is both necessary and inevitable. Furthermore, it's a good idea to get into the habit of following the same conventions for your own code; the advantages of

JavaBeans are much greater than the inconvenience of changing your coding style.

Other Changes in Java

Other new features are scattered throughout the rest of the Java classes, most notably, improvements in the networking and I/O packages and support for internationalization. Some new features were added to the language itself, of which the most important is "inner classes." For the most part, I don't discuss these changes; in fact, I stay away from them and base non-AWT code on the 1.0.2. release. Though these changes are important, covering the new material in AWT is enough for one book. If I used a new feature at this point, I would feel that I owed you an explanation, and this book is already long enough. A future edition will update the code so that it doesn't rely on any older features.

HOME
BOOK INDEX

NEXT
What This Book Covers

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

1. Abstract Window Toolkit Overview

Contents:

For years, programmers have had to go through the hassles of porting software from BSD-based UNIX to System V Release 4-based UNIX, from OpenWindows to Motif, from PC to UNIX to Macintosh (or some combination thereof), and between various other alternatives, too numerous to mention. Getting an application to work was only part of the problem; you also had to port it to all the platforms you supported, which often took more time than the development effort itself. In the UNIX world, standards like POSIX and X made it easier to move applications between different UNIX platforms. But they only solved part of the problem and didn't provide any help with the PC world. Portability became even more important as the Internet grew. The goal was clear: wouldn't it be great if you could just move applications between different operating environments without worrying about the software breaking because of a different operating system, windowing environment, or internal data representation?

In the spring of 1995, Sun Microsystems announced Java, which claimed to solve this dilemma. What started out as a dancing penguin (or Star Trek communicator) named Duke on remote controls for interactive television has become a new paradigm for programming on the Internet. With Java, you can create a program on one platform and deliver the compilation output (byte-codes/class files) to every other supported environment without recompiling or worrying about the local windowing environment, word size, or byte order. The first generation of Java programs consisted mostly of fancy animation applets that ran in a web browser like Netscape Navigator, Internet Explorer, or HotJava. We're beginning to see the next generation now: powerful distributed applications in areas ranging from commerce to medical imaging to network management. All of these applications require extreme portability: Joe's Online Bait Shop doesn't have the time or energy to port its "Online Bait Buyer" program to every platform on the Internet but doesn't want to limit its market to a specific platform. Java neatly solves their problem.

Windowing systems present the biggest challenges for portability. When you move an application from Windows to the Macintosh, you may be able to salvage most of the computational guts, but you'll have to rewrite the window interface code completely. In Java, this part of the portability challenge is addressed by a package called AWT, which stands for Abstract Window Toolkit (although people have come up with many other expansions). AWT provides the magic of maintaining the local look and feel of the user's environment. Because of AWT, the same application program can look appropriate in any environment. For example, if your program uses a pull-down list, that list will look like a Windows list when you run the program under Windows; a Macintosh list when you run the program on a Mac; and a Motif list when you run the program on a UNIX system under Motif. The same code works on all platforms. In addition to providing a common set of user interface components, AWT provides facilities for manipulating images and generating graphics.

This book is a complete programmer's guide and reference to the `java.awt` package (including `java.awt.image`, `java.awt.event`, `java.awt.datatransfer`, and `java.awt.peer`). It assumes that you're already familiar with the Java language and class libraries. If you aren't, *Exploring Java*, by Pat Niemeyer and Josh Peck, provides a general introduction, and other books in the O'Reilly Java series provide detailed references and tutorials on specific topics. This chapter provides a quick overview of AWT: it introduces you to the various GUI elements contained within the `java.awt` package and gives you pointers to the chapters that provide more specific information about each component. If you're interested in some of the more advanced image manipulation capabilities, head right to Chapter 12, *Image Processing*. The book ends with a reference section that summarizes what you need to know about every class in AWT.

In using this book, you should be aware that it covers two versions of AWT: 1.0.2 and 1.1. The Java 1.1 JDK (Java Developer's Kit) occurred in December 1996. This release includes many improvements and additions to AWT and is a major step forward in Java's overall functionality. It would be nice if I could say, "Forget about 1.0.2, it's obsolete--use this book to learn 1.1." However, I can't; at this point, since browsers (Netscape Navigator in particular) still incorporate 1.0.2, and we have no idea when they will incorporate the new release. As of publication, Navigator 4.0 is in beta test and incorporates 1.0.2. Therefore, Java release 1.0.2 will continue to be important, at least for the foreseeable future.

In this summary, we'll point out new features of Java 1.1 as they come up. However, one feature deserves mention and doesn't fit naturally into an overview. Many of the methods of Java 1.0.2 have been renamed in Java 1.1. The old names still work but are "deprecated." The new names adhere strictly to the design patterns discussed in the JavaBeans documentation:[1] all methods that retrieve the value of an object's property begin with "get," all methods that set the value of a property begin with "set," and all methods that test the value of some property begin with "is." For example, the `size()` method is now called `getSize()`. The Java 1.1 compiler issues warnings whenever you used a deprecated method name.

[1] http://splash.javasoft.com/beans/spec.html

1.1 Components

Modern user interfaces are built around the idea of "components": reusable gadgets that implement a specific part of the interface. They don't need much introduction: if you have used a computer since 1985 or so, you're already familiar with buttons, menus, windows, checkboxes, scrollbars, and many other similar items. AWT comes with a repertoire of basic user interface components, along with the machinery for creating your own components (often combinations of the basic components) and for communicating between components and the rest of the program.

The next few sections summarize the components that are part of AWT. If you're new to AWT, you may find it helpful to familiarize yourself with what's available before jumping into the more detailed discussions later in this book.

Static Text

The `Label` class provides a means to display a single line of text on the screen. That's about it. They provide visual aids to the user: for example, you might use a label to describe an input field. You have control over the size, font, and color of the text. Labels are discussed in Labels. Figure 1.1 displays several labels with different attributes.

Figure 1.1: Multiple Label instances

[Graphic: Figure 1-1]

User Input

Java provides several different ways for a user to provide input to an application. The user can type the information or select it from a preset list of available choices. The choice depends primarily on the desired functionality of the program, the user-base, and the amount of back-end processing that you want to do.

The TextField and TextArea classes

Two components are available for entering keyboard input: `TextField` for single line input and `TextArea` for multi-line input. They provide the means to do things from character-level data validation to complex text editing. These are discussed in much more detail in Chapter 8, *Input Fields*. Figure 1.2 shows a screen that contains various `TextField` and `TextArea` components.

Figure 1.2: TextField and TextArea elements

[Graphic: Figure 1-2]

The Checkbox and CheckboxGroup classes

The remaining input-oriented components provide mechanisms for letting the user select from a list of choices. The first such mechanism is `Checkbox`, which lets you select or deselect an option. The left side of the applet in Figure 1.3 shows a checkbox for a Dialog option. Clicking on the box selects the option and makes the box change appearance. A second click deselects the option.

The `CheckboxGroup` class is not a component; it provides a means for grouping checkboxes into a mutual exclusion set, often called a set of radio buttons. Selecting any button in the group automatically deselects the other buttons. This behavior is useful for a set of mutually exclusive choices. For example, the right side of the applet in Figure 1.3 shows a set of checkboxes for selecting a font. It makes sense to select only one font at a time, so these checkboxes have been put in a `CheckboxGroup`.

Figure 1.3: Examples of Checkbox and CheckboxGroup

[Graphic: Figure 1-3]

The appearance of a checkbox varies from platform to platform. On the left, Figure 1.3 shows Windows; the right shows Motif. On most platforms, the appearance also changes when a checkbox is put into a `CheckboxGroup`.

The Choice class

`Checkbox` and `CheckboxGroup` present a problem when the list of choices becomes long. Every element of a `CheckboxGroup` uses precious screen real estate, which limits the amount of space available for other components. The `Choice` class was designed to use screen space more efficiently. When a `Choice` element is displayed on the screen, it takes up the space of a single item in the list, along with some extra space for decorations. This leaves more space for other components. When the user selects a `Choice` component, it displays the available options next to or below the `Choice`. Once the user makes a selection, the choices are removed from the screen, and the `Choice` displays the selection. At any time, only one item in a `Choice` may be selected, so selecting an item implicitly deselects everything else. Choice explores the details of the `Choice` class. Figure 1.4 shows examples of open (on the right of the screens) and closed (on the left) `Choice` items in Windows 95 and Motif.

Figure 1.4: Open and closed Choice items

[Graphic: Figure 1-4]

The List class

Somewhere between `Choice` and `CheckboxGroup` in the screen real estate business is a component called `List`. With a `List`, the user is still able to select any item. However, the programmer recommends how many items to display on the screen at once. All additional choices are still available, but the user moves an attached scrollbar to access them. Unlike a `Choice`, a `List` allows the user to select multiple items. Section 9.2 covers the `List` component. Figure 1.5 shows `List` components in different states.

Figure 1.5: List components in different states

[Graphic: Figure 1-5]

Menus

Most modern user interfaces use menus heavily; therefore, it's no surprise that Java supports menus. As you'd expect, Java menus look like the menus in the windowing environment under which the program runs. Currently, menus can only appear within a `Frame`, although this will probably change in the future. A `Menu` is a fairly complex object, with lots of moving parts: menu bars, menu items, etc. Java 1.1 adds hot keys to menus, allowing users to navigate a menu interface using keyboard shortcuts. The details of `Menu` are explored in Chapter 10, *Would You Like to Choose from the Menu?* Figure 1.6 shows frames with open menus for both Windows and Motif. Since tear-off menus are available on Motif systems, its menus look and act a little differently. Figure 1.6 also includes a tear-off menu. The shortcuts (Ctrl+F8) are newly supported in Java 1.1.

Figure 1.6: Examples of menus

[Graphic: Figure 1-6]

The PopupMenu class

The `PopupMenu` class is new to Java 1.1. Pop-up menus can be used for context-sensitive, component-level menus. Associated with each `Component` can be its own pop-up menu. The details of creating and working with the `PopupMenu` class and the fun time you have catching their events are covered in Chapter 10, *Would You Like to Choose from the Menu?*, *Would You Like to Choose from the Menu?* Figure 1.7 shows an example of a pop-up menu.

Figure 1.7: A Pop-up menu

[Graphic: Figure 1-7]

Event Triggers

Java provides two components whose sole purpose is to trigger actions on the screen: `Button` and `Scrollbar`. They provide the means for users to signal that they are ready to perform an operation. (Note that all components except labels generate events; I'm singling out buttons and scrollbars because their only purpose is to generate events.)

The Scrollbar class

Most people are familiar with scrollbars. In a word processor or a web browser, when an image or document is too large to fit on the screen, the scrollbar allows the user to move to another area. With Java, the `Scrollbar` performs similarly. Selecting or moving the scrollbar triggers an event that allows the program to process the scrollbar movement and respond accordingly. The details of the `Scrollbar` are covered in Scrollbar. Figure 1.8 shows horizontal and vertical scrollbars.

Figure 1.8: Horizontal and vertical scrollbars

[Graphic: Figure 1-8]

Note that a scrollbar is just that. It generates events when the user adjusts it, but the program using the scrollbar is responsible for figuring out what to do with the events, such as displaying a different part of an image or the text, etc. Several of the components we've discussed, like `TextArea` and `List`, have built-in scrollbars, saving you the trouble of writing your own code to do the actual scrolling. Java 1.1 has a new container called a `ScrollPane` that has scrolling built in. By using a scroll pane, you should be able to avoid using scroll bars as a positioning mechanism. An example of `ScrollPane` appears later in this chapter.

The Button class

A button is little more than a label that you can click on. Selecting a button triggers an event telling the program to go to work. Buttons explores the `Button` component. Figure 1.9 shows `Button` examples.

Figure 1.9: Various buttons

[Graphic: Figure 1-9]

The Java Management API includes a fancier button (`ImageButton`) with pictures rather than labels. For the time being, this is a standard extension of Java and not in the Core API. If you don't want to use these extensions, you'll have to implement an image button yourself.

Expansion

The Canvas class

The `Canvas` class is just a blank area; it doesn't have any predefined appearance. You can use `Canvas` for drawing images, building new kinds of components, or creating super-components that are aggregates of other components. For example, you can build a picture button by drawing a picture on a `Canvas` and detecting mouse click events within the area of the `Canvas`. `Canvas` is discussed in Canvas.

PREVIOUS HOME NEXT
Acknowledgments BOOK INDEX Peers

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

2. Simple Graphics

Contents:

This chapter digs into the meat of the AWT classes. After completing this chapter, you will be able to draw strings, images, and shapes via the `Graphics` class in your Java programs. We discuss geometry-related classes--`Polygon`, `Rectangle`, `Point`, and `Dimension`, and the `Shape` interface--you will see these throughout the remaining AWT objects. You will also learn several ways to do smooth animation by using double buffering and the `MediaTracker`.

After reading this chapter, you should be able to do simple animation and image manipulation with AWT. For most applications, this should be sufficient. If you want to look at AWT's more advanced graphics capabilities, be sure to take a look at Chapter 12, *Image Processing*.

2.1 Graphics

The `Graphics` class is an abstract class that provides the means to access different graphics devices. It is the class that lets you draw on the screen, display images, and so forth. `Graphics` is an abstract class because working with graphics requires detailed knowledge of the platform on which the program runs. The actual work is done by concrete classes that are closely tied to a particular platform. Your Java Virtual Machine vendor provides the necessary concrete classes for your environment. You never need to worry about the platform-specific classes; once you have a `Graphics` object, you can call all the methods of the `Graphics` class, confident that the platform-specific classes will work correctly wherever your program runs.

You rarely need to create a `Graphics` object yourself; its constructor is protected and is only called by the subclasses that extend `Graphics`. How then do you get a `Graphics` object to work with? The sole parameter of the `Component.paint()` and `Component.update()` methods is the current graphics context. Therefore, a `Graphics` object is always available when you override a component's `paint()` and `update()` methods. You can ask for the graphics context of a `Component` by calling `Component.getGraphics()`. However, many components do not have a drawable graphics context. `Canvas` and `Container` objects return a valid `Graphics` object; whether or not any other component has a drawable graphics context depends on the run-time environment. (The latest versions of Netscape Navigator provide a drawable graphics context for any component, but you shouldn't get used to writing platform-specific code.) This restriction isn't as harsh as it sounds. For most components, a drawable graphics context doesn't make much sense; for example, why would you want to draw on a `List`? If you want to draw on a component, you probably can't. The notable exception is `Button`, and that may be fixed in future versions of AWT.

Graphics Methods

Constructors

protected Graphics ()

> Because `Graphics` is an abstract class, it doesn't have a visible constructor. The way to get a `Graphics` object is to ask for one by calling `getGraphics()` or to use the one given to you by the `Component.paint()` or `Component.update()` method.

The abstract methods of the `Graphics` class are implemented by some windowing system-specific class. You rarely need to know which subclass of `Graphics` you are using, but the classes you actually get (if you are using the JDK) are `sun.awt.win32.Win32Graphics` (JDK1.0), `sun.awt.window.WGraphics` (JDK1.1), `sun.awt.motif.X11Graphics`, or `sun.awt.macos.MacGraphics`. Pseudo-constructors

In addition to using the graphics contexts given to you by `getGraphics()` or in `Component.paint()`, you can get a `Graphics` object by creating a copy of another `Graphics` object. Creating new graphics contexts has resource implications. Certain platforms have a limited number of graphics contexts that can be active. For instance, on Windows 95 you cannot have more than four in use at one time. Therefore, it's a good idea to call `dispose()` as soon as you are done with a `Graphics` object. Do not rely on the garbage collector to clean up for you.

public abstract Graphics create ()

> This method creates a second reference to the graphics context. It is useful for clipping (reducing the drawable area).

public Graphics create (int x, int y, int width, int height)

> This method creates a second reference to a subset of the drawing area of the graphics context. The new `Graphics` object covers the rectangle from (`x`, `y`) through (`x+width-1`, `y+height-1`) in the original object. The coordinate space of the new `Graphics` context is translated so that the upper left corner is (0, 0) and the lower right corner is (`width`, `height`). Shifting the coordinate system of the new object

makes it easier to work within a portion of the drawing area without using offsets.

Drawing strings

These methods let you draw text strings on the screen. The coordinates refer to the left end of the text's baseline.

public abstract void drawString (String text, int x, int y)

The `drawString()` method draws `text` on the screen in the current font and color, starting at position (`x`, `y`). The starting coordinates specify the left end of the `String`'s baseline.

public void drawChars (char text[], int offset, int length, int x, int y)

The `drawChars()` method creates a `String` from the char array `text` starting at `text[offset]` and continuing for `length` characters. The newly created `String` is then drawn on the screen in the current font and color, starting at position (`x`, `y`). The starting coordinates specify the left end of the `String`'s baseline.

public void drawBytes (byte text[], int offset, int length, int x, int y)

The `drawBytes()` method creates a `String` from the byte array `text` starting at `text[offset]` and continuing for `length` characters. This `String` is then drawn on the screen in the current font and color, starting at position (`x`, `y`). The starting coordinates specify the left end of the `String`'s baseline.

public abstract Font getFont ()

The `getFont()` method returns the current `Font` of the graphics context. See Chapter 3, *Fonts and Colors*, for more on what you can do with fonts. You cannot get meaningful results with `getFont()` until the applet or application is displayed on the screen (generally, not in `init()` of an applet or `main()` of an application).

public abstract void setFont (Font font)

The `setFont()` method changes the current `Font` to `font`. If `font` is not available on the current platform, the system chooses a default. To change the current font to 12 point bold TimesRoman:

```
setFont (new Font ("TimesRoman", Font.BOLD, 12));
```

public FontMetrics getFontMetrics ()

The `getFontMetrics()` method returns the current `FontMetrics` object of the graphics context. You use `FontMetrics` to reveal sizing properties of the current `Font`--for example, how wide the "Hello World" string will be in pixels when displayed on the screen.

public abstract FontMetrics getFontMetrics (Font font)

This version of `getFontMetrics()` returns the `FontMetrics` for the `Font` font instead of the current font. You might use this method to see how much space a new font requires to draw text.

For more information about `Font` and `FontMetrics`, see Chapter 3, *Fonts and Colors*. Painting

public abstract Color getColor ()

The `getColor()` method returns the current foreground `Color` of the `Graphics` object. All future drawing operations will use this color. Chapter 3, *Fonts and Colors* describes the `Color` class.

public abstract void setColor (Color color)

The `setColor()` method changes the current drawing color to `color`. As you will see in the next chapter, the `Color` class defines some common colors for you. If you can't use one of the predefined colors, you can create a color from its RGB values. To change the current color to red, use any of the following:

```
setColor (Color.red);
setColor (new Color (255, 0, 0));
setColor (new Color (0xff0000));
```

public abstract void clearRect (int x, int y, int width, int height)

The `clearRect()` method sets the rectangular drawing area from (`x`, `y`) to (`x+width-1`, `y+height-1`) to the current background color. Keep in mind that the second pair of parameters is not the opposite corner of the rectangle, but the width and height of the area to clear.

public abstract void clipRect (int x, int y, int width, int height)

The `clipRect()` method reduces the drawing area to the intersection of the current drawing area and the rectangular area from (`x`, `y`) to (`x+width-1`, `y+height-1`). Any future drawing operations outside this clipped area will have no effect. Once you clip a drawing area, you cannot increase its size with `clipRect()`; the drawing area can only get smaller. (However, if the `clipRect()` call is in `paint()`, the size of the drawing area will be reset to its original size on subsequent calls to `paint()`.) If you want the ability to draw to the entire area, you must create a second `Graphics` object that contains a copy of the drawing area before calling `clipRect()` or use `setClip()`. The following code is a simple applet that demonstrates clipping; Figure 2.1 shows the result.

```
import java.awt.*;
public class clipping extends java.applet.Applet {
    public void paint (Graphics g) {
        g.setColor (Color.red);
        Graphics clippedGraphics = g.create();
        clippedGraphics.drawRect (0,0,100,100);
        clippedGraphics.clipRect (25, 25, 50, 50);
        clippedGraphics.drawLine (0,0,100,100);
        clippedGraphics.dispose();
        clippedGraphics=null;
        g.drawLine (0,100,100,0);
      }
    }
```

Figure 2.1: Clipping restricts the drawing area

[Graphic: Figure 2-1]

The `paint()` method for this applet starts by setting the foreground color to red. It then creates a copy of the `Graphics` context for clipping, saving the original object so it can draw on the entire screen later. The applet then draws a rectangle, sets the clipping area to a smaller region, and draws a diagonal line across the rectangle from upper left to lower right. Because clipping is in effect, only part of the line is displayed. The applet then discards the clipped `Graphics` object and draws an unclipped line from lower left to upper right using the original object `g`.

public abstract void setClip(int x, int y, int width, int height) ★

> This `setClip()` method allows you to change the current clipping area based on the parameters provided. `setClip()` is similar to `clipRect()`, except that it is not limited to shrinking the clipping area. The current drawing area becomes the rectangular area from (`x`, `y`) to (`x+width-1`, `y+height-1`); this area may be larger than the previous drawing area.

public abstract void setClip(Shape clip) ★

> This `setClip()` method allows you to change the current clipping area based on the `clip` parameter, which may be any object that implements the `Shape` interface. Unfortunately, practice is not as good as theory, and in practice, `clip` must be a `Rectangle`; if you pass `setClip()` a `Polygon`, it throws an `IllegalArgumentException`.[1] (The `Shape` interface is discussed later in this chapter.)

[1] It should be simple for Sun to fix this bug; one would expect clipping to a `Polygon` to be the same as clipping to the `Polygon`'s bounding rectangle.

public abstract Rectangle getClipBounds () ★
public abstract Rectangle getClipRect () ☆

The `getClipBounds()` methods returns a `Rectangle` that describes the clipping area of a `Graphics` object. The `Rectangle` gives you the (`x`, `y`) coordinates of the top left corner of the clipping area along with its width and height. (`Rectangle` objects are discussed later in this chapter.)

`getClipRect()` is the Java 1.0 name for this method.

public abstract Shape getClip () ★

The `getClip()` method returns a `Shape` that describes the clipping area of a `Graphics` object. That is, it returns the same thing as `getClipBounds()` but as a `Shape`, instead of as a `Rectangle`. By calling `Shape.getBounds()`, you can get the (`x`, `y`) coordinates of the top left corner of the clipping area along with its width and height. In the near future, it is hard to imagine the actual object that `getClip()` returns being anything other than a `Rectangle`.

public abstract void copyArea (int x, int y, int width, int height, int delta_x, int delta_y)

The `copyArea()` method copies the rectangular area from (`x`, `y`) to (`x+width`, `y+height`) to the area with an upper left corner of (`x+delta_x`, `y+delta_y`). The `delta_x` and `delta_y` parameters are not the coordinates of the second point but an offset from the first coordinate pair (`x`, `y`). The area copied may fall outside of the clipping region. This method is often used to tile an area of the graphics context. `copyArea()` does not save the contents of the area copied.

Painting mode

There are two painting or drawing modes for the `Graphics` class: paint (the default) and XOR mode. In paint mode, anything you draw replaces whatever is already on the screen. If you draw a red square, you get a red square, no matter what was underneath; this is what most programmers have learned to expect.

The behavior of XOR mode is rather strange, at least to people accustomed to modern programming environments. XOR mode is short for eXclusive-OR mode. The idea behind XOR mode is that drawing the same object twice returns the screen to its original state. This technique was commonly used for simple animations prior to the development of more sophisticated methods and cheaper hardware.

The side effect of XOR mode is that painting operations don't necessarily get the color you request. Instead of replacing the original pixel with the new value, XOR mode merges the original color, the painting color, and an XOR color (usually the background color) to form a new color. The new color is chosen so that if you repaint the pixel with the same color, you get the original pixel back. For example, if you paint a red square in XOR mode, you get a square of some other color on the screen. Painting the same red square again returns the screen to its original state.

public abstract void setXORMode (Color xorColor)

The `setXORMode()` method changes the drawing mode to XOR mode. In XOR mode, the system uses the `xorColor` color to determine an alternate color for anything drawn such that drawing the same item twice restores the screen to its original condition. The `xorColor` is usually the current background color but can be any color. For each pixel, the new color is determined by an exclusive-or of the old pixel color, the painting color, and the `xorColor`.

For example, if the old pixel is red, the XOR color is blue, and the drawing color is green, the end result would be white. To see why, it is necessary to look at the RGB values of the three colors. Red is (255, 0, 0). Blue is (0, 0, 255). Green is (0, 255, 0). The exclusive-or of these three values is (255, 255, 255), which is white. Drawing another green pixel with a blue XOR color yields red, the pixel's original color, since (255, 255, 255) ^ (0, 0, 255) ^ (0, 255, 0) yields (255, 0, 0).[2] The following code generates the display shown in Figure 2.2.

[2] ^ is the Java XOR operator.

```
import java.awt.*;
public class xor extends java.applet.Applet {
    public void init () {
        setBackground (Color.red);
    }
    public void paint (Graphics g) {
        g.setColor (Color.green);
        g.setXORMode (Color.blue);
        g.fillRect (10, 10, 100, 100);
        g.fillRect (10, 60, 100, 100);
    }
}
```

Figure 2.2: Drawing in XOR mode

[Graphic: Figure 2-2]

Although it's hard to visualize what color XOR mode will pick, there is one important special case. Let's say that there are only two colors: a background color (the XOR color) and a foreground color (the painting color). Each pixel must be in one color or the other. Painting "flips" each pixel to the other color. Foreground pixels become background, and vice versa.

public abstract void setPaintMode ()

> The `setPaintMode()` method puts the system into paint mode. When in paint mode, any drawing operation replaces whatever is underneath it. Call `setPaintMode()` to return to normal painting when finished with XOR mode.

Drawing shapes

Most of the drawing methods require you to specify a bounding rectangle for the object you want to draw: the location of the object's upper left corner, plus its width and height. The two exceptions are lines and polygons. For lines, you supply two endpoints; for polygons, you provide a set of points.

Versions 1.0.2 and 1.1 of AWT always draw solid lines that are one pixel wide; there is no support for line width or fill patterns. A future version should support lines with variable widths and patterns.

public abstract void drawLine (int x1, int y1, int x2, int y2)

> The `drawLine()` method draws a line on the graphics context in the current color from (`x1`, `y1`) to (`x2`, `y2`). If (`x1`, `y1`) and (`x2`, `y2`) are the same point, you will draw a point. There is no method specific to drawing a point. The following code generates the display shown in Figure 2.3.

```
g.drawLine (5, 5, 50, 75);    // line
g.drawLine (5, 75, 5, 75);    // point
g.drawLine (50, 5, 50, 5);    // point
```

Figure 2.3: Drawing lines and points with drawLine()

public void drawRect (int x, int y, int width, int height)

The `drawRect()` method draws a rectangle on the drawing area in the current color from (`x`, `y`) to (`x+width`, `y+height`). If `width` or `height` is negative, nothing is drawn.

public abstract void fillRect (int x, int y, int width, int height)

The `fillRect()` method draws a filled rectangle on the drawing area in the current color from (`x`, `y`) to (`x+width-1`, `y+height-1`). Notice that the filled rectangle is one pixel smaller to the right and bottom than requested. If `width` or `height` is negative, nothing is drawn.

public abstract void drawRoundRect (int x, int y, int width, int height, int arcWidth, int arcHeight)

The `drawRoundRect()` method draws a rectangle on the drawing area in the current color from (`x`, `y`) to (`x+width`, `y+height`). However, instead of perpendicular corners, the corners are rounded with a horizontal diameter of `arcWidth` and a vertical diameter of `arcHeight`. If `width` or `height` is a negative number, nothing is drawn. If `width`, `height`, `arcWidth`, and `arcHeight` are all equal, you get a circle.

To help you visualize the `arcWidth` and `arcHeight` of a rounded rectangle, Figure 2.4 shows one corner of a rectangle drawn with an `arcWidth` of 20 and a `arcHeight` of 40.

Figure 2.4: Drawing rounded corners

[Graphic: Figure 2-4]

public abstract void fillRoundRect (int x, int y, int width, int height, int arcWidth, int arcHeight)

The `fillRoundRect()` method draws a filled rectangle on the drawing area in the current color from (`x`, `y`) to (`x+width-1`, `y+height-1`). However, instead of having perpendicular corners, the corners are rounded with a horizontal diameter of `arcWidth` and a vertical diameter of `arcHeight` for the four corners. Notice that the filled rectangle is one pixel smaller to the right and bottom than requested. If `width` or `height` is a negative number, nothing is filled. If `width`, `height`, `arcWidth`, and `arcHeight` are all equal, you get a filled circle.

Figure 2.4 shows how AWT generates rounded corners. Figure 2.5 shows the collection of rectangles created by the following code. The rectangles in Figure 2.5 are filled and unfilled, with rounded and square corners.

```
g.drawRect (25, 10, 50, 75);
g.fillRect (25, 110, 50, 75);
g.drawRoundRect (100, 10, 50, 75, 60, 50);
g.fillRoundRect (100, 110, 50, 75, 60, 50);
```

Figure 2.5: Varieties of rectangles

[Graphic: Figure 2-5]

public void draw3DRect (int x, int y, int width, int height, boolean raised)

The `draw3DRect()` method draws a rectangle in the current color from (`x`, `y`) to (`x+width`, `y+height`); a shadow effect makes the rectangle appear to float slightly above or below the screen. The `raised` parameter has an effect only if the current color is not black. If `raised` is `true`, the rectangle looks like a button waiting to be pushed. If `raised` is `false`, the rectangle looks like a depressed button. If `width` or `height` is negative, the shadow appears from another direction.

public void fill3DRect (int x, int y, int width, int height, boolean raised)

The `fill3DRect()` method draws a filled rectangle in the current color from (`x`, `y`) to (`x+width`, `y+height`); a shadow effect makes the rectangle appear to float slightly above or below the screen. The `raised` parameter has an effect only if the current color is not black. If `raised` is `true`, the rectangle looks like a button waiting to be pushed. If `raised` is `false`, the rectangle looks like a depressed button. To enhance the shadow effect, the depressed area is given a slightly deeper shade of the drawing color. If `width` or `height` is negative, the shadow appears from another direction, and the rectangle isn't filled. (Different platforms could deal with this differently. Try to ensure the parameters have positive values.)

Figure 2.6 shows the collection of three-dimensional rectangles created by the following code. The rectangles in the figure are raised and depressed, filled and unfilled.

```
g.setColor (Color.gray);
g.draw3DRect (25, 10, 50, 75, true);
g.draw3DRect (25, 110, 50, 75, false);
g.fill3DRect (100, 10, 50, 75, true);
g.fill3DRect (100, 110, 50, 75, false);
```

Figure 2.6: Filled and unfilled 3D rectangles

[Graphic: Figure 2-6]

public abstract void drawOval (int x, int y, int width, int height)

The `drawOval()` method draws an oval in the current color within an invisible bounding rectangle from (`x`, `y`) to (`x+width`, `y+height`). You cannot specify the oval's center point and radii. If `width` and `height` are equal, you get a circle. If `width` or `height` is negative, nothing is drawn.

public abstract void fillOval (int x, int y, int width, int height)

The `fillOval()` method draws a filled oval in the current color within an invisible bounding rectangle from (`x`, `y`) to (`x+width-1`, `y+height-1`). You cannot specify the oval's center point and radii. Notice that the filled oval is one pixel smaller to the right and bottom than requested. If `width` or `height` is negative, nothing is drawn.

Figure 2.7 shows the collection of ovals, filled and unfilled, that were generated by the following code:

```
g.drawOval (25, 10, 50, 75);
g.fillOval (25, 110, 50, 75);
g.drawOval (100, 10, 50, 50);
g.fillOval (100, 110, 50, 50);
```

Figure 2.7: Filled and unfilled ovals

[Graphic: Figure 2-7]

public abstract void drawArc (int x, int y, int width, int height, int startAngle, int arcAngle)

The `drawArc()` method draws an arc in the current color within an invisible bounding rectangle from (`x`, `y`) to (`x+width`, `y+height`). The arc starts at `startAngle` degrees and goes to `startAngle` + `arcAngle` degrees. An angle of 0 degrees is at the 3 o'clock position; angles increase counter-clockwise. If `arcAngle` is negative, drawing is in a clockwise direction. If `width` and `height` are equal and `arcAngle` is 360 degrees, `drawArc()` draws a circle. If `width` or `height` is negative, nothing is drawn.

public abstract void fillArc (int x, int y, int width, int height, int startAngle, int arcAngle)

The `fillArc()` method draws a filled arc in the current color within an invisible bounding rectangle from (`x`, `y`) to (`x+width-1`, `y+height-1`). The arc starts at `startAngle` degrees and goes to `startAngle` + `arcAngle` degrees. An angle of 0 degrees is at the 3 o'clock position; angles increase counter-clockwise. If `arcAngle` is negative, drawing is in a clockwise direction. The arc fills like a pie (to the origin), not from arc endpoint to arc endpoint. This makes creating pie charts easier. If `width` and `height` are equal and `arcAngle` is 360 degrees, `fillArc()` draws a filled circle. If `width` or `height` is negative, nothing is drawn.

Figure 2.8 shows a collection of filled and unfilled arcs that were generated by the following code:

```
g.drawArc (25, 10, 50, 75, 0, 360);
g.fillArc (25, 110, 50, 75, 0, 360);
g.drawArc (100, 10, 50, 75, 45, 215);
g.fillArc (100, 110, 50, 75, 45, 215);
```

Figure 2.8: Filled and unfilled arcs

[Graphic: Figure 2-8]

public void drawPolygon (Polygon p)

The `drawPolygon()` method draws a path for the points in polygon `p` in the current color. Polygon discusses the `Polygon` class in detail.

The behavior of `drawPolygon()` changes slightly between Java 1.0.2 and 1.1. With version 1.0.2, if the first and last points of a `Polygon` are not the same, a call to `drawPolygon()` results in an open polygon, since the endpoints are not connected for you. Starting with version 1.1, if the first and last points are not the same, the endpoints are connected for you.

public abstract void drawPolygon (int xPoints[], int yPoints[], int numPoints)

The `drawPolygon()` method draws a path of `numPoints` nodes by plucking one element at a time out of `xPoints` and `yPoints` to make each point. The path is drawn in the current color. If either `xPoints` or `yPoints` does not have `numPoints` elements, `drawPolygon()` throws a run-time exception. In 1.0.2, this exception is an `IllegalArgumentException`; in 1.1, it is an `ArrayIndexOutOfBoundsException`. This change shouldn't break older programs, since you are not required to catch run-time exceptions.

public abstract void drawPolyline (int xPoints[], int yPoints[], int numPoints) ★

The `drawPolyline()` method functions like the 1.0 version of `drawPolygon()`. It plays connect the dots with the points in the `xPoints` and `yPoints` arrays and does not connect the endpoints. If either `xPoints` or `yPoints` does not have `numPoints` elements, `drawPolygon()` throws the run-time exception, `ArrayIndexOutOfBoundsException`.

Filling polygons is a complex topic. It is not as easy as filling rectangles or ovals because a polygon may not be closed and its edges may cross. AWT uses an even-odd rule to fill polygons. This algorithm works by counting the number of times each scan line crosses an edge of the polygon. If the total number of crossings to the left of the current point is odd, the point is colored. If it is even, the point is left alone. Figure 2.9 demonstrates this algorithm for a single scan line that intersects the polygon at x values of 25, 75, 125, 175, 225, and 275.

Figure 2.9: Polygon fill algorithm

[Graphic: Figure 2-9]

The scan line starts at the left edge of the screen; at this point there haven't been any crossings, so the pixels are left untouched. The scan line reaches the first crossing when x equals 25. Here, the total number of crossings to the left is one, so the scan line is inside the polygon, and the pixels are colored. At 75, the scan line crosses again; the total number of crossings is two, so coloring stops.

public void fillPolygon (Polygon p)

> The `fillPolygon()` method draws a filled polygon for the points in `Polygon p` in the current color. If the polygon is not closed, `fillPolygon()` adds a segment connecting the endpoints. Polygon discusses the `Polygon` class in detail.

public abstract void fillPolygon (int xPoints[], int yPoints[], int nPoints)

> The `fillPolygon()` method draws a polygon of `numPoints` nodes by plucking one element at a time out of `xPoints` and `yPoints` to make each point. The polygon is drawn in the current color. If either `xPoints` or `yPoints` does not have `numPoints` elements, `fillPolygon()` throws the run-time exception `IllegalArgumentException`. If the polygon is not closed, `fillPolygon()` adds a segment connecting the endpoints.[3]
>
> > [3] In Java 1.1, this method throws `ArrayIndexOutOfBoundsException`, not `IllegalArgumentException`.

Figure 2.10 shows several polygons created by the following code, containing different versions of `drawPolygon()` and `fillPolygon()`:

```
int[] xPoints[] = {{50, 25, 25, 75, 75},
                   {50, 25, 25, 75, 75},
                   {100, 100, 150, 100, 150, 150, 125, 100, 150},
                   {100, 100, 150, 100, 150, 150, 125, 100, 150}};
int[] yPoints[] = {{10, 35, 85, 85, 35, 10},
                   {110, 135, 185, 185, 135},
                   {85, 35, 35, 85, 85, 35, 10, 35, 85},
                   {185, 135, 135, 185, 185, 135, 110, 135, 185}};
int   nPoints[] = {5, 5, 9, 9};
g.drawPolygon (xPoints[0], yPoints[0], nPoints[0]);
g.fillPolygon (xPoints[1], yPoints[1], nPoints[1]);
g.drawPolygon (new Polygon(xPoints[2], yPoints[2], nPoints[2]));
g.fillPolygon (new Polygon(xPoints[3], yPoints[3], nPoints[3]));
```

Figure 2.10: Filled and unfilled polygons

[Graphic: Figure 2-10]

Drawing images

An `Image` is a displayable object maintained in memory. To get an image on the screen, you must draw it onto a graphics context, using the `drawImage()` method of the `Graphics` class. For example, within a `paint()` method, you would call `g.drawImage(image, ... , this)` to display some image on the screen. In other situations, you might use the `createImage()` method to generate an offscreen `Graphics` object, then use `drawImage()` to draw an image onto this object, for display later.

This begs the question: where do images come from? We will have more to say about the `Image` class later in this chapter. For now, it's enough to say that you can call `getImage()` to load an image from disk or across the Net. There are versions of `getImage()` in the `Applet` and `Toolkit` classes; the latter is for use in applications. You can also call `createImage()`, a method of the `Component` class, to generate an image in memory.

What about the last argument to `drawImage()`? What is `this` for? The last argument of `drawImage()` is always an image observer--that is, an object that implements the `ImageObserver` interface. This interface is discussed in detail in Chapter 12, *Image Processing*. For the time being, it's enough to say that the call to `drawImage()` starts a new thread that loads the requested image. An image observer monitors the process of loading an image; the thread that is loading the image notifies the image observer whenever new data has arrived.

The `Component` class implements the `ImageObserver` interface; when you're writing a `paint()` method, you're almost certainly overriding some component's `paint()` method; therefore, it's safe to use `this` as the image observer in a call to `drawImage()`. More simply, we could say that any component can serve as an image observer for images that are drawn on it.

public abstract boolean drawImage (Image image, int x, int y, ImageObserver observer)

> The `drawImage()` method draws `image` onto the screen with its upper left corner at (`x`, `y`), using `observer` as its `ImageObserver`. Returns `true` if the object is fully drawn, `false` otherwise.

public abstract boolean drawImage (Image image, int x, int y, int width, int height, ImageObserver observer)

> The `drawImage()` method draws `image` onto the screen with its upper left corner at (`x`, `y`), using `observer` as its `ImageObserver`. The system scales `image` to fit into a `width` x `height` area. The scaling may take time. This method returns `true` if the object is fully drawn, `false` otherwise.
>
> With Java 1.1, you don't need to use `drawImage()` for scaling; you can prescale the image with the `Image.getScaledInstance()` method, then use the previous version of `drawImage()`.

public abstract boolean drawImage (Image image, int x, int y, Color backgroundColor, ImageObserver observer)

> The `drawImage()` method draws `image` onto the screen with its upper left corner at (`x`, `y`), using `observer` as its `ImageObserver`. `backgroundColor` is the color of the background seen through the transparent parts of the image. If no part of the image is transparent, you will not see `backgroundColor`. Returns `true` if the object is fully drawn, `false` otherwise.

public abstract boolean drawImage (Image image, int x, int y, int width, int height, Color backgroundColor, ImageObserver observer)

> The `drawImage()` method draws `image` onto the screen with its upper left corner at (`x`, `y`), using `observer` as its `ImageObserver`. `backgroundColor` is the color of the background seen through the transparent parts of the image. The system scales `image` to fit into a `width` x `height` area. The scaling may take time. This method returns `true` if the image is fully drawn, `false` otherwise.
>
> With Java 1.1, you can prescale the image with the `AreaAveragingScaleFilter` or `ReplicateScaleFilter` described in Chapter 12, *Image Processing*, then use the previous version of `drawImage()` to display it.

The following code generated the images in Figure 2.11. The images on the left come from a standard JPEG file. The images on the right come from a file in GIF89a format, in which the white pixel is "transparent." Therefore, the gray background shows through this pair of images.

```
import java.awt.*;
import java.applet.*;
public class drawingImages extends Applet {
    Image i, j;
    public void init () {
        i = getImage (getDocumentBase(), "rosey.jpg");
        j = getImage (getDocumentBase(), "rosey.gif");
    }
    public void paint (Graphics g) {
        g.drawImage (i, 10, 10, this);
        g.drawImage (i, 10, 85, 150, 200, this);
        g.drawImage (j, 270, 10, Color.lightGray, this);
        g.drawImage (j, 270, 85, 150, 200, Color.lightGray, this);
    }
}
```

Figure 2.11: Scaled and unscaled images

[Graphic: Figure 2-11]

public abstract boolean drawImage(Image img, int dx1, int dy1, int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, ImageObserver observer) ★

> The `drawImage()` method draws a portion of `image` onto the screen. It takes the part of the image with corners at (`sx1`, `sy1`) and (`sx2`, `sy2`); it places this rectangular snippet on the screen with one corner at (`dx1`, `dy1`) and another at (`dx2`, `dy2`), using `observer` as its `ImageObserver`. (Think of `d` for destination location and `s` for source image.) This method returns `true` if the object is fully drawn, `false` otherwise.
>
> `drawImage()` flips the image if source and destination endpoints are not the same corners, crops the image if the destination is smaller than the original size, and scales the image if the destination is larger than the original size. It does not do rotations, only flips (i.e., it can produce a mirror image or an image rotated 180 degrees but not an image rotated 90 or 270 degrees).

public abstract boolean drawImage(Image img, int dx1, int dy1, int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, Color backgroundColor, ImageObserver observer) ★

The drawImage() method draws a portion of image onto the screen. It takes the part of the image with corners at (sx1, sy1) and (sx2, sy2); it places this rectangular snippet on the screen with one corner at (dx1, dy1) and another at (dx2, dy2), using observer as its ImageObserver. (Think of d for destination location and s for source image.) backgroundColor is the color of the background seen through the transparent parts of the image. If no part of the image is transparent, you will not see backgroundColor. This method returns true if the object is fully drawn, false otherwise.

Like the previous version of drawImage(), this method flips the image if source and destination endpoints are not the same corners, crops the image if the destination is smaller than the original size, and scales the image if the destination is larger than the original size. It does not do rotations, only flips (i.e., it can produce a mirror image or an image rotated 180 degrees but not an image rotated 90 or 270 degrees).

The following code demonstrates the new drawImage() methods in Java 1.1. They allow you to scale, flip, and crop images without the use of image filters. The results are shown in Figure 2.12.

```
// Java 1.1 only
import java.awt.*;
import java.applet.*;
public class drawingImages11 extends Applet {
    Image i, j;
    public void init () {
        i = getImage (getDocumentBase(), "rosey.gif");
    }
    public void paint (Graphics g) {
        g.drawImage (i, 10, 10, this);
        g.drawImage (i, 10, 85,
                     i.getWidth(this)+10, i.getHeight(this)+85,
                     i.getWidth(this), i.getHeight(this), 0, 0, this);
        g.drawImage (i, 270, 10,
                     i.getWidth(this)+270, i.getHeight(this)*2+10, 0, 0,
                     i.getWidth(this), i.getHeight(this), Color.gray, this);
        g.drawImage (i, 10, 170,
                     i.getWidth(this)*2+10, i.getHeight(this)+170, 0,
                     i.getHeight(this)/2, i.getWidth(this)/2, 0, this);
    }
}
```

Figure 2.12: Flipped, mirrored, and cropped images

[Graphic: Figure 2-12]

Miscellaneous methods

public abstract void translate (int x, int y)

The `translate()` method sets how the system translates the coordinate space for you. The point at the (`x`, `y`) coordinates becomes the origin of this graphics context. Any future drawing will be relative to this location. Multiple translations are cumulative. The following code leaves the coordinate system translated by (100, 50).

```
translate (100, 0);
translate (0, 50);
```

Note that each call to `paint()` provides an entirely new `Graphics` context with its origin in the upper left corner. Therefore, don't expect translations to persist from one call to `paint()` to the next.

public abstract void dispose ()

The `dispose()` method frees any system resources used by the `Graphics` context. It's a good idea to call `dispose()` whenever you are finished with a `Graphics` object, rather than waiting for the garbage collector to call it automatically (through `finalize()`). Disposing of the `Graphics` object yourself will help your programs on systems with limited resources. However, you should not dispose the `Graphics` parameter to `Component.paint()` or `Component.update()`.

public void finalize ()

The garbage collector calls `finalize()` when it determines that the `Graphics` object is no longer needed. `finalize()` calls `dispose()`, which frees any resources that the `Graphics` object has used.

public String toString ()

The `toString()` method of `Graphics` returns a string showing the current font and color. However, `Graphics` is an abstract class, and classes that extend `Graphics` usually override `toString()`. For example, on a Windows 95 machine, `sun.awt.win32.Win32Graphics` is the concrete class that extends `Graphics`. The class's `toString()` method displays the current origin of the `Graphics` object, relative to the original coordinate system:

```
sun.awt.win32.Win32Graphics[0,0]
```

3. Fonts and Colors

Contents:

This chapter introduces the `java.awt` classes that are used to work with different fonts and colors. First, we discuss the `Font` class, which determines the font used to display text strings, whether they are drawn directly on the screen (with `drawString()`) or displayed within a component like a text field. The `FontMetrics` class gives you detailed information about a font, which you can use to position text strings intelligently. Next, the `Color` class is used to represent colors and can be used to specify the background color of any object, as well as the foreground color used to display a text string or a shape. Finally, the `SystemColor` class (which is new to Java 1.1) provides access to the desktop color scheme.

3.1 Fonts

An instance of the `Font` class represents a specific font to the system. Within AWT, a font is specified by its name, style, and point size. Each platform that supports Java provides a basic set of fonts; to find the fonts supported on any platform, call `Toolkit.getDefaultToolkit().getFontList()`. This method returns a `String` array of the fonts available. Under Java 1.0, on any platform, the available fonts were: TimesRoman, Helvetica, Courier, Dialog, DialogInput, and ZapfDingbats. For copyright reasons, the list is substantially different in Java 1.1: the available font names are TimesRoman ☆, Serif, Helvetica ☆, SansSerif, Courier ☆, Monospaced, Dialog, and DialogInput. The actual fonts available aren't changing; the deprecated font names are being replaced by non-copyrighted equivalents. Thus, TimesRoman is now Serif, Helvetica is now SansSerif, and Courier is Monospaced. The ZapfDingbats font name has been dropped completely because the characters in this font have official Unicode mappings in the range \u2700 to \u27ff.

NOTE:

If you desire non-Latin font support with Java 1.1, use the Unicode mappings for the characters. The actual font used is specified in a set of *font.properties* files in the *lib* subdirectory under *java.home*. These localized font files allow you to remap the "Serif", "SansSerif", and "Monospaced" names to different fonts.

The font's `style` is passed with the help of the class variables `Font.PLAIN`, `Font.BOLD`, and `Font.ITALIC`. The combination `Font.BOLD | Font.ITALIC` specifies bold italics.

A font's `size` is represented as an integer. This integer is commonly thought of as a point size; although that's not strictly correct, this book follows common usage and talks about font sizes in points.

It is possible to add additional font names to the system by setting properties. For example, putting the line below in the properties file or a resource file (resource files are new to Java 1.1) defines the name "AvantGarde" as an alias for the font SansSerif:

```
awt.font.avantgarde=SansSerif
```

With this line in the properties file, a Java program can use "AvantGarde" as a font name; when this font is selected, AWT uses the font SansSerif for display. The property name must be all lowercase. Note that we haven't actually added a new font to the system; we've only created a new name for an old font. See the discussion of `getFont()` and `decode()` for more on font properties.

The Font Class

Constants

There are four styles for displaying fonts in Java: plain, bold, italic, and bold italic. Three class constants are used to represent font styles:

public static final int BOLD

> The `BOLD` constant represents a boldface font.

public static final int ITALIC

> The `ITALIC` constant represents an italic font.

public static final int PLAIN

> The `PLAIN` constant represents a plain or normal font.

The combination `BOLD | ITALIC` represents a bold italic font. `PLAIN` combined with either `BOLD` or

ITALIC represents bold or italic, respectively.

There is no style for underlined text. If you want underlining, you have to do it manually, with the help of FontMetrics.

NOTE:

If you are using Microsoft's SDK, the com.ms.awt.FontX class includes direct support for underlined, strike through (line through middle), and outline fonts.

Variables

Three protected variables access the font setting. They are initially set through the Font constructor. To read these variables, use the Font class's "get" methods.

protected String name

The name of the font.

protected int size

The size of the font.

protected int style

The style of the font. The style is some logical combination of the constants listed previously.

Constructors

public Font (String name, int style, int size)

There is a single constructor for Font. It requires a name, style, and size. name represents the name of the font to create, case insensitive.

```
setFont (new Font ("TimesRoman", Font.BOLD | Font.ITALIC, 20));
```

Characteristics

public String getName ()

The getName() method returns the font's logical name. This is the name passed to the constructor for the specific instance of the Font. Remember that system properties can be used to alias font names, so the name used in the constructor isn't necessarily the actual name of a font on the system.

public String getFamily ()

The `getFamily()` method returns the actual name of the font that is being used to display characters. If the font has been aliased to another font, the `getFamily()` method returns the name of the platform-specific font, not the alias. For example, if the constructor was `new Font ("AvantGarde", Font.PLAIN, 10)` and the `awt.font.avantgarde=Helvetica` property is set, then `getName()` returns AvantGarde, and `getFamily()` returns Helvetica. If nobody set the property, both methods return AvantGarde, and the system uses the default font (since AvantGarde is a nonstandard font).

public int getStyle ()

The `getStyle()` method returns the current style of the font as an integer. Compare this value with the constants `Font.BOLD`, `Font.PLAIN`, and `Font.ITALIC` to see which style is meant. It is easier to use the `isPlain()`, `isBold()`, and `isItalic()` methods to find out the current style. `getStyle()` is more useful if you want to copy the style of some font when creating another.

public int getSize ()

The `getSize()` method retrieves the point size of the font, as set by the size parameter in the constructor. The actual displayed size may be different.

public FontPeer getPeer () ★

The `getPeer()` method retrieves the platform-specific peer object. The object `FontPeer` is a platform-specific subclass of `sun.awt.PlatformFont`. For example, on a Windows 95 platform, this would be an instance of `sun.awt.windows.WFontPeer`.

Styles

public boolean isPlain ()

The `isPlain()` method returns `true` if the current font is neither bold nor italic. Otherwise, it returns `false`.

public boolean isBold ()

The `isBold()` method returns `true` if the current font is either bold or bold and italic. Otherwise, it returns `false`.

public boolean isItalic ()

The `isItalic()` method returns `true` if the current font is either italic or bold and italic. Otherwise, it returns `false`.

Font properties

Earlier, you saw how to use system properties to add aliases for fonts. In addition to adding aliases, you can use system properties to specify which fonts your program will use when it runs. This allows your users to customize their environments to their liking; your program reads the font settings at run-time, rather than using hard-coded settings. The format of the settings in a properties file is:

```
propname=fontname--style--size
```

where `propname` is the name of the property being set, `fontname` is any valid font name (including aliases), `style` is `plain`, `bold`, `italic`, or `bolditalic`, and `size` represents the desired size for the font. `style` and `size` default to `plain` and 12 points. Order is important; the font's style must always precede its size.

For example, let's say you have three areas on your screen: one for menus, one for labels, and one for input. In the system properties, you allow users to set three properties: `myPackage.myClass.menuFont`, `myPackage.myClass.labelFont`, and `myPackage.myClass.inputFont`. One user sets two:

```
myPackage.myClass.menuFont=TimesRoman-italic-24
myPackage.myClass.inputFont=Helvetica
```

The user has specified a Times font for menus and Helvetica for other input. The property names are up to the developer. The program uses `getFont()` to read the properties and set the fonts accordingly.

NOTE:

The location of the system properties file depends on the run-time environment and version you are using. Normally, the file goes into a subdirectory of the installation directory, or for environments where users have home directories, in a subdirectory for the user. Sun's HotJava, JDK, and *appletviewer* tools use the *properties* file in the *.hotjava* directory.

Most browsers do not permit modifying properties, so there is no file.

Java 1.1 adds the idea of "resource files," which are syntactically similar to properties files. Resource files are then placed on the server or within a directory found in the `CLASSPATH`. Updating the properties file is no longer recommended.

public static Font getFont (String name)

The getFont() method gets the font specified by the system property name. If name is not a valid system property, null is returned. This method is implemented by a call to the next version of getFont(), with the defaultFont parameter set to null.

Assuming the properties defined in the previous example, if you call the getFont() method with name set to myPackage.myClass.menuFont, the return value is a 24-point, italic, TimesRoman Font object. If called with name set to myPackage.myClass.inputFont, getFont() returns a 12-point, plain Helvetica Font object. If called with myPackage.myClass.labelFont as name, getFont() returns null because this user did not set the property myPackage.myClass.labelFont.

public static Font getFont (String name, Font defaultFont)

The getFont() method gets the font specified by the system property name. If name is not a valid system property, this version of getFont() returns the Font specified by defaultFont. This version allows you to provide defaults in the event the user does not wish to provide his own font settings.

public static Font decode (String name) ★

The decode() method provides an explicit means to decipher font property settings, regardless of where the setting comes from. (The getFont() method can decipher settings, but only if they're in the system properties file.) In particular, you can use decode() to look up font settings in a resource file. The format of name is the same as that used by getFont(). If the contents of name are invalid, a 12-point plain font is returned. To perform the equivalent of getFont(`myPackage.myClass.menuFont`) without using system properties, see the following example. For a more extensive example using resource files, see Appendix A.

```
// Java 1.1 only
InputStream is = instance.getClass().getResourceAsStream("propfile");
Properties p = new Properties();
try {
    p.load (is);
    Font f = Font.decode(p.getProperty("myPackage.myClass.menuFont"));
} catch (IOException e) {
    System.out.println ("error loading props...");
}
```

Miscellaneous methods

public int hashCode ()

The hashCode() method returns a hash code for the font. This hash code is used whenever a Font object is used as the key in a `Hashtable`.

public boolean equals (Object o)

The `equals()` method overrides the `equals()` method of `Object` to define equality for `Font` objects. Two `Font` objects are equal if their size, style, and name are equal. The following example demonstrates why this is necessary.

```
Font a = new Font ("TimesRoman", Font.PLAIN, 10);
Font b = new Font ("TimesRoman", Font.PLAIN, 10);
// displays false since the objects are different objects
System.out.println (a == b);
// displays true since the objects have equivalent settings
System.out.println (a.equals (b));
```

public String toString ()

The `toString()` method of `Font` returns a string showing the current family, name, style, and size settings. For example:

```
java.awt.Font[family=TimesRoman,name=TimesRoman,style=bolditalic,size=20]
```

PREVIOUS HOME NEXT
MediaTracker BOOK INDEX FontMetrics

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

4. Events

Contents:

This chapter covers Java's event-driven programming model. Unlike procedural programs, windows-based programs require an event-driven model in which the underlying environment tells your program when something happens. For example, when the user clicks on the mouse, the environment generates an event that it sends to the program. The program must then figure out what the mouse click means and act accordingly.

This chapter covers two different event models, or ways of handling events. In Java 1.0.2 and earlier, events were passed to all components that could possibly have an interest in them. Events themselves were encapsulated in a single `Event` class. Java 1.1 implements a "delegation" model, in which events are distributed only to objects that have been registered to receive the event. While this is somewhat more complex, it is much more efficient and also more flexible, because it allows any object to receive the events generated by a component. In turn, this means that you can separate the user interface itself from the event-handling code.

In the Java 1.1 event model, all event functionality is contained in a new package, `java.awt.event`. Within this package, subclasses of the abstract class `AWTEvent` represent different kinds of events. The package also includes a number of `EventListener` interfaces that are implemented by classes that want to receive different kinds of events; they define the methods that are called when events of the appropriate type occur. A number of adapter classes are also included; they correspond to the `EventListener` interfaces and provide null implementations of the methods in the corresponding listener. The adapter classes aren't essential but provide a convenient shortcut for developers; rather than declaring that your class implements a particular `EventListener` interface, you can declare that your class extends the appropriate adapter.

The old and new event models are incompatible. Although Java 1.1 supports both, you should not use both models in the same program.

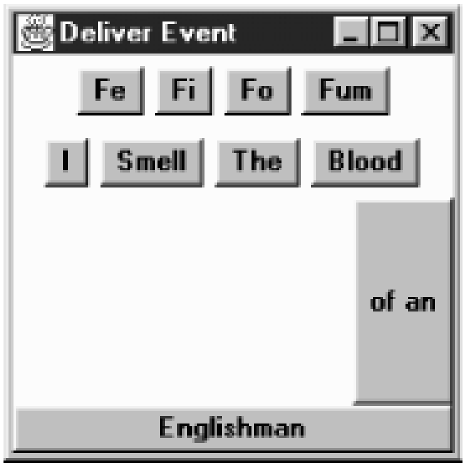
4.1 Java 1.0 Event Model

The event model used in versions 1.0 through 1.0.2 of Java is fairly simple. Upon receiving a user-initiated event, like a mouse click, the system generates an instance of the Event class and passes it along to the program. The program identifies the event's target (i.e., the component in which the event occurred) and asks that component to handle the event. If the target can't handle this event, an attempt is made to find a component that can, and the process repeats. That is all there is to it. Most of the work takes place behind the scenes; you don't have to worry about identifying potential targets or delivering events, except in a few special circumstances. Most Java programs only need to provide methods that deal with the specific events they care about.

Identifying the Target

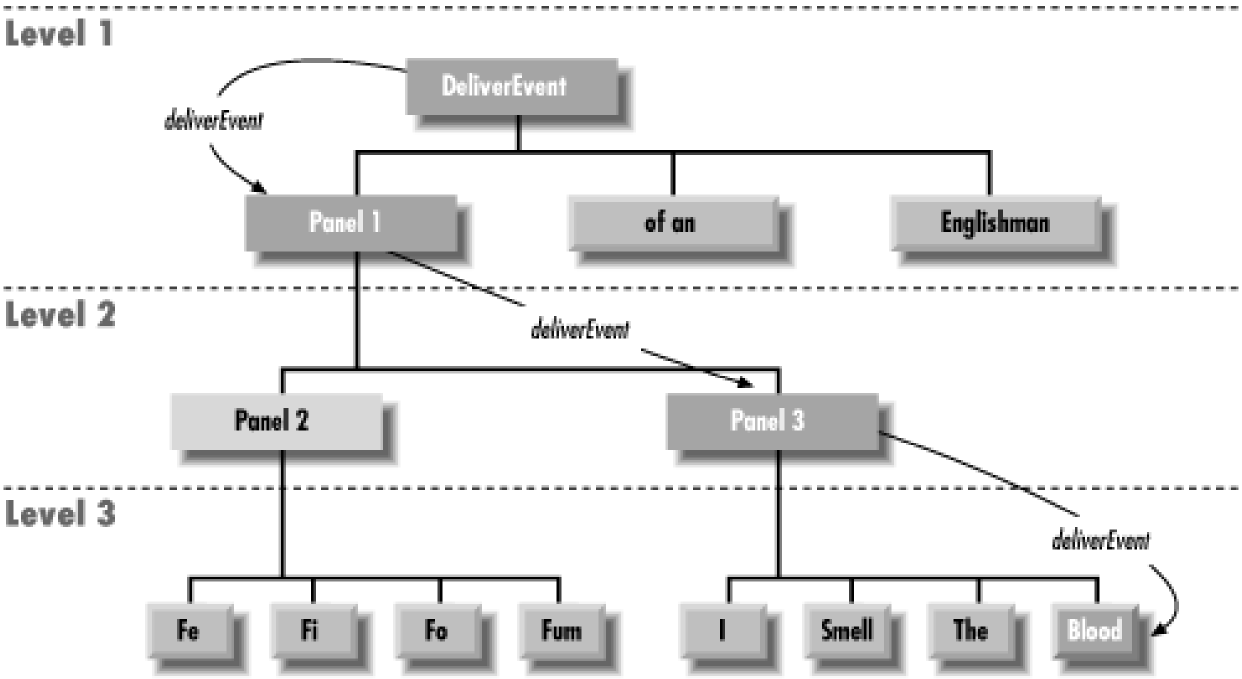
All events occur within a Java Component. The program decides which component gets the event by starting at the outermost level and working in. In Figure 4.1, assume that the user clicks at the location (156, 70) within the enclosing Frame's coordinate space. This action results in a call to the Frame's deliverEvent() method, which determines which component within the frame should receive the event and calls that component's deliverEvent() method. In this case, the process continues until it reaches the Button labeled Blood, which occupies the rectangular space from (135, 60) to (181, 80). Blood doesn't contain any internal components, so it must be the component for which the event is intended. Therefore, an action event is delivered to Blood, with its coordinates translated to fit within the button's coordinate space--that is, the button receives an action event with the coordinates (21, 10). If the user clicked at the location (47, 96) within the Frame's coordinate space, the Frame itself would be the target of the event because there is no other component at this location.

Figure 4.1: deliverEvent



To reach Blood, the event follows the component/container hierarchy shown in Figure 4.2.

Figure 4.2: deliverEvent screen model



Dealing With Events

Once `deliverEvent()` identifies a target, it calls that target's `handleEvent()` method (in this case, the `handleEvent()` method of Blood) to deliver the event for processing. If Blood has not overridden `handleEvent()`, its default implementation would call Blood's `action()` method. If Blood has not overridden `action()`, its default implementation (which is inherited from `Component`) is executed and does nothing. For your program to respond to the event, you would have to provide your own implementation of `action()` or `handleEvent()`.

`handleEvent()` plays a particularly important role in the overall scheme. It is really a dispatcher, which looks at the type of event and calls an appropriate method to do the actual work: `action()` for action events, `mouseUp()` for mouse up events, and so on. Table 4.1 shows the event-handler methods you would have to override when using the default `handleEvent()` implementation. If you create your own `handleEvent()`, either to handle an event without a default handler or to process events differently, it is best to leave these naming conventions in place. Whenever you override an event-handler method, it is a good idea to call the overridden method to ensure that you don't lose any functionality. All of the event handler methods return a boolean, which determines whether there is any further event processing; this is described in the next section, "Passing the Buck."

Table 4.1: Event Types and Event Handlers

Event Type	**Event Handler**
`MOUSE_ENTER`	`mouseEnter()`
`MOUSE_EXIT`	`mouseExit()`
`MOUSE_MOVE`	`mouseMove()`
`MOUSE_DRAG`	`mouseDrag()`
`MOUSE_DOWN`	`mouseDown()`

MOUSE_UP	mouseUp()
KEY_PRESS	keyDown()
KEY_ACTION	keyDown()
KEY_RELEASE	keyUp()
KEY_ACTION_RELEASE	keyUp()
GOT_FOCUS	gotFocus()
LOST_FOCUS	lostFocus()
ACTION_EVENT	action()

Passing the Buck

In actuality, `deliverEvent()` does not call `handleEvent()` directly. It calls the `postEvent()` method of the target component. In turn, `postEvent()` manages the calls to `handleEvent()`. `postEvent()` provides this additional level of indirection to monitor the return value of `handleEvent()`. If the event handler returns `true`, the handler has dealt with the event completely. All processing has been completed, and the system can move on to the next event. If the event handler returns `false`, the handler has not completely processed the event, and `postEvent()` will contact the component's `Container` to finish processing the event. Using the screen in Figure 4.1 as the basis, Example 4.1 traces the calls through `deliverEvent()`, `postEvent()`, and `handleEvent()`. The action starts when the user clicks on the Blood button at coordinates (156, 70). In short, Java dives into the depths of the screen's component hierarchy to find the target of the event (by way of the method `deliverEvent()`). Once it locates the target, it tries to find something to deal with the event by working its way back out (by way of `postEvent()`, `handleEvent()`, and the convenience methods). As you can see, there's a lot of overhead, even in this relatively simple example. When we discuss the Java 1.1 event model, you will see that it has much less overhead, primarily because it doesn't need to go looking for a component to process each event.

Example 4.1: The deliverEvent, postEvent, and handleEvent Methods

```
DeliverEvent.deliverEvent (Event e) called
    DeliverEvent.locate (e.x, e.y)
    Finds Panel1
    Translate Event Coordinates for Panel1
    Panel1.deliverEvent (Event e)
        Panel1.locate (e.x, e.y)
        Finds Panel3
        Translate Event Coordinates for Panel3
        Panel3.deliverEvent (Event e)
            Panel3.locate (e.x, e.y)
            Finds Blood
            Translate Event Coordinates for Blood
            Blood.deliverEvent (Event e)
                Blood.postEvent (Event e)
                    Blood.handleEvent (Event e)
```

```
                        Blood.mouseDown   (Event e, e.x, e.y)
                            returns false
                        return false
                    Get parent Container Panel3
                    Translate Event Coordinates for Panel3
                    Panel3.postEvent (Event e)
                        Panel3.handleEvent (Event e)
                            Component.mouseDown (Event e, e.x, e.y)
                                returns false
                            return false
                        Get parent Container Panel1
                        Translate Event Coordinates for Panel1
                        Panel1.postEvent (Event e)
                            Panel1.handleEvent (Event e)
                                Component.action (Event e, e.x, e.y)
                                    return false
                                return false
                            Get parent Container DeliverEvent
                            Translate Event Coordinates for DeliverEvent
                            DeliverEvent.postEvent (Event e)
                                DeliverEvent.handleEvent
                                    DeliverEvent.action (Event e, e.x, e.y)
                                        return true
                                    return true
                                return true
                            return true
                        return true
                    return true
                return true
            return true
        return true
    return true
```

Overriding handleEvent()

In many programs, you only need to override convenience methods like `action()` and `mouseUp()`; you usually don't need to override `handleEvent()`, which is the high level event handler that calls the convenience methods. However, convenience methods don't exist for all event types. To act upon an event that doesn't have a convenience method (for example, `LIST_SELECT`), you need to override `handleEvent()` itself. Unfortunately, this presents a problem. Unlike the convenience methods, for which the default versions don't take any action, `handleEvent()` does quite a lot: as we've seen, it's the dispatcher that calls the convenience methods. Therefore, when you override `handleEvent()`, either you should reimplement all the features of the method you are overriding (a very bad idea), or you must make sure that the original `handleEvent()`is still executed to ensure that the remaining events get handled properly. The simplest way for you to do this is for your new `handleEvent()` method to act on any events that it is interested in and return `true` if it has handled those events completely. If the incoming event is not an event that your handleEvent() is interested in, you should call super.handleEvent() and return its return value. The following code shows how you might override handleEvent() to deal with a LIST_SELECT event:

```
public boolean handleEvent (Event e) {
    if (e.id == Event.LIST_SELECT) {   // take care of LIST_SELECT
        System.out.println ("Selected item: " + e.arg);
        return true;    // LIST_SELECT handled completely; no further action
    } else {   // make sure we call the overridden method to ensure
                // that other events are handled correctly
        return super.handleEvent (e);
    }
}
```

Basic Event Handlers

The convenience event handlers like mouseDown(), keyUp(), and lostFocus() are all implemented by the Component class. The default versions of these methods do nothing and return false. Because these methods do nothing by default, when overriding them you do not have to ensure that the overridden method gets called. This simplifies the programming task, since your method only needs to return false if it has not completely processed the event. However, if you start to subclass nonstandard components (for example, if someone has created a fancy AudioButton, and you're subclassing that, rather than the standard Button), you probably should explicitly call the overridden method. For example, if you are overriding mouseDown(), you should include a call to super.mouseDown(), just as we called super.handleEvent() in the previous example. This call is "good housekeeping"; most of the time, your program will work without it. However, your program will break as soon as someone changes the behavior of the AudioButton and adds some feature to its mouseDown() method. Calling the super class's event handler helps you write "bulletproof " code.

The code below overrides the mouseDown() method. I'm assuming that we're extending a standard component, rather than extending some custom component, and can therefore dispense with the call to super.mouseDown().

```
public boolean mouseDown (Event e, int x, int y) {
    System.out.println ("Coordinates: " + x + "-" + y);
    if ((x > 100) || (y < 100))
        return false;    // we're not interested in this event; pass it on
    else                 // we're interested;
        ...              // this is where event-specific processing goes
        return true;     // no further event processing
}
```

Here's a debugging hint: when overriding an event handler, make sure that the parameter types are correct--remember that each convenience method has different parameters. If your overriding method has parameters that don't match the original method, the program will still compile correctly. However, it won't work. Because the parameters don't match, your new method simply overloads the original, rather than overriding it. As a result, your method will never be called.

PREVIOUS Using Desktop Colors

HOME

BOOK INDEX

NEXT The Event Class

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

5. Components

Contents:

This chapter introduces the generic graphical widget used within the AWT package, `Component`, along with a trio of specific components: `Label`, `Button`, and `Canvas`. It also covers the `Cursor` class, new to Java 1.1. (Cursor support was previously part of the `Frame` class.) Although many objects within AWT don't subclass `Component`, and though you will never create an instance of `Component`, anything that provides screen-based user interaction and relies on the system for its layout will be a child of `Component`. As a subclass of `Component`, each child inherits a common set of methods and an API for dealing with the different events (i.e., mouse click, keyboard input) that occur within your Java programs.

After discussing the methods in `Component` classes, this chapter goes into detail about two specific components, `Label` and `Button`. A `Label` is a widget that contains descriptive text, usually next to an input field. A `Button` is a basic mechanism that lets the user signal the desire to perform an action. You will learn about the `Canvas` object and how to use a `Canvas` to create your own component. Finally, we cover the `Cursor` class, which lets you change the cursor over a `Component`.

Before going into the mechanics of the `Component` class, it's necessary to say a little about the relationship between components and containers. A `Container` is also a component with the ability to contain other components. There are several different kinds of containers; they are discussed in Chapter 6, *Containers*. To display a component, you have to put it in a container by calling the container's `add()` method. We often call the container that holds a component the component's *parent* ; likewise, we call the components a container holds its *children*. Certain operations are legal only if a component has a parent--that is, the component is in a container. Of course, since containers are components, containers can contain other containers, *ad infinitum*.

NOTE:

If you think some component is missing a method that should obviously be there, check the methods it inherits. For example, the `Label` class appears to lack a `setFont()` method. Obviously, labels ought to be able to change their fonts. The `setFont()` method really is there; it is inherited from the `Component` class, and therefore, not documented as part of the `Label` class. Even if you're familiar with object-oriented techniques, the need to work up a class hierarchy to find all of the class's methods can lead to confusion and frustration. While all Java objects inherit methods from other classes, the potential for confusion is worst with components, which inherit over a hundred methods from `Component` and may only have a few methods of their own.

5.1 Component

Every GUI-based program consists of a screen with a set of objects. With Java, these objects are called components. Some of the more frequently used components are buttons, text fields, and containers.

A container is a special component that allows you to group different components together within it. You will learn more about containers in the next chapter, but they are in fact just another kind of component. Also, some of the parameters and return types for the methods of `Component` have not been explained yet and have their own sections in future chapters.

Component Methods

Constants

Prior to Java 1.1, you could not subclass `Component` or `Container`. With the introduction of the `LightweightPeer`, you can now subclass either `Component` or `Container`. However, since you no longer have a native peer, you must rely on your container to provide a display area and other services that are normally provided by a full-fledged peer. Because you cannot rely on your peer to determine your alignment, the `Component` class now has five constants to indicate six possible alignment settings (one constant is used twice). The alignment constants designate where to position a lightweight component; their values range from 0.0 to 1.0. The lower the number, the closer the component will be placed to the origin (top left corner) of the space allotted to it.[1]

[1] As of Beta 3, these constants appear to be seldom used. The `getAlignmentX()` and `getAlignmentY()` methods return these values, but there are no `setAlignment` methods.

public static final float BOTTOM_ALIGNMENT ★

> The `BOTTOM_ALIGNMENT` constant indicates that the component should align itself to the bottom of its available space. It is a return value from the method `getAlignmentY()`.

public static final float CENTER_ALIGNMENT ★

> The `CENTER_ALIGNMENT` constant indicates that the component should align itself to the middle of its available space. It is a return value from either the `getAlignmentX()` or `getAlignmentY()` method. This constant represents both the horizontal and vertical center.

public static final float LEFT_ALIGNMENT ★

> The `LEFT_ALIGNMENT` constant indicates that the component should align itself to the left side of its available space. It is a return value from `getAlignmentX()`.

public static final float RIGHT_ALIGNMENT ★

> The `RIGHT_ALIGNMENT` constant indicates that the component should align itself to the right side of its available space. It is a return value from the method `getAlignmentX()`.

public static final float TOP_ALIGNMENT ★

> The `TOP_ALIGNMENT` constant indicates that the component should align itself to the top of its available space. It is a return value from `getAlignmentY()`.

Variables

protected Locale locale ★

> The protected `locale` variable can be accessed by calling the `getLocale()` method.

Constructor

Prior to Java 1.1, there was no public or protected constructor for `Component`. Only package members were able to subclass `Component` directly. With the introduction of lightweight peers, components can exist without a native peer, so the constructor was made `protected`, allowing you to create your own `Component` subclasses.

protected Component() ★

> The constructor for `Component` creates a new component without a native peer. Since you no longer have a native peer, you must rely on your container to provide a display area. This allows you to create components that require fewer system resources than components that subclass `Canvas`. The example in the "Using an event multicaster" section of the previous chapter is of a lightweight component. Use the `SystemColor` class to help you colorize the new component appropriately or make it transparent.

Appearance

public Toolkit getToolkit ()

The `getToolkit()` method returns the current `Toolkit` of the `Component`. This returns the parent's `Toolkit` (from a `getParent()` call) when the `Component` has not been added to the screen yet or is lightweight. If there is no parent, `getToolkit()` returns the default `Toolkit`. Through the `Toolkit`, you have access to the details of the current platform (like screen resolution, screen size, and available fonts), which you can use to adjust screen real estate requirements or check on the availability of a font.

public Color getForeground ()

The `getForeground()` method returns the foreground color of the component. If no foreground color is set for the component, you get its parent's foreground color. If none of the component's parents have a foreground color set, `null` is returned.

public void setForeground (Color c)

The `setForeground()` method changes the current foreground color of the area of the screen occupied by the component to `c`. After changing the color, it is necessary for the screen to refresh before the change has any effect. To refresh the screen, call `repaint()`.

public Color getBackground ()

The `getBackground()` method returns the background color of the component. If no background color is set for the component, its parent's background color is retrieved. If none of the component's parents have a background color set, `null` is returned.

public void setBackground (Color c)

The `setBackground()` method changes the current background color of the area of the screen occupied by the component to `c`. After changing the color, it is necessary for the screen to refresh before the change has any affect. To refresh the screen, call `repaint()`.

public Font getFont ()

The `getFont()` method returns the font of the component. If no font is set for the component, its parent's font is retrieved. If none of the component's parents have a font set, `null` is returned.

public synchronized void setFont (Font f)

The `setFont()` method changes the component's font to `f`. If the font family (such as TimesRoman) provided within `f` is not available on the current platform, the system uses a default font family, along with the supplied size and style (plain, bold, italic). Depending upon the platform, it may be necessary to refresh the component/screen before seeing any changes.

Changing the font of a component could have an affect on the layout of the component.

public synchronized ColorModel getColorModel ()

The `getColorModel()` method returns the `ColorModel` used to display the current component. If the component is not displayed, the `ColorModel` from the component's `Toolkit` is used. The normal `ColorModel` for a Java program is 8 bits each for red, green, and blue.

public Graphics getGraphics ()

The `getGraphics()` method gets the component's graphics context. Most noncontainer components do not manage them correctly and therefore throw an `InternalError` exception when you call this method. The `Canvas` component is one that does since you can draw on that directly. If the component is not visible, `null` is returned.

public FontMetrics getFontMetrics (Font f)

The `getFontMetrics()` method retrieves the component's view of the `FontMetrics` for the requested font `f`. Through the `FontMetrics`, you have access to the platform-specific sizing for the appearance of a character or string.

public Locale getLocale () ★

The `getLocale()` method retrieves the current `Locale` of the component, if it has one. Using a `Locale` allows you to write programs that can adapt themselves to different languages and different regional variants. If no `Locale` has been set, `getLocale()` returns the parent's `Locale`.[2] If the component has no locale of its own and no parent (i.e., it isn't in a container), `getLocale()` throws the run-time exception `IllegalComponentStateException`.

> [2] For more on the `Locale` class, see the *Java Fundamental Classes Reference* from O'Reilly & Associates.

public void setLocale (Locale l) ★

The `setLocale()` method changes the current `Locale` of the component to `l`. In order for this change to have any effect, you must localize your components so that they have different labels or list values for different environments. Localization is part of the broad topic of internationalization and is beyond the scope of this book.

public Cursor getCursor () ★

> The `getCursor()` method retrieves the component's current `Cursor`. If one hasn't been set, the default is `Cursor.DEFAULT_CURSOR`. The `Cursor` class is described fully in [Cursor]. Prior to Java 1.1, the ability to associate cursors with components was restricted to frames.

public synchronized void setCursor (Cursor c) ★

> The `setCursor()` method changes the current `Cursor` of the component to `c`. The change takes effect as soon as the cursor is moved. Lightweight components cannot change their cursors.

Positioning/Sizing

`Component` provides a handful of methods for positioning and sizing objects. Most of these are used behind the scenes by the system. You will also need them if you create your own `LayoutManager` or need to move or size an object. All of these depend on support for the functionality from the true component's peer.

public Point getLocation () ★
public Point location () ☆

> The `getLocation()` method returns the current position of the `Component` in its parent's coordinate space. The `Point` is the top left corner of the bounding box around the `Component`.
>
> `location()`is the Java 1.0 name for this method.

public Point getLocationOnScreen () ★

> The `getLocationOnScreen()` method returns the current position of the `Component` in the screen's coordinate space. The `Point` is the top left corner of the bounding box around the `Component`. If the component is not showing, the `getLocationOnScreen()` method throws the `IllegalComponentStateException` run-time exception.

public void setLocation (int x, int y) ★
public void move (int x, int y) ☆

> The `setLocation()` method moves the `Component` to the new position (`x`, `y`). The coordinates provided are in the parent container's coordinate space. This method calls `setBounds()` to move the component. The `LayoutManager` of the container may make it impossible to change a component's location.
>
> Calling this method with a new position for the component generates a `ComponentEvent` with the ID `COMPONENT_MOVED`.

`move()`is the Java 1.0 name for this method.

public void setLocation (Point p) ★

This `setLocation()` method moves the component to the position specified by the given `Point`. It is the same as calling `setLocation(p.x, p.y)`.

Calling this method with a new position for the component generates a `ComponentEvent` with the ID `COMPONENT_MOVED`.

public Dimension getSize () ★
public Dimension size () ☆

The `getSize()` method returns the width and height of the component as a `Dimension` object.

`size()`is the Java 1.0 name for this method.

public void setSize (int width, int height) ★
public void resize (int width, int height) ☆

The `setSize()` method changes the component's width and height to the `width` and `height` provided. `width` and `height` are specified in pixels. The component is resized by a call to `setBounds()`. The `LayoutManager` of the `Container` that contains the component may make it impossible to change a component's size.

Calling this method with a new size for the component generates a `ComponentEvent` with the ID `COMPONENT_RESIZED`.

`resize()`is the Java 1.0 name for this method.

public void setSize (Dimension d) ★
public void resize (Dimension d) ☆

This `setSize()` method changes the component's width and height to the `Dimension d` provided. The `Dimension` object includes the width and height attributes in one object. The component is resized by a call to the `setBounds()` method. The `LayoutManager` of the `Container` that contains the component may make it impossible to change a component's size.

Calling this method with a new size for the component generates a `ComponentEvent` with the ID `COMPONENT_RESIZED`.

`resize()`is the Java 1.0 name for this method.

public Rectangle getBounds () ★
public Rectangle bounds () ☆

The `getBounds()` method returns the bounding rectangle of the object. The fields of the `Rectangle` that you get back contain the component's position and dimensions.

`bounds()`is the Java 1.0 name for this method.

public void setBounds (int x, int y, int width, int height) ★
public void reshape (int x, int y, int width, int height) ☆

The `setBounds()` method moves and resizes the component to the bounding rectangle with coordinates of (`x`, `y`) (top left corner) and `width` x `height`. If the size and shape have not changed, no reshaping is done. If the component is resized, it is invalidated, along with its parent container. The `LayoutManager` of the `Container` that contains the component may make it impossible to change the component's size or position. Calling `setBounds()` invalidates the container, which results in a call to the `LayoutManager` to rearrange the container's contents. In turn, the `LayoutManager` calls `setBounds()` to give the component its new size and position, which will probably be the same size and position it had originally. In short, if a layout manager is in effect, it will probably undo your attempts to change the component's size and position.

Calling this method with a new size for the component generates a `ComponentEvent` with the ID `COMPONENT_RESIZED`. Calling this method with a new position generates a `ComponentEvent` with the ID `COMPONENT_MOVED`.

`reshape()`is the Java 1.0 name for this method.

public void setBounds (Rectangle r) ★

This `setBounds()` method calls the previous method with parameters of `r.x`, `r.y`, `r.width`, and `r.height`.

Calling this method with a new size for the component generates a `ComponentEvent` with the ID `COMPONENT_RESIZED`. Calling this method with a new position generates a `ComponentEvent` with the ID `COMPONENT_MOVED`.

public Dimension getPreferredSize () ★
public Dimension preferredSize () ☆

The `getPreferredSize()` method returns the `Dimension` (width and height) for the preferred size of the component. Each component's peer knows its preferred size. Lightweight objects return `getSize()`.

`preferredSize()`is the Java 1.0 name for this method.

public Dimension getMinimumSize () ★
public Dimension minimumSize () ☆

The `getMinimumSize()` method returns the `Dimension` (width and height) for the minimum size of the component. Each component's peer knows its minimum size. Lightweight objects return `getSize()`. It is possible that the methods `getMinimumSize()` and `getPreferredSize()` will return the same dimensions.

`minimumSize()`is the Java 1.0 name for this method.

public Dimension getMaximumSize () ★

The `getMaximumSize()` method returns the `Dimension` (width and height) for the maximum size of the component. This may be used by a layout manager to prevent a component from growing beyond a predetermined size. None of the `java.awt` layout managers call this method. By default, the value returned is `Short.MAX_VALUE` for both dimensions.

public float getAlignmentX () ★

The `getAlignmentX()` method returns the alignment of the component along the x axis. The alignment could be used by a layout manager to position this component relative to others. The return value is between 0.0 and 1.0. Values nearer 0 indicate that the component should be placed closer to the left edge of the area available. Values nearer 1 indicate that the component should be placed closer to the right. The value 0.5 means the component should be centered. The default setting is `Component.CENTER_ALIGNMENT`.

public float getAlignmentY () ★

The `getAlignmentY()` method returns the alignment of the component along the y axis. The alignment could be used by a layout manager to position this component relative to others. The return value is between 0.0 and 1.0. Values nearer 0 indicate that the component should be placed closer to the top of the area available. Values nearer 1 indicate that the component should be placed closer to the bottom. The value 0.5 means the component should be centered. The default setting is `Component.CENTER_ALIGNMENT`.

public void doLayout () ★
public void layout () ☆

The `doLayout()` method of `Component` does absolutely nothing. It is called when the `Component` is validated (through the `validate()` method). The `Container` class overrides this method.

`layout()`is the Java 1.0 name for this method.

public boolean contains (int x, int y) ★
public boolean inside (int x, int y) ☆

The `contains()` method checks if the `x` and `y` coordinates are within the bounding box of the component. If the `Component` is not rectangular, the method acts as if there is a rectangle around the `Component`. `contains()` returns `true` if the `x` and `y` coordinates are within the component, `false` otherwise.

`inside()`is the Java 1.0 name for this method.

public boolean contains (Point p) ★

This `contains()` method calls the previous method with parameters of `p.x` and `p.y`.

public Component getComponentAt (int x, int y) ★
public Component locate (int x, int y) ☆

The `getComponentAt()` method uses `contains()` to see if the `x` and `y` coordinates are within the component. If they are, this method returns the `Component`. If they aren't, it returns `null`. `getComponentAt()` is overridden by `Container` to provide enhanced functionality.

`locate()`is the Java 1.0 name for this method.

public Component getComponentAt (Point p) ★

This `getComponentAt()` method calls the previous method with parameters of `p.x` and `p.y`.

Painting

The only methods in this section that you call directly are the versions of `repaint()`. The `paint()` and `update()` methods are called by the system when the display area requires refreshing, such as when a user resizes a window. When your program changes the display you should call `repaint()` to trigger a call to `update()` and `paint()`. Otherwise, the system is responsible for updating the display.

public void paint (Graphics g)

The `paint()` method is offered so the system can display whatever you want in a `Component`. In the base `Component` class, this method does absolutely nothing. Ordinarily, it would be overridden in an applet to do something other than the default, which is display a box in the current background color. `g` is the graphics context of the component being drawn on.

public void update (Graphics g)

The `update()` method is automatically called when you ask to repaint the `Component`. If the component is not lightweight, the default implementation of `update()` clears graphics context `g` by drawing a filled rectangle in the background color, resetting the color to the current foreground color, and calling `paint()`. If you do not override `update()` when you do animation, you will see some flickering because `Component` clears the screen. Animation is discussed in Chapter 2, *Simple Graphics*.

public void paintAll (Graphics g)

The `paintAll()` method validates the component and paints its peer if it is visible. `g` represents the graphics context of the component. This method is called when the `paintComponents()` method of `Container` is called.

public void repaint ()

The `repaint()` method requests the scheduler to redraw the component as soon as possible. This will result in `update()` getting called soon thereafter. There is not a one-to-one correlation between `repaint()` and `update()` calls. It is possible that multiple `repaint()` calls can result in a single `update()`.

public void repaint (long tm)

This version of `repaint()` allows for a delay of `tm` milliseconds. It says, please update this component within `tm` milliseconds, which may happen immediately.

public void repaint (int x, int y, int width, int height)

This version of `repaint()` allows you to select the region of the `Component` you desire to be updated. (`x`, `y`) are the coordinates of the upper left corner of the bounding box of the component with dimensions of `width`x`height`. This is similar to creating a clipping area and results in a quicker repaint.

public void repaint (long tm, int x, int y, int width, int height)

This final version of `repaint()` is what the other three `repaint()` methods call. `tm` is the maximum delay in milliseconds before update should be called. (`x`, `y`) are the coordinates of the upper left corner of the clipping area of the component with dimensions of `width` x `height`.

public void print (Graphics g)

The default implementation of the `print()` method calls `paint()`.

In Java 1.0, there was no way to print; in Java 1.1, if the `graphics` parameter implements `PrintGraphics`, anything drawn on `g` will be printed. Printing is covered in Chapter 17, *Printing*.

public void printAll (Graphics g)

The `printAll()` method validates the component and paints its peer if it is visible. `g` represents the graphics context of the component. This method is called when the `printComponents()` method of `Container` is called or when you call it with a `PrintGraphics` parameter.

The default implementation of `printAll()` is identical to `paintAll()`. As with `paintAll()`, `g` represents the graphics context of the component; if `g` implements `PrintGraphics`, it can be printed.

Imaging

Background information about using images is discussed in Chapter 2, *Simple Graphics* and Chapter 12, *Image Processing*. The `imageUpdate()` method of `Component` is the sole method of the `ImageObserver` interface. Since images are loaded in a separate thread, this method is called whenever additional information about the image becomes available.

public boolean imageUpdate (Image image, int infoflags, int x, int y, int width, int height)

`imageUpdate()` is the `java.awt.image.ImageObserver` method implemented by `Component`. It is an asynchronous update interface for receiving notifications about `Image` information as `image` is loaded and is automatically called when additional information becomes available. This method is necessary because image loading is done in a separate thread from the `getImage()` call. Ordinarily, `x` and `y` would be the coordinates of the upper left corner of the image loaded so far, usually (0, 0). However, the method `imageUpdate()` of the component ignores these parameters. `width` and `height` are the `image`'s dimensions, so far, in the loading process.

The `infoflags` parameter is a bit-mask of information available to you about `image`. Please see the text about `ImageObserver` in Chapter 12, *Image Processing* for a complete description of the different flags that can be set. When overriding this method, you can wait for some condition to be true by checking a flag in your program and then taking the desired action. To check for a particular flag, perform an AND (&) of `infoflags` and the constant. For example, to check if the `FRAMEBITS` flag is set:

```
if ((infoflags & ImageObserver.FRAMEBITS) == ImageObserver.FRAMEBITS)
    System.out.println ("The Flag is set");
```

The return value from a call to `imageUpdate()` is `true` if `image` has changed and `false` otherwise.

Two system properties let the user control the behavior of updates:

- `awt.image.incrementaldraw` allows the user to control whether or not partial images are displayed. Initially, the value of `incrementaldraw` is unset and defaults to `true`, which means that partial images are drawn. If `incrementaldraw` is set to `false`, the image will be drawn only when it is complete or when the screen is resized or refreshed.

- `awt.image.redrawrate` allows the user to change the delay between successive repaints. If not set, the default redraw rate is 100 milliseconds.

public Image createImage (int width, int height)

The `createImage()` method creates an empty `Image` of size `width` x `height`. The returned `Image` is an in-memory image that can be drawn on for double buffering to manipulate an image in the background. If an image of size `width` x `height` cannot be created, the call returns `null`. In order for `createImage()` to succeed, the peer of the `Component` must exist; if the component is lightweight, the peer of the component's container must exist.

public Image createImage (ImageProducer producer)

This `createImage()` method allows you to take an existing image and modify it in some way to produce a new `Image`. This can be done through `ImageFilter` and `FilteredImageSource` or a `MemoryImageSource`, which accepts an array of pixel information. You can learn more about these classes and this method in Chapter 12, *Image Processing*.

public boolean prepareImage (Image image, ImageObserver observer)

The `prepareImage()` method forces `image` to start loading, asynchronously, in another thread. `observer` is the `Component` that `image` will be rendered on and is notified (via `imageUpdate()`) as `image` is being loaded. In the case of an `Applet`, `this` would be passed as the `ImageObserver`. If `image` has already been fully loaded, `prepareImage()` returns `true`. Otherwise, `false` is returned. Since `image` is loaded asynchronously, `prepareImage()` returns immediately. Ordinarily, `prepareImage()` would be called by the system when `image` is first needed to be displayed (in `drawImage()` within `paint()`). As more information about the image gets loaded, `imageUpdate()` is called periodically.

If you do not want to go through the trouble of creating a `MediaTracker` instance to start the loading of the image objects, you can call `prepareImage()` to trigger the start of image loading prior to a call to `drawImage()`.

If `image` has already started loading when this is called or if this is an in-memory image, there is no effect.

public boolean prepareImage (Image image, int width, int height, ImageObserver observer)

This version of `prepareImage()` is identical to the previous one, with the addition of a scaling factor of `width`x`height`. As with other `width` and `height` parameters, the units for these parameters are pixels. Also, if `width` and `height` are -1, no scaling factor is assumed. This method is called by one of the internal `MediaTracker` methods.

public int checkImage (Image image, ImageObserver observer)

The `checkImage()` method returns the status of the construction of a screen representation of `image`, being watched by `observer`. If `image` has not started loading yet, this will not start it. The return value is the `ImageObserver` flags ORed together for the data that is now available. The available `ImageObserver` flags are: `WIDTH`, `HEIGHT`, `PROPERTIES`, `SOMEBITS`, `FRAMEBITS`, `ALLBITS`, `ERROR`, and `ABORT`. See Chapter 12, *Image Processing* for a complete description of `ImageObserver`.

public int checkImage (Image image, int width, int height, ImageObserver observer)

This version of `checkImage()` is identical to the previous one, with the addition of a scaling factor of `width`x`height`. If you are using the `drawImage()` version with `width` and `height` parameters, you should use this version of `checkImage()` with the same `width` and `height`.

Peers

public ComponentPeer getPeer () ☆

The `getPeer()` method returns a reference to the component's peer as a `ComponentPeer` object. For example, if you issue this method from a `Button` object, `getPeer()` returns an instance of the `ComponentPeer` subclass `ButtonPeer`.

This method is flagged as deprecated in comments but not with `@deprecated`. There is no replacement method for Java 1.1.

public void addNotify ()

The `addNotify()` method is overridden by each individual component type. When `addNotify()` is called, the peer of the component gets created, and the `Component` is invalidated. The `addNotify()` method is called by the system when it needs to create the peer. The peer needs to be created when a `Component` is first shown, or when a new `Component` is added to a `Container` and the `Container` is already being shown (in which case it already has a peer, but a new one must be created to take account of the new `Component`). If you override this method for a specific `Component`, call `super.addNotify()` first, then do what you need for the `Component`. You will then have information available about the newly created peer.

Certain tasks cannot succeed unless the peer has been created. An incomplete list includes finding the size of a component, laying out a container (because it needs the component's size), and creating an `Image` object. Peers are discussed in more depth in Chapter 15, *Toolkit and Peers*.

public synchronized void removeNotify ()

The `removeNotify()` method destroys the peer of the component and removes it from the screen. The state information about the `Component` is retained by the specific subtype. The `removeNotify()` method is called by the system when it determines the peer is no longer needed. Such times would be when the `Component` is removed from a `Container`, when its container changes, or when the `Component` is disposed. If you override this method for a specific `Component`, issue the particular commands for you need for this `Component`, then call `super.removeNotify()` last.

State Procedures

These methods determine whether the component is ready to be displayed and can be seen by the user. The first requirement is that it be *valid*--that is, whether the system knows its size, and (in the case of a container) whether the layout manager is aware of all its parts and has placed them as requested. A component becomes invalid if the size has changed since it was last displayed. If the component is a container, it becomes invalid when one of the components contained within it becomes invalid.

Next, the component must be *visible*--a possibly confusing term, because components can be considered "visible" without being seen by the user. Frames (because they have their own top-level windows) are not visible until you request that they be shown, but other components are visible as soon as you create them.

Finally, to be seen, a component must be *showing*. You show a component by adding it to its container. For something to be showing, it must be visible and be in a container that is visible and showing.

A subsidiary aspect of state is the *enabled* quality, which determines whether a component can accept input.

public boolean isValid ()

The `isValid()` method tells you whether or not the component needs to be laid out.

public void validate ()

The `validate()` method sets the component's valid state to `true`. Ordinarily, this is done for you when the `Component` is laid out by its `Container`. Since objects are invalid when they are first drawn on the screen, you should call `validate()` to tell the system you are finished adding objects so that it can validate the screen and components. One reason you can override `validate()` is to find out when the container that the component exists in has been resized. The only requirement when overriding is that the original `validate()` be called. With Java 1.1, instead of overriding, you can listen for resize events.

public void invalidate ()

The `invalidate()` method sets the component's valid state to `false` and propagates the invalidation to its parent. Ordinarily, this is done for you, or should be, whenever anything that affects the layout is changed.

public boolean isVisible ()

The `isVisible()` methods tells you if the component is currently visible. Most components are initially visible, except for top-level objects like frames. Any component that is visible will be shown on the screen when the screen is painted.

public boolean isShowing ()

The `isShowing()` method tells you if the component is currently shown on the screen. It is possible for `isVisible()` to return `true` and `isShowing()` to return `false` if the screen has not been painted yet.

Table 5.1 compares possible return values from `isVisible()` and `isShowing()`. The first two entries are for objects that have their own `Window`. These will always return the same values for `isVisible()` and `isShowing()`. The next three are for `Component` objects that exist within a `Window`, `Panel`, or `Applet`. The visible setting is always initially `true`. However, the showing setting is not true until the object is actually drawn. The last case shows another possibility. If the component exists within an invisible `Container`, the component will be visible but will not be shown.

Table 5.1: isVisible vs. isShowing

Happenings	**isVisible**	**isShowing**
`Frame` created	`false`	`false`
`Frame f = new Frame ()`		
`Frame` showing	`true`	`true`
`f.show ()`		
`Component` created	`true`	`false`
`Button b= new Button ("Help")`		

`Button` added to screen in `init()`	`true`	`false`
`add (b)`		
`Container` laid out with `Button` in it	`true`	`true`
`Button` within `Panel` that is not visible	`true`	`false`

public void show ()

The `show()` method displays a component by making it visible and showing its peer. The parent `Container` becomes invalid because the set of children to display has changed. You would call `show()` directly to display a `Frame` or `Dialog`.

In Java 1.1, you should use `setVisible()` instead.

public void hide ()

The `hide()` method hides a component by making it invisible and hiding its peer. The parent `Container` becomes invalid because the set of children to display has changed. If you call `hide()` for a `Component` that does not subclass `Window`, the component's `Container` reserves space for the hidden object.

In Java 1.1, you should use `setVisible()` instead.

public void setVisible(boolean condition) ★
public void show (boolean condition) ☆

The `setVisible()` method calls either `show()` or `hide()` based on the value of `condition`. If `condition` is `true`, `show()` is called. When `condition` is `false`, `hide()` is called.

`show()` is the Java 1.0 name for this method.

public boolean isEnabled ()

The `isEnabled()` method checks to see if the component is currently enabled. An enabled `Component` can be selected and trigger events. A disabled `Component` usually has a slightly lighter font and doesn't permit the user to select or interact with it. Initially, every `Component` is enabled.

public synchronized void enable ()

The `enable()` method allows the user to interact with the component. Components are enabled by default but can be disabled by a call to `disabled()` or `setEnabled(false)`.

In Java 1.1, you should use `setEnabled()` instead.

public synchronized void disable ()

The `disable()` method disables the component so that it is unresponsive to user interactions.

In Java 1.1, you should use `setEnabled()` instead.

public void setEnabled (boolean condition) ★
public void enable (boolean condition) ☆

The `setEnabled()` method calls either `enable()` or `disable()` based on the value of `condition`. If `condition` is `true`, `enable()` is called. When `condition` is `false`, `disable()` is called. Enabling and disabling lets you create components that can be operated only under certain conditions--for example, a `Button` that can be pressed only after the user has typed into a `TextArea`.

`enable()` is the Java 1.0 name for this method.

Focus

Although there was some support for managing input focus in version 1.0, 1.1 improved on this greatly by including support for Tab and Shift+Tab to move input focus to the next or previous component, and by being more consistent across different platforms. This support is provided by the package-private class `FocusManager`.

public boolean isFocusTraversable() ★

The `isFocusTraversable()` method is the support method that tells you whether or not a component is capable of receiving the input focus. Every component asks its peer whether or not it is traversable. If there is no peer, this method returns `false`.

If you are creating a component by subclassing `Component` or `Canvas` and you want it to be traversable, you should override this method; a `Canvas` is not traversable by default.

public void requestFocus ()

The `requestFocus()` method allows you to request that a component get the input focus. If it can't (`isFocusTraversable()` returns `false`), it won't.

public void transferFocus () ★
public void nextFocus () ☆

The `transferFocus()` method moves the focus from the current component to the next one.

`nextFocus()` is the Java 1.0 name for this method.

Miscellaneous methods

public final Object getTreeLock () ★

The `getTreeLock()` method retrieves the synchronization lock for all AWT components. Instead of using `synchronized` methods in Java 1.1, previously synchronized methods lock the tree within a `synchronized (component.getTreeLock()) {}` code block. This results in a more efficient locking mechanism to improve performance.

public String getName () ★

The `getName()` method retrieves the current name of the component. The component's name is useful for object serialization. Components are given a name by default; you can change the name by calling `setName()`.

public void setName (String name) ★

The `setName()` method changes the name of the component to `name`.

public Container getParent ()

The `getParent()` method returns the component's `Container`. The container for anything added to an applet is the applet itself, since it subclasses `Panel`. The container for the applet is the browser. In the case of Netscape Navigator versions 2.0 and 3.0, the return value would be a specific instance of the `netscape.applet.EmbeddedAppletFrame` class. If the applet is running within the *appletviewer*, the return value would be an instance of `sun.applet.AppletViewerPanel`.

public synchronized void add(PopupMenu popup) ★

The `add()` method introduced in Java 1.1 provides the ability to associate a `PopupMenu` with a `Component`. The pop-up menu can be used to provide context-sensitive menus for specific components. (On some platforms for some components, pop-up menus exist already and cannot be overridden.) Interaction with the menu is discussed in Chapter 10, *Would You Like to Choose from the Menu?*

Multiple pop-up menus can be associated with a component. To display the appropriate pop-up menu, call the pop-up menu's `show()`method.

public synchronized void remove(MenuComponent popup) ★

The `remove()` method is the `MenuContainer` interface method to disassociate the `popup` from the component. (`PopupMenu` is a subclass of `MenuComponent`.) If `popup` is not associated with the `Component`, nothing happens.

protected String paramString ()

The `paramString()` method is a protected method that helps build a `String` listing the different parameters of the `Component`. When the `toString()` method is called for a specific `Component`, `paramString()` is called for the lowest level and works its way up the inheritance hierarchy to build a complete parameter string to display. At the `Component` level, potentially seven (Java1.0) or eight (1.1) items are added. The first five items added are the component's name (if non-null and using Java 1.1), x and y coordinates (as returned by `getLocation()`), along with its width and height (as returned by `getSize()`). If the component is not valid, "invalid" is added next. If the component is not visible, "hidden" is added next. Finally, if the component is not enabled, "disabled" is added.

public String toString ()

The `toString()` method returns a `String` representation of the object's values. At the `Component` level, the class's name is placed before the results of `paramString()`. This method is called automatically by the system if you try to print an object using `System.out.println()`.

public void list ()

The `list()` method prints the contents of the `Component` (as returned by `toString()`) to `System.out`. If `c` is a type of `Component`, the two statements `System.out.println(c)` and `c.list()` are equivalent. This method is more useful at the `Container` level, because it prints all the components within the container.

public void list (PrintWriter out) ★
public void list (PrintStream out)

This version of `list()` prints the contents of the `Component` (as returned by `toString()`) to a different `PrintStream`, `out`.

public void list (PrintWriter out, int indentation) ★
public void list (PrintStream out, int indentation)

These versions of `list()` are called by the other two. They print the component's contents (as returned by `toString()`) with the given indentation. This allows you to prepare nicely formatted lists of a container's contents for debugging; you could use the indentation to reflect how deeply the component is nested within the container.

Component Events

Chapter 4, *Events* covers event handling in detail. This section summarizes what `Component` does for the different event-related methods.

With the Java 1.0 event model, many methods return `true` to indicate that the program has handled the event and `false` to indicate that the event was not handled (or only partially handled); when `false` is returned, the system passes the event up to the parent container. Thus, it is good form to return `true` only when you have fully handled the event, and no further processing is necessary.

With the Java 1.1 event model, you register a listener for a specific event type. When that type of event happens, the listener is notified. Unlike the 1.0 model, you do not need to override any methods of `Component` to handle the event. Controllers

The Java 1.0 event model controllers are `deliverEvent()`, `postEvent()`, and `handleEvent()`. With 1.1, the controller is a method named `dispatchEvent()`.

public void deliverEvent (Event e) ☆

> The `deliverEvent()` method delivers the 1.0 `Event e` to the `Component` in which an event occurred. Internally, this method calls `postEvent()`. The `deliverEvent()` method is an important enhancement to `postEvent()` for `Container` objects since they have to determine which component in the `Container` gets the event.

public boolean postEvent (Event e) ☆

> The `postEvent()` method tells the `Component` to deal with 1.0 `Event e`. It calls `handleEvent()`, which returns `true` if some other object handled `e` and `false` if no one handles it. If `handleEvent()` returns `false`, `postEvent()` posts the `Event` to the component's parent. You can use `postEvent()` to hand any events you generate yourself to some other component for processing. (Creating your own events is a useful technique that few developers take advantage of.) You can also use `postEvent()` to reflect an event from one component into another.

public boolean handleEvent (Event e) ☆

> The `handleEvent()` method determines the type of event `e` and passes it along to an appropriate method to deal with it. For example, when a mouse motion event is delivered to `postEvent()`, it is passed off to `handleEvent()`, which calls `mouseMove()`. As shown in the following listing, `handleEvent()` can be implemented as one big switch statement. Since not all event types have default event handlers, you may need to override this method. If you do, remember to call the overridden method to ensure that the default behavior still takes place. To do so, call `super.handleEvent(event)` for any event your method does not deal with.

```
public boolean handleEvent(Event event) {
    switch (event.id) {
      case Event.MOUSE_ENTER:
        return mouseEnter (event, event.x, event.y);
      case Event.MOUSE_EXIT:
        return mouseExit (event, event.x, event.y);
      case Event.MOUSE_MOVE:
        return mouseMove (event, event.x, event.y);
      case Event.MOUSE_DOWN:
        return mouseDown (event, event.x, event.y);
      case Event.MOUSE_DRAG:
        return mouseDrag (event, event.x, event.y);
      case Event.MOUSE_UP:
        return mouseUp (event, event.x, event.y);
      case Event.KEY_PRESS:
      case Event.KEY_ACTION:
        return keyDown (event, event.key);
      case Event.KEY_RELEASE:
      case Event.KEY_ACTION_RELEASE:
        return keyUp (event, event.key);
      case Event.ACTION_EVENT:
        return action (event, event.arg);
      case Event.GOT_FOCUS:
        return gotFocus (event, event.arg);
      case Event.LOST_FOCUS:
        return lostFocus (event, event.arg);
    }
    return false;
}
```

public final void dispatchEvent(AWTEvent e) ★

The `dispatchEvent()` method allows you to post new AWT events to this component's listeners. `dispatchEvent()` tells the `Component` to deal with the `AWTEvent e` by calling its `processEvent()` method. This method is similar to Java 1.0's `postEvent()` method. Events delivered in this way bypass the system's event queue. It's not clear why you would want to bypass the event queue, except possibly to deliver some kind of high priority event.

Action

public boolean action (Event e, Object o) ☆

The `action()` method is called when the user performs some action in the `Component`. `e` is the 1.0

`Event` instance for the specific event, while the content of `o` varies depending upon the specific `Component`. The particular action that triggers a call to `action()` depends on the `Component`. For example, with a `TextField`, `action()` is called when the user presses the carriage return. This method should not be called directly; to deliver any event you generate, call `postEvent()`, and let it decide how the event should propagate.

The default implementation of the `action()` method does nothing and returns `false`. When you override this method, return `true` only if you fully handle the event. Your method should always have a default case that returns `false` or calls `super.action(e, o)` to ensure that the event propagates to the component's container or component's superclass, respectively.

Keyboard

public boolean keyDown (Event e, int key) ★

The `keyDown()` method is called whenever the user presses a key. `e` is the 1.0 `Event` instance for the specific event, while `key` is the integer representation of the character pressed. The identifier for the event (`e.id`) could be either `Event.KEY_PRESS` for a regular key or `Event.KEY_ACTION` for an action-oriented key (e.g., arrow or function key). The default `keyDown()` method does nothing and returns `false`. If you are doing input validation, return `true` if the character is invalid; this keeps the event from propagating to a higher component. If you wish to alter the input (i.e., convert to uppercase), return `false`, but change `e.key` to the new character.

public boolean keyUp (Event e, int key)

The `keyUp()` method is called whenever the user releases a key. `e` is the `Event` instance for the specific event, while `key` is the integer representation of the character pressed. The identifier for the event (`e.id`) could be either `Event.KEY_RELEASE` for a regular key or `Event.KEY_ACTION_RELEASE` for an action-oriented key (e.g., arrow or function key). `keyUp()` may be used to determine how long `key` has been pressed. The default `keyUp()` method does nothing and returns `false`.

Mouse

NOTE:

Early releases of Java (1.0.2 and earlier) propagated only mouse events from `Canvas` and `Container` objects. However, Netscape Navigator seems to have jumped the gun and corrected the situation with their 3.0 release, which is based on Java release 1.0.2.1. Until other Java releases catch up, use these events with care. For more information on platform dependencies, see Appendix C, *Platform-Specific Event Handling*.

public boolean mouseDown (Event e, int x, int y) ☆

The `mouseDown()` method is called when the user presses a mouse button over the `Component`. `e` is the `Event` instance for the specific event, while `x` and `y` are the coordinates where the cursor was located when the event was initiated. It is necessary to examine the modifiers field of `e` to determine which mouse button the user pressed. The default `mouseDown()` method does nothing and returns `false`. When you override this method, return `true` only if you fully handle the event. Your method should always have a default case that returns `false` or calls `super.mouseDown(e, x, y)` to ensure that the event propagates to the component's container or component's superclass, respectively.

public boolean mouseDrag (Event e, int x, int y) ☆

The `mouseDrag()` method is called when the user is pressing a mouse button and moves the mouse. `e` is the `Event` instance for the specific event, while `x` and `y` are the coordinates where the cursor was located when the event was initiated. `mouseDrag()` could be called multiple times as the mouse is moved. The default `mouseDrag()` method does nothing and returns `false`. When you override this method, return `true` only if you fully handle the event. Your method should always have a default case that returns `false` or calls `super.mouseDrag(e, x, y)` to ensure that the event propagates to the component's container or component's superclass, respectively.

public boolean mouseEnter (Event e, int x, int y) ☆

The `mouseEnter()` method is called when the mouse enters the `Component`. `e` is the `Event` instance for the specific event, while `x` and `y` are the coordinates where the cursor was located when the event was initiated. The default `mouseEnter()` method does nothing and returns `false`. `mouseEnter()` can be used for implementing balloon help. When you override this method, return `true` only if you fully handle the event. Your method should always have a default case that returns `false` or calls `super.mouseEnter(e, x, y)` to ensure that the event propagates to the component's container or component's superclass, respectively.

public boolean mouseExit (Event e, int x, int y) ☆

The `mouseExit()` method is called when the mouse exits the `Component`. `e` is the `Event` instance for the specific event, while `x` and `y` are the coordinates where the cursor was located when the event was initiated. The default method `mouseExit()` does nothing and returns `false`. When you override this method, return `true` only if you fully handle the event. Your method should always have a default case that returns `false` or calls `super.mouseExit(e, x, y)` to ensure that the event propagates to the component's container or component's superclass, respectively.

public boolean mouseMove (Event e, int x, int y) ☆

The `mouseMove()` method is called when the user moves the mouse without pressing a mouse button. `e` is the `Event` instance for the specific event, while `x` and `y` are the coordinates where the cursor was located when the event was initiated. `mouseMove()` will be called numerous times as the mouse is moved. The default `mouseMove()` method does nothing and returns `false`. When you override this method, return `true` only if you fully handle the event. Your method should always have a default case that returns `false` or calls `super.mouseMove(e, x, y)` to ensure that the event propagates to the component's container or component's superclass, respectively.

public boolean mouseUp (Event e, int x, int y) ☆

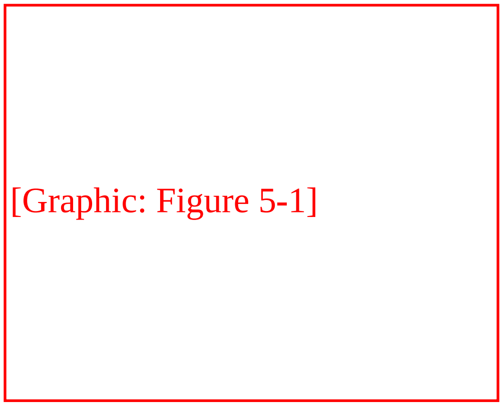
The `mouseUp()` method is called when the user releases a mouse button over the `Component`. `e` is the `Event` instance for the specific event, while `x` and `y` are the coordinates where the cursor was located when the event was initiated. The default `mouseUp()` method does nothing and returns `false`. When you override this method, return `true` only if you fully handle the event. Your method should always have a default case that returns `false` or calls `super.mouseUp(e, x, y)` to ensure that the event propagates to the component's container or component's superclass, respectively.

Focus

Focus events indicate whether a component can get keyboard input. Not all components can get focus (e.g., `Label` cannot). Precisely which components can get the focus is platform specific.

Ordinarily, the item with the focus has a light gray rectangle around it, though the actual display depends on the platform and the component. Figure 5.1 displays the effect of focus for buttons in Windows 95.

Figure 5.1: Focused and UnFocused buttons



NOTE:

Early releases of Java (1.0.2 and earlier) do not propagate all focus events on all platforms. Java 1.1 seems to propagate them properly. For more information on platform dependencies, see Appendix C, *Platform-Specific Event Handling*.

public boolean gotFocus (Event e, Object o) ☆

The `gotFocus()` method is triggered when the `Component` gets the input focus. `e` is the 1.0 `Event` instance for the specific event, while the content of `o` varies depending upon the specific

`Component`. The default `gotFocus()` method does nothing and returns `false`. For a `TextField`, when the cursor becomes active, it has the focus. When you override this method, return `true` to indicate that you have handled the event completely or `false` if you want the event to propagate to the component's container.

public boolean lostFocus (Event e, Object o) ☆

The `lostFocus()` method is triggered when the input focus leaves the `Component`. `e` is the `Event` instance for the specific event, while the content of `o` varies depending upon the specific `Component`. The default `lostFocus()` method does nothing and returns `false`. When you override this method, return `true` to indicate that you have handled the event completely or `false` if you want the event to propagate to the component's container.

Listeners and 1.1 Event Handling

With the 1.1 event model, you receive events by registering event listeners, which are told when the event happens. Components don't have to receive and handle their own events; you can cleanly separate the event-handling code from the user interface itself. This section covers the methods used to add and remove event listeners, which are part of the `Component` class. There is a pair of methods to add and remove listeners for each event type that is appropriate for a `Component`: `ComponentEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent`, and `MouseMotionEvent`. Subclasses of `Component` may have additional event types and therefore will have additional methods for adding and removing listeners. For example, `Button`, `List`, `MenuItem`, and `TextField` each generate action events and therefore have methods to add and remove action listeners. These additional listeners are covered with their respective components.

public void addComponentListener(ComponentListener listener) ★

The `addComponentListener()` method registers `listener` as an object interested in being notified when a `ComponentEvent` passes through the `EventQueue` with this `Component` as its target. When such an event occurs, a method in the `ComponentListener` interface is called. Multiple listeners can be registered.

public void removeComponentListener(ComponentListener listener) ★

The `removeComponentListener()` method removes `listener` as a interested listener. If `listener` is not registered, nothing happens.

public void addFocusListener(FocusListener listener) ★

The `addFocusListener()` method registers `listener` as an object interested in being notified when a `FocusEvent` passes through the `EventQueue` with this `Component` as its target. When such an event occurs, a method in the `FocusListener` interface is called. Multiple listeners can be registered.

public void removeFocusListener(FocusListener listener) ★

The `removeFocusListener()` method removes `listener` as a interested listener. If `listener` is not registered, nothing happens.

public void addKeyListener(KeyListener listener) ★

The `addKeyListener()` method registers `listener` as an object interested in being notified when a `KeyEvent` passes through the `EventQueue` with this `Component` as its target. When such an event occurs, a method in the `KeyListener` interface is called. Multiple listeners can be registered.

public void removeKeyListener(KeyListener listener) ★

The `removeKeyListener()` method removes `listener` as a interested listener. If listener is not registered, nothing happens.

public void addMouseListener(MouseListener listener) ★

The `addMouseListener()` method registers `listener` as an object interested in being notified when a nonmotion-oriented `MouseEvent` passes through the `EventQueue` with this `Component` as its target. When such an event occurs, a method in the `MouseListener` interface is called. Multiple listeners can be registered.

public void removeMouseListener(MouseListener listener) ★

The `removeMouseListener()` method removes `listener` as a interested listener. If listener is not registered, nothing happens.

public void addMouseMotionListener(MouseMotionListener listener) ★

The `addMouseMotionListener()` method registers `listener` as an object interested in being notified when a motion-oriented `MouseEvent` passes through the `EventQueue` with this `Component` as its target. When such an event occurs, a method in the `MouseMotionListener` interface is called. Multiple listeners can be registered.

The mouse motion-oriented events are separate from the other mouse events because of their frequency of generation. If they do not have to propagate around, resources can be saved.

public void removeMouseMotionListener(MouseMotionListener listener) ★

> The `removeMouseMotionListener()` method removes `listener` as a interested listener. If `listener` is not registered, nothing happens.

Handling your own events

Under the 1.1 event model, it is still possible for components to receive their own events, simulating the old event mechanism. If you want to write components that process their own events but are also compatible with the new model, you can override `processEvent()` or one of its related methods. `processEvent()` is logically similar to `handleEvent()` in the old model; it receives all the component's events and sees that they are forwarded to the appropriate listeners. Therefore, by overriding `processEvent()`, you get access to every event the component generates. If you want only a specific type of event, you can override `processComponentEvent()`, `processKeyEvent()`, or one of the other event-specific methods.

However, there is one problem. In Java 1.1, events aren't normally generated if there are no listeners. Therefore, if you want to receive your own events without registering a listener, you should first enable event processing (by a call to `enableEvent()`) to make sure that the events you are interested in are generated.

protected final void enableEvents(long eventsToEnable) ★

> The `enableEvents()` method allows you to configure a component to listen for events without having any active listeners. Under normal circumstances (i.e., if you are not subclassing a component), it is not necessary to call this method.
>
> The `eventsToEnable` parameter contains a mask specifying which event types you want to enable. The `AWTEvent` class (covered in [Chapter 4, *Events*]) contains constants for the following types of events:
>
> *COMPONENT_EVENT_MASK*
> *CONTAINER_EVENT_MASK*
> *FOCUS_EVENT_MASK*
> *KEY_EVENT_MASK*
> *MOUSE_EVENT_MASK*
> *MOUSE_MOTION_EVENT_MASK*
> *WINDOW_EVENT_MASK*
> *ACTION_EVENT_MASK*
> *ADJUSTMENT_EVENT_MASK*
> *ITEM_EVENT_MASK*
> *TEXT_EVENT_MASK*
>
> OR the masks for the events you want; for example, call `enableEvents(MOUSE_EVENT_MASK | MOUSE_MOTION_EVENT_MASK)` to enable all mouse events. Any previous event mask settings are retained.

protected final void disableEvents(long eventsToDisable) ★

The `disableEvents()` method allows you to stop the delivery of events when they are no longer needed. `eventsToDisable` is similar to the `eventsToEnable` parameter but instead contains a mask specifying which event types to stop. A disabled event would still be delivered if someone were listening.

protected void processEvent(AWTEvent e) ★

The `processEvent()` method receives all `AWTEvent` with this `Component` as its target. `processEvent()` then passes them along to one of the event-specific processing methods (e.g., `processKeyEvent()`). When you subclass `Component`, overriding `processEvent()` allows you to process all events without providing listeners. Remember to call `super.processEvent(e)` last to ensure that normal event processing still occurs; if you don't, events won't get distributed to any registered listeners. Overriding `processEvent()` is like overriding the `handleEvent()` method using the 1.0 event model.

protected void processComponentEvent(ComponentEvent e) ★

The `processComponentEvent()` method receives `ComponentEvent` with this `Component` as its target. If any listeners are registered, they are then notified. When you subclass `Component`, overriding `processComponentEvent()` allows you to process component events without providing listeners. Remember to call `super.processComponentEvent(e)` last to ensure that normal event processing still occurs; if you don't, events won't get distributed to any registered listeners. Overriding `processComponentEvent()` is roughly similar to overriding `resize()`, `move()`, `show()`, and `hide()` to add additional functionality when those methods are called.

protected void processFocusEvent(FocusEvent e) ★

The `processFocusEvent()` method receives `FocusEvent` with this `Component` as its target. If any listeners are registered, they are then notified. When you subclass `Component`, overriding `processFocusEvent()` allows you to process the focus event without providing listeners. Remember to call `super.processFocusEvent(e)` last to ensure that normal event processing still occurs; if you don't, events won't get distributed to any registered listeners. Overriding `processFocusEvent()` is like overriding the methods `gotFocus()` and `lostFocus()` using the 1.0 event model.

protected void processKeyEvent(KeyEvent e) ★

The `processKeyEvent()` method receives `KeyEvent` with this `Component` as its target. If any listeners are registered, they are then notified. When you subclass `Component`, overriding `processKeyEvent()` allows you to process key events without providing listeners. Be sure to remember to call `super.processKeyEvent(e)` last to ensure that normal event processing still occurs; if you don't, events won't get distributed to any registered listeners. Overriding `processKeyEvent()` is roughly similar to overriding `keyDown()` and `keyUp()` with one method using the 1.0 event model.

protected void processMouseEvent(MouseEvent e) ★

This `processMouseEvent()` method receives all nonmotion-oriented `MouseEvents` with this `Component` as its target. If any listeners are registered, they are then notified. When you subclass `Component`, overriding the method `processMouseEvent()` allows you to process mouse events without providing listeners. Remember to call `super.processMouseEvent(e)` last to ensure that normal event processing still occurs; if you don't, events won't get distributed to any registered listeners. Overriding the method `processMouseEvent()` is roughly similar to overriding `mouseDown()`, `mouseUp()`, `mouseEnter()`, and `mouseExit()` with one method using the 1.0 event model.

protected void processMouseMotionEvent(MouseEvent e) ★

The `processMouseMotionEvent()` method receives all motion-oriented `MouseEvents` with this `Component` as its target. If there are any listeners registered, they are then notified. When you subclass `Component`, overriding `processMouseMotionEvent()` allows you to process mouse motion events without providing listeners. Remember to call `super.processMouseMotionEvent(e)` last to ensure that normal event processing still occurs; if you don't, events won't get distributed to any registered listeners. Overriding the method `processMouseMotionEvent()` is roughly similar to overriding `mouseMove()` and `mouseDrag()` with one method using the 1.0 event model.

PREVIOUS	HOME	NEXT
The Java 1.1 Event Model	BOOK INDEX	Labels

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

6. Containers

Contents:

This chapter covers a special type of `Component` called `Container`. A `Container` is a subclass of `Component` that can contain other components, including other containers. `Container` allows you to create groupings of objects on the screen. This chapter covers the methods in the `Container` class and its subclasses: `Panel`, `Window`, `Frame`, `Dialog`, and `FileDialog`. It also covers the `Insets` class, which provides an internal border area for the `Container` classes.

Every container has a layout associated with it that controls how the container organizes the components in it. The layouts are described in Chapter 7, *Layouts*.

Java 1.1 introduces a special `Container` called `ScrollPane`. Because of the similarities between scrolling and `ScrollPane`, the new `ScrollPane` container is covered with the `Scrollbar` class in Chapter 11, *Scrolling*.

6.1 Container

`Container` is an abstract class that serves as a general purpose holder of other `Component` objects. The `Container` class holds the methods for grouping the components together, laying out the components inside it, and dealing with events occurring within it. Because `Container` is an abstract class, you never see a pure `Container` object; you only see subclasses that add specific behaviors to a generic container.

Container Methods

Constructors

The abstract `Container` class contains a single constructor to be called by its children. Prior to Java 1.1, the constructor was package private.

protected Container() ★

> The constructor for `Container` creates a new component without a native peer. Since you no longer have a native peer, you must rely on your container to provide a display area. This allows you to create containers that require fewer system resources. For example, if you are creating panels purely for layout management, you might consider creating a `LightweightPanel` class to let you assign a layout manager to a component group. Using `LightweightPanel` will speed things up since events do not have to propagate through the panel and you do not have to get a peer from the native environment. The following code creates the `LightweightPanel` class:

```
import java.awt.*;
public class LightweightPanel extends Container {
    LightweightPanel () {}
    LightweightPanel (LayoutManager lm) {
        setLayout(lm);
    }
}
```

Grouping

A `Container` holds a set of objects within itself. This set of methods describes how to examine and add components to the set.

public int getComponentCount () ★
public int countComponents () ☆

> The `getComponentCount()` method returns the number of components within the container at this level. `getComponentCount()` does not count components in any child `Container` (i.e., containers within the current container).
>
> `countComponents()` is the Java 1.0 name for this method.

public Component getComponent (int position)

The `getComponent()` method returns the component at the specific `position` within it. If `position` is invalid, this method throws the run-time exception `ArrayIndexOutOfBoundsException`.

public Component[] getComponents ()

`getComponents()` returns an array of all the components held within the container. Since these are references to the actual objects on the screen, any changes made to the components returned will be reflected on the display.

public Component add (Component component, int position)

The `add()` method adds `component` to the container at `position`. If `position` is -1, `add()` inserts `component` as the last object within the container. What the container does with `position` depends upon the `LayoutManager` of the container. If `position` is invalid, the `add()` method throws the run-time exception `IllegalArgumentException`. If you try to add `component`'s container to itself (anywhere in the containment tree), this method throws an `IllegalArgumentException`. In Java 1.1, if you try to add a `Window` to a container, `add()` throws the run-time exception `IllegalArgumentException`. If you try to add `component` to a container that already contains it, the container is removed and re-added, probably at a different position.

Assuming that nothing goes wrong, the parent of `component` is set to the container, and the container is invalidated. `add()` returns the `component` just added.

Calling this method generates a `ContainerEvent` with the id `COMPONENT_ADDED`.

public Component add (Component component)

The `add()` method adds `component` to the container as the last object within the container. This is done by calling the earlier version of `add()` with a `position` of -1. If you try to add `component`'s container to itself (anywhere in the containment tree), this method throws the run-time exception `IllegalArgumentException`. In Java 1.1, if you try to add a `Window` to a container, `add()` throws the run-time exception `IllegalArgumentException`.

Calling this method generates a `ContainerEvent` with the id `COMPONENT_ADDED`.

public void add (Component component, Object constraints) ★
public Component add (String name, Component component)

This next version of `add()` is necessary for layouts that require additional information in order to place components. The additional information is provided by the `constraints` parameter. This version of the `add()` method calls the `addLayoutComponent()` method of the `LayoutManager`. What the container does with `constraints` depends upon the actual `LayoutManager`. It can be used for naming containers within a `CardLayout`, specifying a screen area for `BorderLayout`, or providing a set of `GridBagConstraints` for a `GridBagLayout`. In the event that this `add()` is called and the current `LayoutManager` does not take advantage of `constraints`, `component` is added at the end with a position of -1. If you try to add `component`'s container to itself (anywhere in the containment tree), this method throws the run-time exception `IllegalArgumentException`. In Java 1.1, if you try to add a `Window` to a container, `add()` throws the run-time exception `IllegalArgumentException`.

The `add(String`, `Component)` method was changed to `add(component, object)` in Java 1.1 to accommodate the `LayoutManager2` interface (discussed in Chapter 7, *Layouts*) and to provide greater flexibility. In all cases, you can just flip the parameters to bring the code up to 1.1 specs. The string used as an identifier in Java 1.0 is just treated as a particular kind of constraint.

Calling this method generates a `ContainerEvent` with the id `COMPONENT_ADDED`.

public void add (Component component, Object constraints, int index) ★

This final version of `add()` is necessary for layouts that require an `index` and need additional information to place components. The additional information is provided by the `constraints` parameter. This version of `add()` also calls the `addLayoutComponent()` method of the `LayoutManager`. `component` is added with a position of `index`. If you try to add `component`'s container to itself (anywhere in the containment tree), this method throws the run-time exception `IllegalArgumentException`. In Java 1.1, if you try to add a `Window` to a `Container`, `add()` throws the run-time exception `IllegalArgumentException`.

Some layout managers ignore any index. For example, if you call `add(aButton, BorderLayout.NORTH, 3)` to add a `Button` to a `BorderLayout` panel, the `Button` appears in the north region of the layout, no matter what the index.

Calling this method generates a `ContainerEvent` with the id `COMPONENT_ADDED`.

protected void addImpl(Component comp, Object constraints, int index) ★

The protected `addImpl()` method is the helper method that all the others call. It deals with synchronization and enforces all the restrictions on adding components to containers.

The `addImpl()` method tracks the container's components in an internal list. The index with which each component is added determines its position in the list. The lower the component's index, the higher it appears in the stacking order. In turn, the stacking order determines how components are displayed when sufficient space isn't available to display all of them. Components that are added without indices are placed at the end of the list (i.e., at the end of the stacking order) and therefore displayed behind other components. If all components are added without indices, the first component added to the container is first in the stacking order and therefore displayed in front.

You could override `addImpl()` to track when components are added to a container. However, the proper way to find out when components are added is to register a `ContainerListener` and watch for the `COMPONENT_ADDED` and the `COMPONENT_REMOVED` events.

public void remove (int index) ★

The `remove()` method deletes the `component` at position `index` from the container. If `index` is invalid, the `remove()` method throws the run-time exception `IllegalArgumentException`. This method calls the `removeLayoutComponent()` method of the container's `LayoutManager`.

`removeAll()` generates a `ContainerEvent` with the id `COMPONENT_REMOVED`.

public void remove (Component component)

The `remove()` method deletes `component` from the container, if the container directly contains `component`. `remove()` does not look through nested containers trying to find `component`. This method calls the `removeLayoutComponent()` method of the container's `LayoutManager`.

When you call this method, it generates a `ContainerEvent` with the id `COMPONENT_REMOVED`.

public void removeAll ()

The `removeAll()` method removes all components from the container. This is done by looping through all the components, setting each component's parent to `null`, setting the container's reference to the component to `null`, and invalidating the container.

When you call this method, it generates a `ContainerEvent` with the id `COMPONENT_REMOVED` for each component removed.

public boolean isAncestorOf(Component component) ★

The `isAncestorOf()` method checks to see if `component` is a parent (or grandparent or great grandparent) of this container. It could be used as a helper method for `addImpl()` but is not. If `component` is an ancestor of the container, `isAncestorOf()` returns `true`; otherwise, it returns `false`.

Layout and sizing

Every container has a `LayoutManager`. The `LayoutManager` is responsible for positioning the components inside the container. The `Container` methods listed here are used in sizing the objects within the container and specifying a layout.

public LayoutManager getLayout ()

The `getLayout()` method returns the container's current `LayoutManager`.

public void setLayout (LayoutManager manager)

The `setLayout()` method changes the container's `LayoutManager` to `manager` and invalidates the container. This causes the components contained inside to be repositioned based upon `manager`'s rules. If `manager` is `null`, there is no layout manager, and you are responsible for controlling the size and position of all the components within the container yourself.

public Dimension getPreferredSize () ★
public Dimension preferredSize () ☆

The `getPreferredSize()` method returns the `Dimension` (`width` and `height`) for the preferred size of the components within the container. The container determines its preferred size by calling the `preferredLayoutSize()` method of the current `LayoutManager`, which says how much space the layout manager needs to arrange the components. If you override this method, you are overriding the default preferred size.

`preferredSize()` is the Java 1.0 name for this method.

public Dimension getMinimumSize () ★
public Dimension minimumSize () ☆

The `getMinimumSize()` method returns the minimum `Dimension` (`width` and `height`) for the size of the components within the container. This container determines its minimum size by calling the `minimumLayoutSize()` method of the current `LayoutManager`, which computes the minimum amount of space the layout manager needs to arrange the components. It is possible for `getMinimumSize()` and `getPreferredSize()` to return the same dimensions. There is no guarantee that you will get this amount of space for the layout.

`minimumSize()` is the Java 1.0 name for this method.

public Dimension getMaximumSize () ★

The `getMaximumSize()` method returns the maximum `Dimension` (`width` and `height`) for the size of the components within the container. This container determines its maximum size by calling the `maximumLayoutSize()` method of the current `LayoutManager2`, which computes the maximum amount of space the layout manager needs to arrange the components. If the layout manager is not an instance of `LayoutManager2`, this method calls the `getMaximumSize()` method of the `Component`, which returns `Integer.MAX_VALUE` for both dimensions. None of the `java.awt` layout managers use the concept of maximum size yet.

public float getAlignmentX () ★

The `getAlignmentX()` method returns the alignment of the components within the container along the x axis. This container determines its alignment by calling the current `LayoutManager2`'s `getLayoutAlignmentX()` method, which computes it based upon its children. The return value is between 0.0 and 1.0. Values nearer 0 indicate that the component should be placed closer to the left edge of the area available. Values nearer 1 indicate that the component should be placed closer to the right. The value 0.5 means the component should be centered. If the layout manager is not an instance of `LayoutManager2`, this method calls `Component`'s `getAlignmentX()` method, which returns the constant `Component.CENTER_ALIGNMENT`. None of the `java.awt` layout managers use the concept of alignment yet.

public float getAlignmentY () ★

The `getAlignmentY()` method returns the alignment of the components within the container along the y axis. This container determines its alignment by calling the current `LayoutManager2`'s `getLayoutAlignmentY()` method, which computes it based upon its children. The return value is between 0.0 and 1.0. Values nearer 0 indicate that the component should be placed closer to the top of the area available. Values nearer 1 indicate that the component should be placed closer to the bottom. The value 0.5 means the component should be centered. If the layout manager is not an instance of `LayoutManager2`, this method calls `Component`'s `getAlignmentY()` method, which returns the constant `Component.CENTER_ALIGNMENT`. None of the `java.awt` layout managers use the concept of alignment yet.

public void doLayout () ★
public void layout () ☆

The `doLayout()` method of `Container` instructs the `LayoutManager` to lay out the container. This is done by calling the `layoutContainer()` method of the current `LayoutManager`.

`layout()`is the Java 1.0 name for this method.

public void validate ()

The `validate()` method sets the container's valid state to `true` and recursively validates all of its children. If a child is a `Container`, its children are in turn validated. Some components are not completely initialized until they are validated. For example, you cannot ask a `Button` for its display dimensions or position until it is validated.

protected void validateTree () ★

The `validateTree()` method is a helper for `validate()` that does all the work.

public void invalidate () ★

The `invalidate()` method invalidates the container and recursively invalidates the children. If the layout manager is an instance of `LayoutManager2`, its `invalidateLayout()` method is called to invalidate any cached values.

Event delivery

The event model for Java is described in Chapter 4, *Events*. These methods help in the handling of the various system events at the container level.

public void deliverEvent (Event e) ☆

The `deliverEvent()` method is called by the system when the Java 1.0 `Event e` happens. `deliverEvent()` tries to locate a component contained in the container that should receive it. If one is found, the x and y coordinates of `e` are translated for the new target, and `Event e` is delivered to this by calling its `deliverEvent()`. If `getComponentAt()` fails to find an appropriate target, the event is just posted to the container with `postEvent()`.

public Component getComponentAt (int x, int y) ★

public Component locate (int x, int y) ☆

> The container's `getComponentAt()` method calls each component's `contains()` method to see if the `x` and `y` coordinates are within it. If they are, that component is returned. If the coordinates are not in any child component of this container, the container is returned. It is possible for `getComponentAt()` to return `null` if the `x` and `y` coordinates are not within the container. The method `getComponentAt()` can return another `Container` or a lightweight component.
>
> `locate()`is the Java 1.0 name for this method.

public Component getComponentAt (Point p) ★

> This `getComponentAt()` method is identical to the previous method, with the exception that the location is passed as a single point, rather than as separate `x` and `y` coordinates.

Listeners and 1.1 event handling

With the 1.1 event model, you register listeners, which are told when events occur. Container events occur when a component is added or removed.

public synchronized void addContainerListener(ContainerListener listener) ★

> The `addContainerListener()` method registers `listener` as an object interested in receiving notifications when an `ContainerEvent` passes through the `EventQueue` with this `Container` as its target. The `listener.componentAdded()` or `listener.componentRemoved()` method is called when these events occur. Multiple listeners can be registered. The following code demonstrates how to use a `ContainerListener` to register action listeners for all buttons added to an applet. It is similar to the `ButtonTest11` example in Button Events. The trick that makes this code work is the call to `enableEvents()` in `init()`. This method makes sure that container events are delivered in the absence of listeners. In this applet, we know there won't be any container listeners, so we must enable container events explicitly before adding any components.

```
// Java 1.1 only
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
public class NewButtonTest11 extends Applet implements ActionListener {
    Button b;
    public void init () {
        enableEvents (AWTEvent.CONTAINER_EVENT_MASK);
        add (b = new Button ("One"));
        add (b = new Button ("Two"));
        add (b = new Button ("Three"));
        add (b = new Button ("Four"));
    }
    protected void processContainerEvent (ContainerEvent e) {
        if (e.getID() == ContainerEvent.COMPONENT_ADDED) {
            if (e.getChild() instanceof Button) {
                Button b = (Button)e.getChild();
                b.addActionListener (this);
            }
        }
    }
    public void actionPerformed (ActionEvent e) {
        System.out.println ("Selected: " + e.getActionCommand());
    }
}
```

public void removeContainerListener(ContainerListener listener) ★

> The `removeContainerListener()` method removes `listener` as an interested listener. If `listener` is not registered, nothing happens.

protected void processEvent(AWTEvent e) ★

> The `processEvent()` method receives all `AWTEvent`s with this `Container` as its target. `processEvent()` then passes them along to any listeners for processing. When you subclass `Container`, overriding `processEvent()` allows you to process all events yourself, before sending them to any listeners. There is no equivalent under the 1.0 event model.
>
> If you override `processEvent()`, remember to call `super.processEvent(e)` last to ensure that regular event processing can occur. If you want to process your own events, it's a good idea to call `enableEvents()` (inherited from `Component`) to ensure that events are delivered even in the absence of registered listeners.

protected void processContainerEvent(ContainerEvent e) ★

> The `processContainerEvent()` method receives all `ContainerEvent`s with this `Container` as its target. `processContainerEvent()` then passes them along to any listeners for processing. When you subclass `Container`, overriding the `processContainerEvent()` method allows you to process all container events yourself,

before sending them to any listeners. There is no equivalent under the 1.0 event model.

If you override the `processContainerEvent()` method, remember to call `super.processContainerEvent(e)` last to ensure that regular event processing can occur. If you want to process your own events, it's a good idea to call `enableEvents()` (inherited from `Component`) to ensure that events are delivered even in the absence of registered listeners.

Painting

The following methods are early vestiges of an approach to painting and printing. They are not responsible for anything that couldn't be done with a call to `paintAll()` or `printAll()`. However, they are available if you wish to call them.

public void paintComponents (Graphics g)

The `paintComponents()` method of `Container` paints the different components it contains. It calls each component's `paintAll()` method with a clipped graphics context `g`, which is eventually passed to `paint()`.

public void printComponents (Graphics g)

The `printComponents()` method of `Container` prints the different components it contains. It calls each component's `printAll()` method with a clipped graphics context `g`, which is passed to `print()`, and eventually works its way to `paint()`.

Since it is the container's responsibility to deal with painting lightweight peers, the `paint()` and `print()` methods are overridden in Java 1.1.

public void paint(Graphics g) ★

The `paint()` method of `Container` paints the different lightweight components it contains.

public void print(Graphics g) ★

The `print()` method of `Container` prints the different lightweight components it contains.

NOTE:

If you override `paint()` or `print()` in your containers (especially applets), call `super.paint(g)` or `super.print(g)`, respectively, to make sure that lightweight components are rendered. This is a good practice even if you don't currently use any lightweight components; you don't want your code to break mysteriously if you add a lightweight component later.

Peers

The container is responsible for creating and destroying all the peers of the components within it.

public void addNotify ()

The `addNotify()` method of `Container` creates the peer of all the components within it. After `addNotify()` is called, the `Container` is invalid. It is useful for top-level containers to call this method explicitly before calling the method `setVisible(true)` to guarantee that the container is laid out before it is displayed.

public void removeNotify ()

The `removeNotify()` method destroys the peer of all the top-level objects contained within it. This in effect destroys the peers of all the components within the container.

Miscellaneous methods

protected String paramString ()

When you call the `toString()` method of a container, the default `toString()` method of `Component` is called. This in turn calls `paramString()` which builds up the string to display. At the `Container` level, `paramString()` appends the layout manager name, like `layout=java.awt.BorderLayout`, to the output.

public Insets getInsets () ★
public Insets insets () ☆

The `getInsets()` method gets the container's current insets. An inset is the amount of space reserved for the container to use between its edge and the area actually available to hold components. For example, in a `Frame`, the inset for the top would be the space required for the title bar and menu bar. Insets exist for top, bottom, right, and left. When you override this method, you are providing an area within the container that is reserved for free space. If the container has insets, they would be the default. If not, the default values are all zeroes.

The following code shows how to override `insets()` to provide values other than the default. The top and bottom have 20 pixels of inset. The left and right have 50. Insets describes the `Insets` class in more detail.

```
public Insets insets () {            // getInsets() for Java 1.1
        return new Insets (20, 50, 20, 50);
}
```

To find out the current value, just call the method and look at the results. For instance, for a `Frame` the results could be the following in the format used by `toString()`:

```
java.awt.Insets[top=42,left=4,right=4,bottom=4]
```

The `42` is the space required for the title and menu bar, while the `4` around the edges are for the window decorations. These results are platform specific and allow you to position items based upon the user's run-time environment.

When drawing directly onto the graphics context of a container with a large inset such as `Frame`, remember to work around the insets. If you do something like `g.drawString("Hello World", 5, 5)` onto a `Frame`, the user won't see the text. It will be under the title bar and menu bar.

`insets()` is the Java 1.0 name for this method.

public void list (PrintWriter output, int indentation) ★
public void list (PrintStream output, int indentation)

The `list()` method is very helpful if you need to find out what is inside a container. It recursively calls itself for each container level of objects inside it, increasing the `indentation` at each level. The results are written to the `PrintStream` or `PrintWriter` `output`.

PREVIOUS Cursor | HOME BOOK INDEX | NEXT Panel

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

7. Layouts

Contents:

This chapter expands upon the idea of a layout manager, which was mentioned briefly in the previous chapter. Every container has a `LayoutManager` that is responsible for positioning the component objects within it, regardless of the platform or the screen size. Layout managers eliminate the need to compute component placement on your own, which would be a losing proposition since the size required for any component depends on the platform on which it is displayed. Even for a simple layout, the code required to discover component sizes and compute absolute positions could be hundreds of lines, particularly if you concern yourself with what happens when the user resizes a window. A layout manager takes care of this for you. It asks each component in the layout how much space it requires, then arranges the components on the screen as best it can, based on the component sizes on the platform in use and the space available, resizing the components as needed.

To find out how much space a component needs, a layout manager calls the component's `getMinimumSize()` and `getPreferredSize()` methods. (Java 1.1 also has a `getMaximumSize()` method; the existing layout managers don't take advantage of it.) These methods report the minimum space that a component requires to be displayed correctly and the optimal size at which it looks best. Thus, each component must know its space requirements; the layout manager uses these to arrange the screen; and your Java program never has to worry about platform-dependent positioning.

The `java.awt` package provides five layout managers: `FlowLayout`, `BorderLayout`, `GridLayout`, `CardLayout`, and `GridBagLayout`. Four additional layouts are provided in the `sun.awt` package: `HorizBagLayout`, `VerticalBagLayout`, `OrientableFlowLayout`, and `VariableGridLayout`. `OrientableFlowLayout` is new to Java 1.1. Of the 1.0 layouts, all are available in the JDK and Internet Explorer. The `VariableGridLayout` is also available with Netscape Navigator. This chapter discusses all of them, along with the `LayoutManager` and `LayoutManager2` interfaces; we'll pay particular attention to how each layout manager computes positions for its components. We will also discuss how to combine layouts to generate more complex screens and how to create your own `LayoutManager` for special situations.

7.1 The LayoutManager Interface

The `LayoutManager` interface defines the responsibilities of something that wants to lay out `Component`s within a `Container`. It is the `LayoutManager`'s duty to determine the position and size of each component within the `Container`. You will never directly call the methods of the `LayoutManager` interface; for the most part, layout managers do their work behind the scenes. Once you have created a `LayoutManager` object and told the container to use it (by calling `setLayout()`), you're finished with it. The system calls the appropriate methods in the layout manager when necessary.

Therefore, the `LayoutManager` interface is most important when you are writing a new layout manager; we'll discuss it here because it's the scaffolding on which all layout managers are based. Like any interface, `LayoutManager` specifies the methods a layout manager must implement but says nothing about how the `LayoutManager` does its job. Therefore, we'll make a few observations before proceeding. First, a layout manager is free to ignore some of its components; there is no requirement that a layout manager display everything. For example, a `Container` using a `BorderLayout` might include thirty or forty components. However, the `BorderLayout` will display at most five of them (the last component placed in each of its five named areas). Likewise, a `CardLayout` may manage many components but displays only one at a time.

Second, a layout manager can do anything it wants with the components' minimum and preferred sizes. It is free to ignore either. It makes sense that a layout manager can ignore a preferred size; after all, "preferred" means "give me this if it's available." However, a layout manager can also ignore a minimum size. At times, there is no reasonable alternative: the container may not have enough room to display a component at its minimum size. How to handle this situation is left to the layout manager's discretion. All layout managers currently ignore a component's maximum size, though this may change in the future.

Methods of the LayoutManager Interface

Five methods make up the `LayoutManager` interface. If you create your own class that implements `LayoutManager`, you must define all five. As you will see, many of the methods do not have to do anything, but there must still be a stub with the appropriate method signature.

public abstract void addLayoutComponent (String name, Component component)

> The `addLayoutComponent()` method is called only when the program assigns a `name` to the `component` when adding it to the layout (i.e., the program calls `add(String, Component)` rather than simply calling `add(Component)` or the Java 1.1 `add(Component, Object)`). It is up to the layout manager to decide what, if anything, to do with the name. For example, `BorderLayout` uses `name` to specify an area on the screen in which to display the component. Most layout managers don't require a name and will only implement a stub.

public abstract void removeLayoutComponent (Component component)

> The `removeLayoutComponent()` method's responsibility is to remove `component` from any internal storage used by the layout manager. This method will probably be stubbed out for your own layouts and do nothing. However, it may need to do something if your layout manager associates components with names.

public abstract Dimension preferredLayoutSize (Container parent)

> The `preferredLayoutSize()` method is called to determine the preferred size of the components within the `Container`. It returns a `Dimension` object that contains the required height and width. `parent` is the object whose components need to be laid out. Usually, the `LayoutManager` determines how to size `parent` by calculating the sizes of the components within it and calculating the dimensions required to display them. On other occasions, it may just return `parent.setSize()`.

public abstract Dimension minimumLayoutSize (Container parent)

> The `minimumLayoutSize()` method is called to determine the minimum size of the components within the `Container`. It returns a `Dimension` object that contains the required height and width. `parent` is the object whose components need to be laid out.

public abstract void layoutContainer (Container parent)

> The `layoutContainer()` method is where a `LayoutManager` does most of its work. The `layoutContainer()` method is responsible for the positioning of all the `Components` of `parent`. Each specific layout positions the enclosed components based upon its own rules.

The LayoutManager2 Interface

Numerous changes were introduced in Java 1.1 to make it conform to various design patterns. These patterns provide consistency in usage and make Java programming easier. The `LayoutManager2` interface was introduced for this reason. This new interface solves a problem that occurs when working with the `GridBagLayout`. While the `addLayoutComponent(String, Component)` method of `LayoutManager` works great for `BorderLayout` and `CardLayout`, you can't use it for a `GridBagLayout`. The position of a component in a `GridBagLayout` is controlled by a number of constraints, which are encapsulated in a `GridBagConstraints` object. To associate constraints with a component, you needed to call a `setConstraints()` method. Although this works, it is not consistent with the way you add components to other layouts. Furthermore, as more and more people create their own layout managers, the number of ways to associate positioning information with a component could grow endlessly. `LayoutManager2` defines a version of `addLayoutComponent()` that can be used by all constraint-based layout managers, including older managers like `BorderLayout` and `CardLayout`. This method lets you pass an arbitrary object to the layout manager to provide positioning information. Layout managers that need additional information (like the `GridBagConstraints` object) now implement `LayoutManager2` instead of `LayoutManager`.

In addition to swapping the parameters to the `addLayoutComponent(Component, Object)`, the new `LayoutManager2` interface also defines several methods that aren't really needed now but will facilitate the introduction of "peerless components" in a later release. Methods of the LayoutManager2 interface

public abstract void addLayoutComponent(Component comp, Object constraints) ★

The `addLayoutComponent()` method is called when a program assigns `constraints` to the component `comp` when adding it to the layout. In practice, this means that the program added the component by calling the new method `add(Component component, Object constraints)` rather than the older methods `add(Component component)` or `add(String name, Component component)`). It is up to the layout manager to decide what, if anything, to do with the `constraints`. For example, `GridBagLayout` uses `constraints` to associate a `GridBagConstraints` object to the component `comp`. `BorderLayout` uses `constraints` to associate a location string (like "Center") with the component.

public abstract Dimension maximumLayoutSize(Container target) ★

The `maximumLayoutSize()` method must return the maximum size of the `target` container under this layout manager. Previously, only minimum and preferred sizes were available. Now a container can have a maximum size. Once layout managers support the concept of maximum sizes, containers will not grow without bounds when additional space is available. If there is no actual maximum, the `Dimension` should have a width and height of the constant `Integer.MAX_VALUE`.

public abstract float getLayoutAlignmentX(Container target) ★

The `getLayoutAlignmentX()` method must return the alignment of `target` along the x axis. The return value should be between 0.0 and 1.0. Values nearer 0 mean that the container will be positioned closer to the left edge of the area available. Values nearer 1 mean that the container will be positioned closer to the right. The value 0.5 means the container should be centered.

public abstract float getLayoutAlignmentY(Container target) ★

The `getLayoutAlignmentY()` method must return the alignment of `target` along the y axis. The return value should be between 0.0 and 1.0. Values nearer 0 mean that the container will be positioned closer to the top of the area available. Values nearer 1 mean that the container will be positioned closer to the bottom. The value 0.5 means the container should be centered.

public abstract void invalidateLayout(Container target) ★

The `invalidateLayout()` method tells the layout manager that any layout information it has for `target` is invalid. This method will usually be implemented as a stub (i.e., `{}`). However, if the layout manager caches any information about `target` when this method is called, the manager should consider that information invalid and discard it.

PREVIOUS FileDialog | HOME BOOK INDEX | NEXT FlowLayout

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

8. Input Fields

Contents:

There are two fundamental ways for users to provide input to a program: they can type on a keyboard, or they can select something (a button, a menu item, etc.) using a mouse. When you want a user to provide input to your program, you can display a list of choices to choose from or allow the user to interact with your program by typing with the keyboard. Presenting choices to the user is covered in Chapter 9, *Pick Me*. As far as keyboard input goes, the `java.awt` package provides two options. The `TextField` class is a single line input field, while the `TextArea` class is a multiline one. Both `TextField` and `TextArea` are subclasses of the class `TextComponent`, which contains all the common functionality of the two. `TextComponent` is a subclass of `Component`, which is a subclass of `Object`. So you inherit all of these methods when you work with either `TextField` or `TextArea`.

8.1 Text Component

By themselves, the `TextField` and `TextArea` classes are fairly robust. However, in order to reduce duplication between the classes, they both inherit a number of methods from the `TextComponent` class. The constructor for `TextComponent` is package private, so you cannot create an instance of it yourself. Some of the activities shared by `TextField` and `TextArea` through the `TextComponent` methods include setting the text, getting the text, selecting the text, and making it read-only.

TextComponent Methods

Contents

Both `TextField` and `TextArea` contain a set of characters whose content determines the current value of the `TextComponent`. The following methods are usually called in response to an external event.

public String getText ()

> The `getText()` method returns the current contents of the `TextComponent` as a `String` object.

public void setText (String text)

The `setText()` method sets the content of the `TextComponent` to `text`. If the `TextComponent` is a `TextArea`, you can embed newline characters (`\n`) in the `text` so that it will appear on multiple lines.

Text selection

Users can select text in `TextComponents` by pressing a mouse button at a starting point and dragging the cursor across the text. The selected text is displayed in reverse video. Only one block of text can be selected at any given time within a single `TextComponent`. Once selected, this block could be used to provide the user with some text-related operation such as cut and paste (on a `PopupMenu`).

Depending on the platform, you might or might not be able to get selected text when a `TextComponent` does not have the input focus. In general, the component with selected text must have input focus in order for you to retrieve any information about the selection. However, in some environments, the text remains selected when the component no longer has the input focus.

public int getSelectionStart ()

The `getSelectionStart()` method returns the initial position of any selected text. The position can be considered the number of characters preceding the first selected character. If there is no selected text, `getSelectionStart()` returns the current cursor position. If the start of the selection is at beginning of the text, the return value is 0.

public int getSelectionEnd ()

The `getSelectionEnd()` method returns the ending cursor position of any selected text--that is, the number of characters preceding the end of the selection. If there is no selected text, `getSelectionEnd()` returns the current cursor position.

public String getSelectedText ()

The `getSelectedText()` method returns the currently selected text of the `TextComponent` as a `String`. If nothing is selected, `getSelectedText()` returns an empty `String`, not `null`.

public void setSelectionStart (int position) ★

The `setSelectionStart()` method changes the beginning of the current selection to `position`. If `position` is after `getSelectionEnd()`, the cursor position moves to `getSelectionEnd()`, and nothing is selected.

public void setSelectionEnd (int position) ★

The `setSelectionEnd()` method changes the end of the current selection to `position`. If `position` is before `getSelectionStart()`, the cursor position moves to `position`, and nothing is selected.

public void select (int selectionStart, int selectionEnd)

> The `select()` method selects the text in the `TextComponent` from `selectionStart` to `selectionEnd`. If `selectionStart` is after `selectionEnd`, the cursor position moves to `selectionEnd`. Some platforms allow you to use `select()` to ensure that a particular position is visible on the screen.

public void selectAll ()

> The `selectAll()` method selects all the text in the `TextComponent`. It basically does a `select()` call with a `selectionStart` position of 0 and a `selectionEnd` position of the length of the contents.

Carets

Introduced in Java 1.1 is the ability to set and get the current insertion position within the text object.

public int getCaretPosition () ★

> The `getCaretPosition()` method returns the current text insertion position (often called the "cursor") of the `TextComponent`. You can use this position to paste text from the clipboard with the `java.awt.datatransfer` package described in Chapter 16, *Data Transfer*.

public void setCaretPosition (int position) ★

> The `setCaretPosition()` method moves the current text insertion location of the `TextComponent` to `position`. If the `TextComponent` does not have a peer yet, `setCaretPosition()` throws the `IllegalComponentStateException` run-time exception. If `position` < 0, this method throws the run-time exception `IllegalArgumentException`. If `position` is too big, the text insertion point is positioned at the end.
>
> Prior to Java version 1.1, the insertion location was usually set by calling `select(position, position)`.

Read-only text

By default, a `TextComponent` is editable. If a user types while the component has input focus, its contents will change. A `TextComponent` can also be used in an output-only (read-only) mode.

public void setEditable (boolean state)

> The `setEditable()` method allows you to change the current editable state of the `TextComponent` to `state`. `true` means the component is editable; `false` means read-only.

public boolean isEditable ()

> The `isEditable()` method tells you if the `TextComponent` is editable (`true`) or read-only (`false`).

The following listing is an applet that toggles the editable status for a `TextArea` and sets a label to show the current status. As you can see in Figure 8.1, platforms can change the display characteristics of the `TextComponent` to reflect whether the component is editable. (Windows 95 darkens the background. Motif and Windows NT do nothing.)

```
import java.awt.*;
import java.applet.*;
public class readonly extends Applet {
    TextArea area;
    Label label;
    public void init () {
        setLayout (new BorderLayout (10, 10));
        add ("South", new Button ("toggleState"));
        add ("Center", area = new TextArea ("Help Me", 5, 10));
        add ("North", label = new Label ("Editable", Label.CENTER));
    }
    public boolean action (Event e, Object o) {
        if (e.target instanceof Button) {
            if ("toggleState".equals(o)) {
               area.setEditable (!area.isEditable ());
               label.setText ((area.isEditable () ? "Editable" : "Read-only"));
               return true;
            }
        }
        return false;
    }
}
```

Figure 8.1: Editable and read-only TextAreas

[Graphic: Figure 8-1]

Miscellaneous methods

public synchronized void removeNotifiy ()

> The `removeNotify()` method destroys the peer of the `TextComponent` and removes it from the screen. Prior to the `TextComponent` peer's destruction, the current state is saved so that a subsequent call to `addNotify()` will put it back. (`TextArea` and `TextField` each have their own `addNotify()` methods.) These methods deal with the peer object, which hides the native platform's implementation of the component. If you override this method for a specific `TextComponent`, put in the customizations for your new class first, and call `super.removeNotify()` last.

protected String paramString ()

> When you call the `toString()` method of a `TextField` or `TextArea`, the default `toString()` method of `Component` is called. This in turn calls `paramString()`, which builds up the string to display. The `TextComponent` level potentially adds four items. The first is the current contents of the `TextComponent` (`getText()`). If the text is editable, `paramString()` adds the word *editable* to the string. The last two items included are the current selection range (`getSelectionStart()` and `getSelectionEnd()`).

TextComponent Events

With the 1.1 event model, you can register listeners for text events. A text event occurs when the component's content changes, either because the user typed something or because the program called a method like `setText()`. Listeners are registered with the `addTextListener()` method. When the content changes, the `TextListener.textValueChanges()` method is called through the protected method `processTextEvent()`. There is no equivalent to `TextEvent` in Java 1.0; you would have to direct keyboard changes and all programmatic changes to a common method yourself.

In addition to `TextEvent` listeners, Key, mouse, and focus listeners are registered through the `Component` methods `addKeyListener()`, `addMouseListener()`, `addMouseMotionListener()`, and `addFocusListener()`, respectively. Listeners and 1.1 event handling

public synchronized void addTextListener(TextListener listener) ★

> The `addTextListener()` method registers `listener` as an object interested in receiving notifications when a `TextEvent` passes through the `EventQueue` with this `TextComponent` as its target. The `listener.textValueChanged()` method is called when these events occur. Multiple listeners can be registered.
>
> The following applet, `text13`, demonstrates how to use a `TextListener` to handle the events that occur when a `TextField` is changed. Whenever the user types into the `TextField`, a `TextEvent` is delivered to the `textValueChanged()` method, which prints a message on the Java console. The applet includes a button that, when pressed, modifies the text field `tf` by calling `setText()`. These changes also generate a `TextEvent`.

```
// Java 1.1 only
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
class TextFieldSetter implements ActionListener {
    TextField tf;
    TextFieldSetter (TextField tf) {
        this.tf = tf;
    }
    public void actionPerformed(ActionEvent e) {
        if (e.getActionCommand().equals ("Set")) {
            tf.setText ("Hello");
        }
    }
}
public class text13 extends Applet implements TextListener {
    TextField tf;
    int i=0;
    public void init () {
        Button b;
        tf = new TextField ("Help Text", 20);
        add (tf);
        tf.addTextListener (this);
        add (b = new Button ("Set"));
        b.addActionListener (new TextFieldSetter (tf));
    }
    public void textValueChanged(TextEvent e) {
        System.out.println (++i + ": " + e);
    }
}
```

public void removeTextListener(TextListener listener) ★

The `removeTextListener()` method removes `listener` as an interested listener. If `listener` is not registered, nothing happens.

protected void processEvent(AWTEvent e) ★

The `processEvent()` method receives all `AWTEvent`s with this `TextComponent` as its target. `processEvent()` then passes the events along to any listeners for processing. When you subclass `TextComponent`, overriding `processEvent()` allows you to process all events yourself, before sending them to any listeners. In a way, overriding `processEvent()` is like overriding `handleEvent()` using the 1.0 event model.

If you override `processEvent()`, remember to call `super.processEvent(e)` last to ensure that regular event processing can occur. If you want to process your own events, it's a good idea to call `enableEvents()` (inherited from `Component`) to ensure that events are delivered even in the absence of registered listeners.

protected void processTextEvent(TextEvent e) ★

The `processTextEvent()` method receives all `TextEvent`s with this `TextComponent` as its target. `processTextEvent()` then passes them along to any listeners for processing. When you subclass `TextField` or `TextArea`, overriding the `processTextEvent()` method allows you to process all text events yourself, before sending them to any listeners. There is no equivalent to `processTextEvent()` within the 1.0 event model.

If you override `processTextEvent()`, remember to call the method `super.processTextEvent(e)` last to ensure that regular event processing can occur. If you want to process your own events, it's a good idea to call `enableEvents()` (inherited from `Component`) to ensure that events are delivered even in the absence of registered listeners.

PREVIOUS HOME NEXT
Other Layouts Available on the Net BOOK INDEX TextField

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

9. Pick Me

Contents:

Three AWT components let you present a list of choices to users: `Choice`, `List`, and `Checkbox`. All three components implement the `ItemSelectable` interface (Java1.1). These components are comparable to selection mechanisms in modern GUIs so most readers will be able to learn them easily, but I'll point out some special enhancements that they provide.

`Choice` and `List` are similar; both offer a list of choices for the user to select. `Choice` provides a pull-down list that offers one selection at a time, whereas `List` is a scrollable list that allows a user to make one or multiple selections. From a design standpoint, which you choose depends at least partially on screen real estate; if you want the user to select from a large group of alternatives, `Choice` requires the least space, `List` requires somewhat more, while `Checkbox` requires the most. `Choice` is the only component in this group that does not allow multiple selections. A `List` allows multiple or single selection; because each `Checkbox` is a separate component, checkboxes inherently allow multiple selection. In order to create a list of mutually exclusive checkboxes, in which only one box can be selected at a time (commonly known as radio buttons), you can put several checkboxes together into a `CheckboxGroup`, which is discussed at the end of this chapter.

9.1 Choice

The `Choice` component provides pop-up/pull-down lists. It is the equivalent of Motif's OptionMenu or Windows MFC's ComboBox. (Java 1.1 departs from the MFC world.) With the `Choice` component, you can provide a short list of choices to the user, while taking up the space of a single item on the screen. When the component is selected, the complete list of available choices appears on the screen. After the user has selected an option, the list is removed from the screen and the selected item is displayed. Selecting any item automatically deselects the previous selection.

Component Methods

Constructors

public Choice ()

There is only one constructor for `Choice`. When you call it, a new instance of `Choice` is created. The component is initially empty, with no items to select. Once you add some items using `addItem()` (version 1.0) or `add()` (version 1.1) and display the `Choice` on the screen, it will look something like the leftmost component in Figure 9.1. The center component shows what a `Choice` looks like when it is selected, while the one on the right shows what a `Choice` looks like before any items have been added to it.

Figure 9.1: How Choices are displayed

[Graphic: Figure 9-1]

Items

public int getItemCount () ★
public int countItems () ☆

The `getItemCount()` method returns the number of selectable items in the `Choice` object. In Figure 9.1, `getItemCount()` would return 6.

`countItems()` is the Java 1.0 name for this method.

public String getItem (int index)

The `getItem()` method returns the text for the item at position `index` in the `Choice`. If `index` is invalid--either `index` < 0 or `index` >= `getItemCount()`--the `getItem()` method throws the `ArrayIndexOutOfBoundsException` run-time exception.

public synchronized void add (String item) ★
public synchronized void addItem (String item) ☆

`add()` adds `item` to the list of available choices. If `item` is already an option in the `Choice`, this method adds it again. If `item` is `null`, `add()` throws the run-time exception `NullPointerException`. The first `item` added to a `Choice` becomes the initial (default) selection.

`addItem()` is the Java 1.0 name for this method.

public synchronized void insert (String item, int index) ★

`insert()` adds `item` to the list of available choices at position `index`. An index of 0 adds the item at the beginning. An index larger than the number of choices adds the item at the end. If `item` is `null`, `insert()` throws the run-time exception `NullPointerException`. If `index` is negative, `insert()` throws the run-time exception `IllegalArgumentException`.

public synchronized void remove (String item) ★

`remove()` removes `item` from the list of available choices. If `item` is present in `Choice` multiple times, a call to `remove()` removes the first instance. If `item` is `null`, `remove()` throws the run-time exception `NullPointerException`. If `item` is not found in the `Choice`, `remove()` throws the `IllegalArgumentException` run-time exception.

public synchronized void remove (int position) ★

`remove()` removes the item at `position` from the list of available choices. If `position` is invalid--either `position` < 0 or `position` >= `getItemCount()`--`remove()` throws the run-time exception `ArrayIndexOutOfBoundsException`.

public synchronized void removeAll () ★

The `removeAll()` method removes every option from the `Choice`. This allows you to refresh the list from scratch, rather than creating a new `Choice` and repopulating it.

Selection

The `Choice` has one item selected at a time. Initially, it is the first item that was added to the `Choice`.

public String getSelectedItem ()

The `getSelectedItem()` method returns the currently selected item as a `String`. The text returned is the parameter used in the `addItem()` or `add()` call that put the option in the `Choice`. If `Choice` is empty, `getSelectedItem()` returns `null`.

public Object[] getSelectedObjects () ★

The getSelectedObjects() method returns the currently selected item as an Object array, instead of a String. The array will either be a one-element array, or null if there are no items. This method is required by the ItemSelectable interface and allows you to use the same method to look at the items selected by a Choice, List, or Checkbox.

public int getSelectedIndex ()

The getSelectedIndex() method returns the position of the currently selected item. The Choice list uses zero-based indexing, so the position of the first item is zero. The position of the last item is the value of countItems()-1. If the list is empty, this method returns -1.

public synchronized void select (int position)

This version of the select() method makes the item at position the selected item in the Choice. If position is too big, select() throws the run-time exception IllegalArgumentException. If position is negative, nothing happens.

public void select (String string)

This version of select() makes the item with the label string the selected item. If string is in the Choice multiple times, this method selects the first. If string is not in the Choice, nothing happens.

Miscellaneous methods

public synchronized void addNotify ()

The addNotify() method creates the Choice's peer. If you override this method, call super.addNotify() first, then add your customizations for the new class. You will then be able to do everything you need with the information about the newly created peer.

protected String paramString ()

When you call the toString() method of a Choice, the default toString() method of Component gets called. This in turn calls paramString() which builds up the string to display. At the Choice level, paramString() appends the currently selected item (the result of getSelectedItem()) to the output. Using the first Choice instance in Figure 9.1, the results would be:

```
java.awt.Choice[139,5,92x27,current=Dialog]
```

Choice Events

The primary event for a Choice occurs when the user selects an item in the list. With the 1.0 event model, selecting an item generates an ACTION_EVENT, which triggers a call to the action() method. Once the

Choice has the input focus, the user can change the selection by using the arrow or keyboard keys. The arrow keys scroll through the list of choices, triggering the KEY_ACTION, ACTION_EVENT, and KEY_ACTION_RELEASE event sequence, which in turn invokes the keyDown(), action(), and keyUp() methods, respectively. If the mouse is used to choose an item, no mouse events are triggered as you scroll over each item, and an ACTION_EVENT occurs only when a specific choice is selected.

With the 1.1 event model, you register ItemListener with addItemListener(). Then when the user selects the Choice, the ItemListener.itemStateChanged() method is called through the protected Choice.processItemEvent() method. Key, mouse, and focus listeners are registered through the Component methods of addKeyListener(), addMouseListener(), and addFocusListener(), respectively. Action

public boolean action (Event e, Object o)

> The action() method for a choice signifies that the user selected an item. e is the Event instance for the specific event, while o is the String from the call to addItem() or add() that represents the current selection. Here's a trivial implementation of the method:

```
public boolean action (Event e, Object o) {
    if (e.target instanceof Choice) {
        System.out.println ("Choice is now set to " + o);
    }
    return false;
}
```

Keyboard

The keyboard events for a Choice can be generated once the Choice has the input focus. In addition to the KEY_ACTION and KEY_ACTION_RELEASE events you get with the arrow keys, an ACTION_EVENT is generated over each entry.

public boolean keyDown (Event e, int key)

> The keyDown() method is called whenever the user presses a key and the Choice has the input focus. e is the Event instance for the specific event, while key is the integer representation of the character pressed. The identifier for the event (e.id) for keyDown() could be either Event.KEY_PRESS for a regular key or Event.KEY_ACTION for an action-oriented key (i.e., arrow or function key). If you check the current selection in this method through the method getSelectedItem() or getSelectedIndex(), you will be given the previously selected item because the Choice's selection has not changed yet. keyDown() is not called when the Choice is changed by using the mouse.

public boolean keyUp (Event e, int key)

> The keyUp() method is called whenever the user releases a key. e is the Event instance for the specific

event, while `key` is the integer representation of the character pressed. The identifier for the event (`e.id`) for `keyUp()` could be either `KEY_RELEASE` for a regular key or `KEY_ACTION_RELEASE` for an action oriented key (i.e., arrow or function key).

Mouse

Ordinarily, the `Choice` component does not trigger any mouse events. Focus

Ordinarily, the `Choice` component does not trigger any focus events. Listeners and 1.1 event handling

With the 1.1 event model, you register listeners for different event types; the listeners are told when the event happens. These methods register listeners, and let the `Choice` component inspect its own events.

public void addItemListener(ItemListener listener) ★

The `addItemListener()` method registers `listener` as an object interested in being notified when an `ItemEvent` passes through the `EventQueue` with this `Choice` as its target. The `listener.itemStateChanged()` method is called when an event occurs. Multiple listeners can be registered.

public void removeItemListener(ItemListener listener) ★

The `removeItemListener()` method removes `listener` as a interested listener. If `listener` is not registered, nothing happens.

protected void processEvent(AWTEvent e) ★

The `processEvent()` method receives all `AWTEvent`s with this `Choice` as its target. `processEvent()` then passes them along to any listeners for processing. When you subclass `Choice`, overriding `processEvent()` allows you to process all events yourself, before sending them to any listeners. In a way, overriding `processEvent()` is like overriding `handleEvent()` using the 1.0 event model.

If you override `processEvent()`, remember to call `super.processEvent(e)` last to ensure that regular event processing can occur. If you want to process your own events, it's a good idea to call `enableEvents()` (inherited from `Component`) to ensure that events are delivered even in the absence of registered listeners.

protected void processItemEvent(ItemEvent e) ★

The `processItemEvent()` method receives all `ItemEvent`s with this `Choice` as its target. `processItemEvent()` then passes them along to any listeners for processing. When you subclass `Choice`, overriding `processItemEvent()` allows you to process all events yourself, before sending them to any listeners. In a way, overriding `processItemEvent()` is like overriding `handleEvent()` using the 1.0 event model.

If you override `processItemEvent()`, remember to call the method `super.processItemEvent(e)` last to ensure that regular event processing can occur. If you want to process your own events, it's a good idea to call `enableEvents()` (inherited from `Component`) to ensure that events are delivered even in the absence of registered listeners.

The following simple applet below demonstrates how a component can receive its own events by overriding `processItemEvent()`, while still allowing other objects to register as listeners. `MyChoice11` is a subclass of `Choice` that processes its own item events. `choice11` is an applet that uses the `MyChoice11` component and registers itself as a listener for item events.

```
// Java 1.1 only
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
class MyChoice11 extends Choice {
    MyChoice11 () {
        super ();
        enableEvents (AWTEvent.ITEM_EVENT_MASK);
    }
    protected void processItemEvent(ItemEvent e) {
        ItemSelectable ie = e.getItemSelectable();
        System.out.println ("Item Selected: " + ie.getSelectedObjects()[0]);
        // If you do not call super.processItemEvent()
        // no listener will be notified
        super.processItemEvent (e);
    }
}
public class choice11 extends Applet implements ItemListener {
    Choice c;
    public void init () {
        String []fonts;
        fonts = Toolkit.getDefaultToolkit().getFontList();
        c = new MyChoice11();
        for (int i = 0; i < fonts.length; i++) {
            c.add (fonts[i]);
        }
        add (c);
        c.addItemListener (this);
   }
    public void itemStateChanged(ItemEvent e)  {
        ItemSelectable ie = e.getItemSelectable();
        System.out.println ("State Change: " + ie.getSelectedObjects()[0]);
    }
}
```

A few things are worth noticing. `MyChoice11` calls `enableEvents()` in its constructor to make sure that item events are delivered, even if nobody registers as a listener: `MyChoice11` needs to make sure that it receives events, even in the absence of listeners. Its `processItemEvent()` method ends by calling the superclass's `processItemEvent()` method, with the original item event. This call ensures that normal item event processing occurs; `super.processItemEvent()` is responsible for distributing the event to any registered listeners. The alternative would be to implement the whole registration and event distribution mechanism inside `myChoice11`, which is precisely what object-oriented programming is supposed to avoid, or being absolutely sure that you will only use `MyChoice11` in situations in which there won't be any listeners, drastically limiting the usefulness of this class.

`choice11` doesn't contain many surprises. It implements `ItemListener`, the listener interface for item events; provides the required `itemStateChanged()` method, which is called whenever an item event occurs; and calls `MyChoice11`'s method `addItemListener()` to register as a listener for item events. (`MyChoice11` inherits this method from the `Choice` class.)

PREVIOUS	HOME	NEXT
Extending TextField	BOOK INDEX	Lists

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

10. Would You Like to Choose from the Menu?

Contents:

In Chapter 6, *Containers*, I mentioned that a `Frame` can have a menu. Indeed, to offer a menu in the AWT, you have to attach it to a `Frame`. With versions 1.0.2 and 1.1, Java does not support menu bars within an applet or any other container. We hope that future versions of Java will allow menus to be used with other containers. Java 1.1 goes partway toward solving this problem by introducing a `PopupMenu` that lets you attach context menus to any `Component`. Java 1.1 also adds `MenuShortcut` events, which represent keyboard accelerator events for menus.

Implementing a menu in a `Frame` involves connections among a number of different objects: `MenuBar`, `Menu`, `MenuItem`, and the optional `CheckboxMenuItem`. Several of these classes implement the `MenuContainer` interface. Once you've created a few menus, you'll probably find the process quite natural, but it's hard to describe until you see what all the objects are. So this chapter describes most of the menu classes first and then shows an example demonstrating their use.

All the components covered in previous chapters were subclasses of `Component`. Most of the objects in this chapter subclass `MenuComponent`, which encapsulates the common functionality of menu objects. The `MenuComponent` class hierarchy is shown in Figure 10.1.

Figure 10.1: MenuComponent class hierarchy

[Graphic: Figure 10-1]

To display a Menu, you must first put it in a MenuBar, which you add to a Frame. (Pop-up menus are different in that they don't need a Frame.) A Menu can contain MenuItem as well as other menus that form submenus. CheckboxMenuItem is a specialized MenuItem that (as you might guess) the user can toggle like a Checkbox. One way to visualize how all these things work together is to imagine a set of curtains. The different MenuItem components are the fabrics and panels that make up the curtains. The Menus are the curtains. They get hung from the MenuBar, which is like a curtain rod. Then you place the MenuBar curtain rod into the Frame (the window, in our metaphor), curtains and all.

It might puzzle you that a Menu is a subclass of MenuItem, not the other way around. This is because a Menu can appear on a Menu just like another MenuItem, which would not be possible if the hierarchy was the other way around. Figure 10.2 points out the different pieces involved in the creation of a menu: the MenuBar and various kinds of menu items, including a submenu.

Figure 10.2: The pieces that make up a Menu

[Graphic: Figure 10-2]

10.1 MenuComponent

MenuComponent is an abstract class that is the parent of all menu-related objects. You will never create an instance of the object. Nor are you likely to subclass it yourself--to make the subclass work, you'd have to provide your own peer on every platform where you want the application to run.

MenuComponent Methods

Constructor

public MenuComponent ()--cannot be called directly

> Since MenuComponent is an abstract class, you cannot create an instance of the object. This method is called when you create an instance of one of its children.

Fonts

public Font getFont ()

> The `getFont()` method retrieves the font associated with the `MenuComponent` from `setFont()`. If the current object's font has not been set, the parent menu's font is retrieved. If there is no parent and the current object's font has not been set, `getFont()` returns `null`.

public void setFont (Font f)

> The `setFont()` method allows you to change the font of the particular menu-related component to `f`. When a `MenuComponent` is first created, the initial font is `null`, so the parent menu's font is used.
>
> **NOTE:**

Some platforms do not support changing the fonts of menu items. Where supported, it can make some pretty ugly menus.

Names

The name serves as an alternative, nonlocalized reference identifier for menu components. If your event handlers compare menu label strings to an expected value and labels are localized for a new environment, the approach fails.

public String getName ()

> The `getName()` method retrieves the name of the menu component. Every instance of a subclass of `MenuComponent` is named when it is created.

public void setName (String name)

> The `setName()` method changes the current name of the component to `name`.

Peers

public MenuComponentPeer getPeer () ☆

> The `getPeer()` method returns a reference to the `MenuComponent` peer as a `MenuComponentPeer`.

public synchronized void removeNotify ()

The `removeNotify()` method destroys the peer of the `MenuComponent` and removes it from the screen. `addNotify()` will be specific to the subclass.

Events

Event handling is slightly different between versions. If using the 1.0 event model, use `postEvent()`. Otherwise, use `dispatchEvent()` to post an event to this `MenuComponent` or `processEvent()` to receive and handle an event. Remember not to mix versions within your programs.

public boolean postEvent (Event e) ☆

The `postEvent()` method posts `Event e` to the `MenuComponent`. The event is delivered to the `Frame` at the top of the object hierarchy that contains the selected `MenuComponent`. The only way to capture this event before it gets handed to the `Frame` is to override this method. There are no helper functions as there are for `Components`. Find out which `MenuComponent` triggered the event by checking `e.arg`, which contains its label, or `((MenuItem)e.target).getName()` for the nonlocalized name of the target.

```
public boolean postEvent (Event e) {
    // Use getName() vs. e.arg for localization possibility
    if ("About".equals (((MenuItem)e.target).getName()))
        playLaughingSound(); // Help request
    return super.postEvent (e);
}
```

If you override this method, in order for this `Event` to propagate to the `Frame` that contains the `MenuComponent`, you must call the original `postEvent()` method (`super.postEvent(e)`).

The actual value returned by `postEvent()` is irrelevant.

public final void dispatchEvent(AWTEvent e) ★

The `dispatchEvent()` method allows you to post new AWT events to this menu component's listeners. `dispatchEvent()` tells the `MenuComponent` to deal with the `AWTEvent e` by calling its `processEvent()` method. This method is similar to Java 1.0's `postEvent()` method. Events delivered in this way bypass the system's event queue. It's not clear why you would want to bypass the event queue, except possibly to deliver some kind of high priority event.

protected void processEvent(AWTEvent e) ★

The `processEvent()` method receives all `AWTEvent`s with a subclass of `MenuComponent` as its target. `processEvent()` then passes them along for processing. When you subclass a child class, overriding `processEvent()` allows you to process all events without having to provide listeners. However, remember to call `super.processEvent(e)` last to ensure regular functionality is still executed. This is like overriding `postEvent()` using the 1.0 event model.

Miscellaneous methods

public MenuContainer getParent ()

The `getParent()` method returns the parent `MenuContainer` for the `MenuComponent`. `MenuContainer` is an interface that is implemented by `Component` (in 1.1 only), `Frame`, `Menu`, and `MenuBar`. This means that `getParent()` could return any one of the four.

protected String paramString ()

The `paramString()` method of `MenuComponent` helps build up the string to display when `toString()` is called for a subclass. At the `MenuComponent` level, the current name of the object is appended to the output.

public String toString ()--can be called by user for subclass

The `toString()` method at the `MenuComponent` level cannot be called directly. This `toString()` method is called when you call a subclass's `toString()` and the specifics of the subclass is added between the brackets ([and]). At this level, the results would be:

```
java.awt.MenuComponent[aname1]
```

PREVIOUS
ItemSelectable

HOME
BOOK INDEX

NEXT
MenuContainer

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

11. Scrolling

Contents:

This chapter describes how Java deals with scrolling. AWT provides two means for scrolling. The first is the fairly primitive `Scrollbar` object. It really provides only the means to read a value from a slider setting. Anything else is your responsibility: if you want to display the value of the setting (for example, if you're using the scrollbar as a volume control) or want to change the display (if you're using scrollbars to control an area that's too large to display), you have to do it yourself. The `Scrollbar` reports scrolling actions through the standard event mechanisms; it is up to the programmer to handle those events and perform the scrolling.

Unlike other components, which generate an `ACTION_EVENT` when something exciting happens, the `Scrollbar` generates five events: `SCROLL_LINE_UP`, `SCROLL_LINE_DOWN`, `SCROLL_PAGE_UP`, `SCROLL_PAGE_DOWN`, and `SCROLL_ABSOLUTE`. In Java 1.0, none of these events trigger a default event handler like the `action()` method. To work with them, you must override the `handleEvent()` method. With Java 1.1, you handle scrolling events by registering an `AdjustmentListener` with the `Scrollbar.addAdjustmentListener()` method; when adjustment events occur, the listener's `adjustmentValueChanged()` method is called.

Release 1.1 of AWT also includes a `ScrollPane` container object; it is a response to one of the limitations of AWT 1.0. A `ScrollPane` is like a `Panel`, but it has scrollbars and scrolling built in. In this sense, it's like `TextArea`, which contains its own scrollbars. You could use a `ScrollPane` to implement a drawing pad that could cover an arbitrarily large area. This saves you the burden of implementing scrolling yourself: generating scrollbars, handling their events, and figuring out how to redisplay the screen accordingly.

Both `Scrollbar` and `ScrollPane` take advantage of the `Adjustable` interface. `Adjustable` defines the common scrolling activities of the two classes. The `Scrollbar` class implements `Adjustable`; a `ScrollPane` has two methods that return an `Adjustable` object, one for each scrollbar. Currently, you can use the `ScrollPane`'s "adjustables" to find out the scrollbar settings in each direction. You can't change the settings or register `AdjustmentListener`s; the appropriate methods exist, but they don't do anything. It's not clear whether this is appropriate behavior or a bug (remember, an interface only lists methods that must be present but doesn't require them to do anything); it may change in a later release.

11.1 Scrollbar

Scrollbars come in two flavors: horizontal and vertical. Although there are several methods for setting the page size, scrollbar range (minimum and maximum values), and so on, basically all you can do is get and set the scrollbar's value. Scrollbars don't contain any area to display their value, though if you want one, you could easily attach a label.

To work with a `Scrollbar`, you need to understand the pieces from which it is built. Figure 11.1 identifies each of the pieces. At both ends are arrows, which are used to change the `Scrollbar` value the default amount (one unit) in the direction selected. The paging areas are used to change the `Scrollbar` value one page (ten units by default) at a time in the direction selected. The slider can be moved to set the scrollbar to an arbitrary value within the available range.

Figure 11.1: Scrollbar elements

[Graphic: Figure 11-1]

Scrollbar Methods

Constants

There are two direction specifiers for `Scrollbar`. The direction tells the `Scrollbar` which way to orient itself. They are used in the constructors, as a parameter to `setOrientation()`, and as the return value for the `getOrientation()` method.

public final static int HORIZONTAL

HORIZONTAL is the constant for horizontal orientation.

public final static int VERTICAL

VERTICAL is the constant for vertical orientation.

Constructors

public Scrollbar (int orientation, int value, int visible, int minimum, int maximum)

The Scrollbar constructor creates a Scrollbar with a direction of orientation and initial value of value. visible is the size of the slider. minimum and maximum are the range of values that the Scrollbar can be. If orientation is not HORIZONTAL or VERTICAL, the constructor throws the run-time exception IllegalArgumentException. If maximum is below the value of minimum, the scrollbar's minimum and maximum values are both set to minimum. If value is outside the range of the scrollbar, it is set to the limit it exceeded. The default line scrolling amount is one. The default paging amount is ten.

If you are using the scrollbar to control a visual object, visible should be set to the amount of a displayed object that is on the screen at one time, relative to the entire size of the object (i.e., relative to the scrollbar's range: maximum - minimum). Some platforms ignore this parameter and set the scrollbar to a fixed size.

public Scrollbar (int orientation)

This constructor for Scrollbar creates a Scrollbar with the direction of orientation. In Java 1.0, the initial settings for value, visible, minimum, and maximum are 0. In Java 1.1, the default value for visible is 10, and the default for maximum is 100; the other values default to 0. If orientation is not HORIZONTAL or VERTICAL, the constructor throws the run-time exception IllegalArgumentException. This constructor is helpful if you want to reserve space for the Scrollbar on the screen, to be configured later. You would then use the setValues() method to configure the scrollbar.

public Scrollbar ()

This constructor creates a VERTICAL Scrollbar. In Java 1.0, the initial settings for value, visible, minimum, and maximum are 0. In Java 1.1, the default value for visible is 10, and the default for maximum is 100; the other values default to 0. You would then use the `setValues()` method to configure the scrollbar.

Figure 11.2 shows both vertical and horizontal scrollbars. It also demonstrates a problem you'll run into if you're not careful. If not constrained by the `LayoutManager`, scrollbars can get very fat. The result is rarely pleasing. The solution is to place scrollbars in layout managers that restrict width for vertical scrollbars or height for horizontal ones. The side regions (i.e., everything except the center) of a border layout are ideal. In the long term, the solution will be scrollbars that give you their maximum size and layout managers that observe the maximum size.

Figure 11.2: Vertical and horizontal scrollbars

[Graphic: Figure 11-2]

Adjustable Methods

public int getOrientation ()

The `getOrientation()` method returns the current orientation of the scrollbar: either `Scrollbar.HORIZONTAL` or `Scrollbar.VERTICAL`.

public synchronized void setOrientation (int orientation) ★

The `setOrientation()` method changes the orientation of the scrollbar to `orientation`, which must be either `Scrollbar.HORIZONTAL` or `Scrollbar.VERTICAL`. If `orientation` is not `HORIZONTAL` or `VERTICAL`, this method throws the run-time exception `IllegalArgumentException`. It was not possible to change the orientation of a scrollbar prior to Java 1.1.

public int getVisibleAmount () ★
public int getVisible () ☆

The `getVisibleAmount()` method gets the `visible` setting of the `Scrollbar`. If the scrollbar's `Container` is resized, the `visible` setting is not automatically changed.

`getVisible()` is the Java 1.0 name for this method.

public synchronized void setVisibleAmount (int amount) ★

The `setVisibleAmount()` method changes the current `visible` setting of the `Scrollbar` to `amount`.

public int getValue ()

The `getValue()` method is probably the most frequently called method of `Scrollbar`. It returns the current value of the scrollbar queried.

public synchronized void setValue (int value)

The `setValue()` method changes the value of the scrollbar to `value`. If `value` exceeds a scrollbar limit, the scrollbar's new value is set to that limit. In Java 1.1, this method is synchronized; it was not in earlier versions.

public int getMinimum ()

The `getMinimum()` method returns the current minimum setting for the scrollbar.

public synchronized void setMinimum (int minimum) ★

The `setMinimum()` method changes the `Scrollbar`'s minimum value to `minimum`. The current setting for the `Scrollbar` may change to `minimum` if `minimum` increases above `getValue()`.

public int getMaximum ()

The `getMaximum()` method returns the current maximum setting for the scrollbar.

public synchronized void setMaximum (int maximum) ★

The `setMaximum()` method changes the maximum value of the `Scrollbar` to `maximum`. The current setting for the `Scrollbar` may change to `maximum` if `maximum` decreases below `getValue()`.

public synchronized void setValues (int value, int visible, int minimum, int maximum)

The `setValues()` method changes the `value`, `visible`, `minimum`, and `maximum` settings all at once. In Java 1.0.2, separate methods do not exist for changing `visible`, `minimum`, or `maximum`. The scrollbar's value is set to `value`, visible to `visible`, minimum to `minimum`, and maximum to `maximum`. If `maximum` is below the value of `minimum`, it is set to `minimum`. If `value` is outside the range of the scrollbar, it is set to the limit it exceeded. In Java 1.1, this method is synchronized; it was not in earlier versions.

public int getUnitIncrement () ★
public int getLineIncrement () ☆

The `getUnitIncrement()` method returns the current line increment. This is the amount the scrollbar will scroll if the user clicks on one of the scrollbar's arrows.

`getLineIncrement()` is the Java 1.0 name for this method.

public void setUnitIncrement (int amount) ★
public void setLineIncrement (int amount) ☆

The `setUnitIncrement()` method changes the line increment amount to `amount`.

`setLineIncrement()` is the Java 1.0 name for this method.

Changing the line increment amount was not possible in Java 1.0.2. This method acted like it returned successfully, and `getLineIncrement()` returned the new value, but the `Scrollbar` changed its value by only one (the default) when you clicked on one of the arrows. However, you could work around this defect by explicitly handling the `SCROLL_LINE_UP` and `SCROLL_LINE_DOWN` events: get the correct line increment, adjust the display appropriately, and then set call `setValue()` to correct the scrollbar's value. This workaround is not needed in Java 1.1.

public int getBlockIncrement () ★
public int getPageIncrement () ☆

The `getBlockIncrement()` method returns the current paging increment. This is the amount the scrollbar will scroll if the user clicks between the slider and one of the scrollbar's arrows.

`getPageIncrement()` is the Java 1.0 name for this method.

public void setBlockIncrement (int amount) ★
public void setPageIncrement (int amount) ☆

The `setBlockIncrement()` method changes the paging increment amount to `amount`.

`setPageIncrement()` is the Java 1.0 name for this method.

Changing the paging increment amount was not possible in Java 1.0.2. This method acts like it returns successfully, and `getPageIncrement()` returns the new value, but the `Scrollbar` changes its value only by 10 (the default) when you click on one of the paging areas. However, you can work around this defect by explicitly handling the `SCROLL_PAGE_UP` and `SCROLL_PAGE_DOWN` events: get the correct page increment, adjust the display appropriately, and then set call `setValue()` to correct the scrollbar's value. This workaround is not necessary in Java 1.1.

Miscellaneous methods

public synchronized void addNotify ()

The `addNotify()` method creates the `Scrollbar`'s peer. If you override this method, call `super.addNotify()` first. You will then be able to do everything you need with the information about the newly created peer.

protected String paramString ()

`Scrollbar` doesn't have its own `toString()` method; when you call the `toString()` method of a `Scrollbar`, you are actually calling the method `Component.toString()`. This in turn calls `paramString()`, which builds the string to display. For a `Scrollbar`, `paramString()` puts the scrollbar's value, visibility, minimum, maximum, and direction into the string. In Java 1.0, there is a minor bug in the output. Instead of displaying the scrollbar's `visible` setting (an integer), `paramString()` displays the component's `visible` setting (a boolean). (This is corrected in Java 1.1.) The following `String` is the result of calling `toString()` for a horizontal `Scrollbar` that hasn't been configured yet:

```
java.awt.Scrollbar[0,0,0x0,invalid,val=0,vis=true,min=0,max=0,horz]
```

Scrollbar Events

With the 1.0 event model, scrollbars generate five kinds of events in response to user interaction: `SCROLL_LINE_UP`, `SCROLL_LINE_DOWN`, `SCROLL_PAGE_UP`, `SCROLL_PAGE_DOWN`, and `SCROLL_ABSOLUTE`. The event that occurs depends on what the user did, as shown in Table 11.1; the event type is specified in the `id` field of the `Event` object passed to `handleEvent()`. However, as a programmer, you often do not care which of these five events happened. You care only about the scrollbar's new value, which is always passed as the `arg` field of the `Event` object.

Table 11.1: Scrollbar Events

Event Type (Event.id)	**Event Meaning**
`SCROLL_ABSOLUTE`	User drags slider.
`SCROLL_LINE_DOWN`	User presses down arrow.
`SCROLL_LINE_UP`	User presses up arrow.
`SCROLL_PAGE_DOWN`	User selects down paging area.
`SCROLL_PAGE_UP`	User selects up paging area.

Because scrollbar events do not trigger any default event handlers (like `action()`), it is necessary to override the `handleEvent()` method to deal with them. Unless your version of `handleEvent()` deals with all conceivable events, you must ensure that the original `handleEvent()` method is called. The simplest way is to have the return statement call `super.handleEvent()`.

Most `handleEvent()` methods first identify the type of event that occurred. The following two code blocks demonstrate different ways of checking for the `Scrollbar` events.

```
if ((e.id == Event.SCROLL_LINE_UP) ||
    (e.id == Event.SCROLL_LINE_DOWN) ||
    (e.id == Event.SCROLL_PAGE_UP) ||
    (e.id == Event.SCROLL_PAGE_DOWN) ||
    (e.id == Event.SCROLL_ABSOLUTE)) {
    // Then determine which Scrollbar was selected and act upon it
}
```

Or more simply:

```
if (e.target instanceof Scrollbar) {
    // Then determine which Scrollbar was selected and act upon it.
}
```

Although the second code block is simpler, the first is the better choice because it is more precise. For example, what would happen if mouse events are passed to scrollbars? Different Java platforms differ most in the types of events passed to different objects; Netscape Navigator 3.0 for Windows 95 sends `MOUSE_ENTER`, `MOUSE_EXIT`, and `MOUSE_MOVE` events to the `Scrollbar`.[1] The second code block executes for all the mouse events--in fact, any event coming from a `Scrollbar`. Therefore, it executes much more frequently (there can be many `MOUSE_MOVE` events), leading to poor interactive performance.

[1] MOUSE_UP, MOUSE_DOWN, and MOUSE_DRAG are not generated since these operations generate SCROLL events.

Another platform-specific issue is the way the system generates SCROLL_ABSOLUTE events. Some platforms generate many events while the user drags the scrollbar. Others don't generate the event until the user stops dragging the scrollbar. Some implementations wait until the user stops dragging the scrollbar and then generate a flood of SCROLL_ABSOLUTE events for you to handle. In theory, it does not matter which is happening, as long as your event-processing code is tight. If your event-processing code is time consuming, you may wish to start another thread to perform the work. If the thread is still alive when the next event comes along, flag it down, and restart the operation. Listeners and 1.1 event handling

With the 1.1 event model, you register an AdjustmentListener by calling the addAdjustmentListener() method. Then when the user moves the Scrollbar slider, the AdjustmentListener.adjustmentValueChanged() method is called through the protected Scrollbar.processAdjustmentEvent() method. Key, mouse, and focus listeners are registered through the three Component methods of addKeyListener(), addMouseListener(), and addFocusListener(), respectively. Because you need to register a separate listener for mouse events, you no longer have the problem of distinguishing between mouse events and slider events. An adjustment listener will never receive mouse events.

public void addAdjustmentListener(AdjustmentListener listener) ★

The addAdjustmentListener() method registers listener as an object interested in being notified when an AdjustmentEvent passes through the EventQueue with this Scrollbar as its target. The method listener.adjustmentValueChanged() is called when an event occurs. Multiple listeners can be registered.

public void removeAdjustmentListener(ItemListener listener) ★

The removeAdjustmentListener() method removes listener as a interested listener. If listener is not registered, nothing happens.

protected void processEvent(AWTEvent e) ★

The processEvent() method receives every AWTEvent with this Scrollbar as its target. processEvent() then passes it along to any listeners for processing. When you subclass Scrollbar, overriding processEvent() allows you to process all events yourself, before sending them to any listeners. In a way, overriding processEvent() is like overriding handleEvent() using the 1.0 event model.

If you override the `processEvent()` method, remember to call the `super.processEvent(e)` method last to ensure that regular event processing can occur. If you want to process your own events, it's a good idea to call `enableEvents()` (inherited from `Component`) to ensure that events are delivered even in the absence of registered listeners.

protected void processAdjustmentEvent(ItemEvent e) ★

The `processAdjustmentEvent()` method receives all `AdjustmentEvent`s with this `Scrollbar` as its target. `processAdjustmentEvent()` then passes them along to any listeners for processing. When you subclass `Scrollbar`, overriding `processAdjustmentEvent()` allows you to process all events yourself, before sending them to any listeners.

If you override `processAdjustmentEvent()`, you must remember to call `super.processAdjustmentEvent(e)` last to ensure that regular event processing can occur. If you want to process your own events, it's a good idea to call `enableEvents()` (inherited from `Component`) to ensure that events are delivered even in the absence of registered listeners.

PREVIOUS
PopupMenu

HOME
BOOK INDEX

NEXT
Scrolling An Image

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

12. Image Processing

Contents:

The image processing parts of Java are buried within the `java.awt.image` package. The package consists of three interfaces and eleven classes, two of which are abstract. They are as follows:

- The `ImageObserver` interface provides the single method necessary to support the asynchronous loading of images. The interface implementers watch the production of an image and can react when certain conditions arise. We briefly touched on `ImageObserver` when we discussed the `Component` class (in Chapter 5, *Components*), because `Component` implements the interface.

- The `ImageConsumer` and `ImageProducer` interfaces provide the means for low level image creation. The `ImageProducer` provides the source of the pixel data that is used by the `ImageConsumer` to create an `Image`.

- The `PixelGrabber` and `ImageFilter` classes, along with the `AreaAveragingScaleFilter`, `CropImageFilter`, `RGBImageFilter`, and `ReplicateScaleFilter` subclasses, provide the tools for working with images. `PixelGrabber` consumes pixels from an `Image` into an array. The `ImageFilter` classes modify an existing image to produce another `Image` instance. `CropImageFilter` makes smaller images; `RGBImageFilter` alters pixel colors, while `AreaAveragingScaleFilter` and `ReplicateScaleFilter` scale images up and down using different algorithms. All of these classes implement `ImageConsumer` because they take pixel data as input.

- `MemoryImageSource` and `FilteredImageSource` produce new images. `MemoryImageSource` takes an array and creates an image from it. `FilteredImageSource` uses an `ImageFilter` to read and modify data from another image and produces the new image based on the original. Both `MemoryImageSource` and `FilteredImageSource` implement `ImageProducer` because they produce new pixel data.

- `ColorModel` and its subclasses, `DirectColorModel` and `IndexColorModel`, provide the palette of colors available when creating an image or tell you the palette used when using `PixelGrabber`.

The classes in the `java.awt.image` package let you create `Image` objects at run-time. These classes can be used to rotate images, make images transparent, create image viewers for unsupported graphics formats, and more.

12.1 ImageObserver

As you may recall from Chapter 2, *Simple Graphics*, the last parameter to the `drawImage()` method is the image's `ImageObserver`. However, in Chapter 2, *Simple Graphics* I also said that you can use `this` as the image observer and forget about it. Now it's time to ask the obvious questions: what is an image observer, and what is it for?

Because `getImage()` acquires an image asynchronously, the entire `Image` object might not be fully loaded when `drawImage()` is called. The `ImageObserver` interface provides the means for a component to be told asynchronously when additional information about the image is available. The `Component` class implements the `imageUpdate()` method (the sole method of the `ImageObserver` interface), so that method is inherited by any component that renders an image. Therefore, when you call `drawImage()`, you can pass `this` as the final argument; the component on which you are drawing serves as the `ImageObserver` for the drawing process. The communication between the image observer and the image consumer happens behind the scenes; you never have to worry about it, unless you want to write your own `imageUpdate()` method that does something special as the image is being loaded.

If you call `drawImage()` to display an image created in local memory (either for double buffering or from a `MemoryImageSource`), you can set the `ImageObserver` parameter of `drawImage()` to `null` because no asynchrony is involved; the entire image is available immediately, so an `ImageObserver` isn't needed.

ImageObserver Interface

Constants

The various flags associated with the `ImageObserver` are used for the `infoflags` argument to `imageUpdate()`. The flags indicate what kind of information is available and how to interpret the other arguments to `imageUpdate()`. Two or more flags are often combined (by an OR operation) to show that several kinds of information are available.

public static final int WIDTH

When the `WIDTH` flag is set, the `width` argument to `imageUpdate()` correctly indicates the image's width. Subsequent calls to `getWidth()` for the `Image` return the valid image width. If you call `getWidth()` before this flag is set, expect it to return -1.

public static final int HEIGHT

When the `HEIGHT` flag is set, the `height` argument to `imageUpdate()` correctly indicates the image's height. Subsequent calls to `getHeight()` for the `Image` return the valid image height. If you call `getHeight()` before this flag is set, expect it to return -1.

public static final int PROPERTIES

When the `PROPERTIES` flag is set, the image's properties are available. Subsequent calls to `getProperty()` return valid image properties.

public static final int SOMEBITS

When the `SOMEBITS` flag of `infoflags` (from `imageUpdate()`) is set, the image has started loading and at least some of its content are available for display. When this flag is set, the `x`, `y`, `width`, and `height` arguments to `imageUpdate()` indicate the bounding rectangle for the portion of the image that has been delivered so far.

public static final int FRAMEBITS

When the `FRAMEBITS` flag of `infoflags` is set, a complete frame of a multiframe image has been loaded and can be drawn. The remaining parameters to `imageUpdate()` should be ignored (`x`, `y`, `width`, `height`).

public static final int ALLBITS

When the `ALLBITS` flag of infoflags is set, the image has been completely loaded and can be drawn. The remaining parameters to `imageUpdate()` should be ignored (`x`, `y`, `width`, `height`).

public static final int ERROR

When the `ERROR` flag is set, the production of the image has stopped prior to completion because of a severe problem. `ABORT` may or may not be set when `ERROR` is set. Attempts to reload the image will fail. You might get an `ERROR` because the URL of the `Image` is invalid (file not found) or the image file itself is invalid (invalid size/content).

public static final int ABORT

When the `ABORT` flag is set, the production of the image has aborted prior to completion. If `ERROR` is not set, a subsequent attempt to draw the image may succeed. For example, an image would abort without an error if a network error occurred (e.g., a timeout on the HTTP connection).

Method

public boolean imageUpdate (Image image, int infoflags, int x, int y, int width, int height)

The `imageUpdate()` method is the sole method in the `ImageObserver` interface. It is called whenever information about an image becomes available. To register an image observer for an image, pass an object that implements the `ImageObserver` interface to `getWidth()`, `getHeight()`, `getProperty()`, `prepareImage()`, or `drawImage()`.

The `image` parameter to `imageUpdate()` is the image being rendered on the observer. The `infoflags` parameter is a set of `ImageObserver` flags ORed together to signify the current information available about `image`. The meaning of the `x`, `y`, `width`, and `height` parameters depends on the current `infoflags` settings.

Implementations of `imageUpdate()` should return `true` if additional information about the image is desired; returning `false` means that you don't want any additional information, and consequently, `imageUpdate()` should not be called in the future for this image. The default `imageUpdate()` method returns `true` if neither `ABORT` nor `ALLBITS` are set in the `infoflags`--that is, the method `imageUpdate()` is interested in further information if no errors have occurred and the image is not complete. If either flag is set, `imageUpdate()` returns `false`.

You should not call `imageUpdate()` directly--unless you are developing an `ImageConsumer`, in which case you may find it worthwhile to override the default `imageUpdate()` method, which all components inherit from the `Component` class.

Overriding imageUpdate

Instead of bothering with the `MediaTracker` class, you can override the `imageUpdate()` method and use it to notify you when an image is completely loaded. Example 12.1 demonstrates the use of `imageUpdate()`, along with a way to force your images to load immediately. Here's how it works: the `init()` method calls `getImage()` to request image loading at some time in the future. Instead of waiting for `drawImage()` to trigger the loading process, `init()` forces loading to start by calling `prepareImage()`, which also registers an image observer. `prepareImage()` is a method of the

`Component` class discussed in Chapter 5, *Components*.

The `paint()` method doesn't attempt to draw the image until the variable `loaded` is set to `true`. The `imageUpdate()` method checks the `infoflags` argument to see whether `ALLBITS` is set; when it is set, `imageUpdate()` sets `loaded` to `true`, and schedules a call to `paint()`. Thus, `paint()` doesn't call `drawImage()` until the method `imageUpdate()` has discovered that the image is fully loaded.

Example 12.1: imageUpdate Override.

```
import java.applet.*;
import java.awt.*;
import java.awt.image.ImageObserver;
public class imageUpdateOver extends Applet {
    Image image;
    boolean loaded = false;
    public void init () {
        image = getImage (getDocumentBase(), "rosey.jpg");
        prepareImage (image, -1, -1, this);
    }
    public void paint (Graphics g) {
        if (loaded)
            g.drawImage (image, 0, 0, this);
    }
    public void update (Graphics g) {
        paint (g);
    }
    public synchronized boolean imageUpdate (Image image, int infoFlags,
                    int x, int y, int width, int height) {
        if ((infoFlags & ImageObserver.ALLBITS) != 0) {
            loaded = true;
            repaint();
            return false;
        } else {
            return true;
        }
    }
}
```

Note that the call to `prepareImage()` is absolutely crucial. It is needed both to start image loading and to register the image observer. If `prepareImage()` were omitted, `imageUpdate()` would never be called, `loaded` would not be set, and `paint()` would never attempt to draw the image. As an alternative, you could use the `MediaTracker` class to force loading to start and monitor the loading process; that approach might give you some additional flexibility.

PREVIOUS | HOME | NEXT
ScrollPane | BOOK INDEX | ColorModel

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

13. AWT Exceptions and Errors

Contents:

This chapter describes `AWTException`, `IllegalComponentStateException`, and `AWTError`. `AWTException` is a subclass of `Exception`. It is not used by any of the public classes in `java.awt`; you may, however, find it convenient to throw `AWTException` within your own code. `IllegalComponentStateException` is another `Exception` subclass, which is new to Java 1.1. This exception is used when you try to do something with a `Component` that is not yet appropriate. `AWTError` is a subclass of `Error` that is thrown when a serious problem occurs in AWT--for example, the environment is unable to get the platform's `Toolkit`.

13.1 AWTException

`AWTException` is a generic exception that can be thrown when an exceptional condition has occurred within AWT. None of the AWT classes throw this. If you subclass any of the AWT classes, you can throw an `AWTException` to indicate a problem. Using `AWTException` is slightly preferable to creating your own `Exception` subclass because you do not have to generate another class file. Since it is a part of Java, `AWTException` is guaranteed to exist on the run-time platform.

If you throw an instance of `AWTException`, like any other `Exception`, it must be caught in a `catch` clause or declared in the `throws` clause of the method.

AWTException Method

Constructor

public AWTException (String message)

> The sole constructor creates an `AWTException` with a detailed message of `message`. This message can be retrieved using `getMessage()`, which it inherits from `Exception` (and which is required by the `Throwable` interface). If you do not want a detailed message, `message` may be `null`.

Throwing an AWTException

An `AWTException` is used the same way as any other `Throwable` object. Here's an example:

```
if (someProblem) {
    throw new AWTException ("Problem Encountered While Initializing");
}
```

PREVIOUS | HOME | NEXT

ImageFilter | BOOK INDEX | IllegalComponentStateException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

14. And Then There Were Applets

Contents:

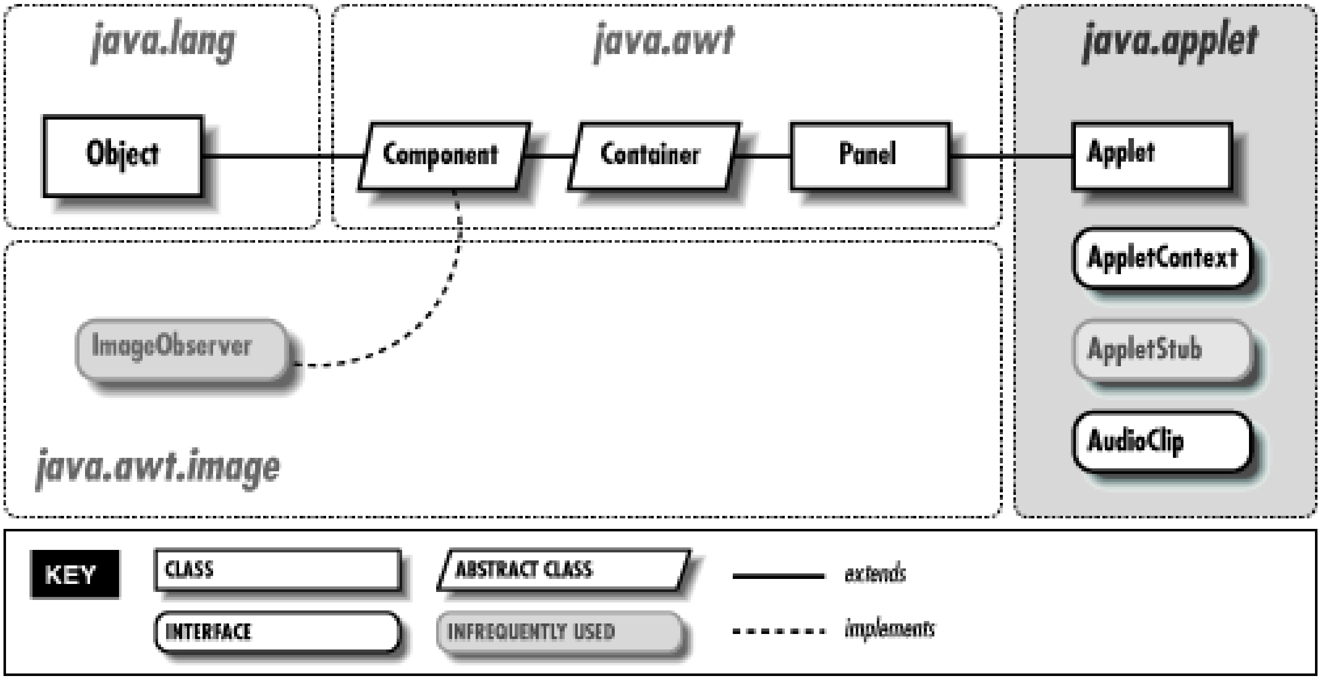
Although it is not part of the `java.awt` package, the `java.applet` package is closely related. The `java.applet` package provides support for running an applet in the context of a World Wide Web browser. It consists of one class (`Applet`) and three interfaces (`AppletContext`, `AudioClip`, and `AppletStub`). The `Applet` class supports the "applet life cycle" methods (`init()`, `start()`, `stop()`, `destroy()`) that you override to write an applet. `AudioClip` provides support for audio within applets. (Applications use the `sun.audio` package for audio support; `sun.audio` is also covered in this chapter.) The `AppletStub` and `AppletContext` interfaces provide a way for the applet to interact with its run-time environment. Many of the methods of `AppletStub` and `AppletContext` are duplicated in the `Applet` class.

14.1 What's a Java Applet?

Much of the initial excitement about Java centered around applets. Applets are small Java programs that can be embedded within HTML pages and downloaded and executed by a web browser. Because executing code from random Internet sites presents a security risk, Java goes to great lengths to ensure the integrity of the program executing and to prevent it from performing any unauthorized tasks.

An applet is a specific type of Java `Container`. The class hierarchy of an applet is shown in Figure 14.1.

Figure 14.1: Applet class hierarchy



When you are writing an applet, remember that you can use the features of its ancestors. In particular, remember to check the methods of the `Component`, `Container`, and `Panel` classes, which are inherited by the `Applet` class.

Applet Methods

All the methods of `Applet`, except `setStub()`, either need to be overridden or are methods based on one of the `java.applet` interfaces. The system calls `setStub()` to set up the context of the interfaces. The browser implements the `AppletContext` and `AppletStub` interfaces. Constructor

public Applet ()

> The system calls the `Applet` constructor when the applet is loaded and before it calls `setStub()`, which sets up the applet's stub and context. When you subclass `Applet`, you usually do not provide a constructor. If you do provide a constructor, you do not have access to the `AppletStub` or `AppletContext` and, therefore, may not call any of their methods.

AppletStub setup

public final void setStub (AppletStub stub)

> The `setStub()` method of `Applet` is called by the browser when the applet is loaded into the system. It sets the `AppletStub` of the applet to `stub`. In turn, the `AppletStub` contains the applet's `AppletContext`.

Applet information methods

Several methods of `Applet` provide information that can be used while the applet is running.

public AppletContext getAppletContext ()

> The `getAppletContext()` method returns the current `AppletContext`. This is part of the applet's stub, which is set by the system when `setStub()` is called.

public URL getCodeBase ()

> The `getCodeBase()` method returns the complete URL of the *.class* file that contains the applet. This method can be used with the `getImage()` or the `getAudioClip()` methods, described later in this chapter, to load an image or audio file relative to the *.class* file location.

public URL getDocumentBase ()

> The `getDocumentBase()` method returns the complete URL of the *.html* file that loaded the applet. This can be used with the `getImage()` or `getAudioClip()` methods, described later in this chapter, to load an image or audio file relative to the *.html* file.

public String getParameter (String name)

> The `getParameter()` method allows you to get run-time parameters from within the `<APPLET>` tag of the *.html* file that loaded the applet. Parameters are defined by HTML `<PARAM>` tags, which have the form:
>
> ```
> <PARAM name="parameter" value="value>
> ```
>
> If the `name` parameter of `getParameter()` matches the `name` string of a `<PARAM>` tag, `getParameter()` returns the tag's `value` as a string. If `name` is not found within the `<PARAM>` tags of the `<APPLET>`, `getParameter()` returns `null`. The argument `name` is not case sensitive; that is, it matches parameter names regardless of case. Remember that `getParameter()` always returns a string, even though the parameter values might appear as integers or floating point numbers in the HTML file. In some situations, it makes sense to pass multiple values in a single parameter; if you do this, you have to parse the parameter string manually. Using a `StringTokenizer` will make the job easier.
>
> Enabling your applets to accept parameters allows them to be customized at run-time by the HTML author, without providing the source code. This provides greater flexibility on the Web without requiring any recoding. Example 14.1 shows how an applet reads parameters from an HTML file. It contains three parts: the HTML file that loads the applet, the applet source code, and the output from the applet.

Example 14.1: Getting Parameters from an HTML File

```
<APPLET CODE=ParamApplet WIDTH=100 HEIGHT=100>
<PARAM NAME=one VALUE=1.0>
<PARAM name=TWO value=TOO>
</APPLET>
public class ParamApplet extends java.applet.Applet {
    public void init () {
        String param;
        float one;
        String two;
        if ((param = getParameter ("ONE")) == null) {
            one = -1.0f;  // Not present
        } else {
            one = Float.valueOf (param).longValue();
        }
        if ((param = getParameter ("two")) == null) {
            two = "two";
        } else {
            two = param.toUpperCase();
        }
        System.out.println ("One: " + one);
        System.out.println ("Two: " + two);
    }
}
One: 1
Two: TOO
```

public String getAppletInfo ()

The `getAppletInfo()` method lets an applet provide a short descriptive string to the browser. This method is frequently overridden to return a string showing the applet's author and copyright information. How (or whether) to display this information is up to the browser. With *appletviewer*, this information is displayed when the user selects the Info choice under the Applet menu. Neither Netscape Navigator nor Internet Explorer currently display this information.

public String[][] getParameterInfo ()

The `getParameterInfo()` method lets an applet provide a two-dimensional array of strings describing the parameters it reads from `<PARAM>` tags. It returns an array of three strings for each parameter. In each array, the first `String` represents the parameter name, the second describes the data type, and the third is a brief description or range of values. Like `getAppletInfo()`, how (or whether) to display this information is up to the browser. With appletviewer, this information is displayed when the user selects the Info choice under the Applet menu. Neither Netscape Navigator nor Internet Explorer currently display this information. The following code shows how an applet might use `getParameterInfo()` and `getAppletInfo()`:

```
public String getAppletInfo() {
    String whoami = "By John Zukowski (c) 1997";
    return whoami;
}
public String[][] getParameterInfo() {
    String[][] strings = {
        {"parameter1",    "String",      "Background Color name"},
        {"parameter2",    "URL",         "Image File"},
        {"parameter3",    "1-10",        "Number in Series"}
    };
    return strings;
}
```

public void showStatus (String message)

The `showStatus()` method displays `message` on the browser's status line, if it has one. Again, how to display this string is up to the browser, and the browser can overwrite it whenever it wants. You should only use `showStatus()` for messages that the user can afford to miss.

public boolean isActive ()

The `isActive()` method returns the current state of the applet. While an applet is initializing, it is not active, and calls to `isActive()` return `false`. The system marks the applet active just prior to calling `start()`; after this point, calls to `isActive()` return `true`.

public Locale getLocale () ★

The `getLocale()` method retrieves the current `Locale` of the applet, if it has one. Using a `Locale` allows you to write programs that can adapt themselves to different languages and different regional variants. If no `Locale` has been set, `getLocale()` returns the default `Locale`. The default `Locale` has a user language of English and no region. To change the default `Locale`, set the system properties `user.language` and `user.region`, or call `Locale.setDefault()` (`setDefault()` verifies access rights with the security manager).[1]

[1] For more on the `Locale` class, see *Java Fundamental Classes Reference,* by Mark Grand, from O'Reilly & Associates.

Applet life cycle

The browser calls four methods of the `Applet` class to execute the applet. These methods constitute the applet's life cycle. The default versions don't do anything; you must override at least one of them to create a useful applet.

public void init ()

> The `init()` method is called once when the applet is first loaded. It should be used for tasks that need to be done only once. `init()` is often used to load images or sound files, set up the screen, get parameters out of the HTML file, and create objects the applet will need later. You should not do anything that might "hang" or wait indefinitely. In a sense, `init()` does things that might otherwise be done in an applet's constructor.

public void start ()

> The `start()` method is called every time the browser displays the web page containing the applet. `start()` usually does the "work" of the applet. It often starts threads, plays sound files, or does computation. `start()` may also be called when the browser is de-iconified.

public void stop ()

> The `stop()` method is called whenever the browser leaves the web page containing the applet. It should stop or suspend anything that the applet is doing. For example, it should suspend any threads that have been created and stop playing any sound files. `stop()` may also be called when the browser is iconified.

public void destroy ()

> The `destroy()` method is called when the browser determines that it no longer needs to keep the applet around--in practice, when the browser decides to remove the applet from its cache or the browser exits. After this point, if the browser needs to display the applet again, it will reload the applet and call the applet's `init()` method. `destroy()` gives the applet a final opportunity to release any resources it is using (for example, close any open sockets). Most applets don't need to implement `destroy()`. It is always a good idea to release resources as soon as they aren't needed, rather than waiting for `destroy()`. There are no guarantees about when `destroy()` will be called; if your browser has a sufficiently large cache, the applet may stay around for a very long time.

Applet-sizing methods

public void resize(int width, int height)

> The `resize()` method changes the size of the applet space to `width` x `height`. The browser

must support changing the applet space or else the sizing does not change. Netscape Navigator does not allow an applet to change its size; the applet is sized to the region allocated by the `<APPLET>` tag, period.

Because `Applet` is a subclass of `Component`, it inherits the Java 1.1 method `setSize()`, which has the same function.

public void resize (Dimension dim)

This `resize()` method calls the previous version of `resize()` with a width of `dim.width` and a height of `dim.height`.

Images

We have discussed `Image` objects extensively in Chapter 2, *Simple Graphics*, and Chapter 12, *Image Processing*, and used them in many of our examples. When writing an applet, you can use the `getImage()` method directly. In applications, you must go through `Toolkit` (which the following methods call) to get images.

public Image getImage (URL url)

The `getImage()` method loads the image file located at `url`. `url` must be a complete and valid URL. The method returns a system-specific object that subclasses `Image` and returns immediately. The `Image` is not loaded until needed, either by `prepareImage()`, `MediaTracker`, or `drawImage()`.

public Image getImage (URL url, String filename)

The `getImage()` method loads the image file located at `url` in `filename`. The applet locates the file relative to the specified URL; that is, if the URL ends with a filename, the applet removes the filename and appends the `filename` argument to produce a new URL. `getImage()` returns a system-specific object that subclasses `Image` and returns immediately. The `Image` is not loaded until needed, either by `prepareImage()`, `MediaTracker`, or `drawImage()`.

In most cases, the `url` argument is a call to `getDocumentBase()` or `getCodeBase()`; most often, image files are located in the same directory as the HTML file, the applet's Java class file, or their own subdirectory.

Audio

Every Java platform is guaranteed to understand Sun's AU file format, which contains a single channel of 8000 Hz µLaw encoded audio data.[2] Java applets do not require any helper applications to play audio;

they use the browser's audio capabilities. You can use an independent application, like Sun's *audiotool*, to control the volume. Of course, the user's workstation or PC needs audio hardware, but these days, it's hard to buy a computer that isn't equipped for audio.

> [2] The AU format is explained in the Audio File Format FAQ (version 3.10) located at *ftp://ftp.cwi.nl/pub/audio/index.html* in files *AudioFormats.part1* and *AudioFormats.part2*.

The Java Media Framework API is rumored to provide support for additional audio formats, like Microsoft's *.wav* files or Macintosh/SGI *.aiff* audio files. At present, if you want your Java program to play audio files in other formats, you must first convert the audio file to the *.au* format, using a utility like SOX (Sound Exchange).[3] Once converted, your Java program can play the resulting *.au* file normally. (If you are interested in more information about audio, look in the *alt.binaries.sounds.d* newsgroup.)

> [3] SOX is available at http://www.spies.com/Sox. The current version of SOX is 10; version 11 is in gamma release. The UNIX source is located in *sox10.tar.gz* , while the DOS executable is *sox10dos.zip*.

The `Applet` class provides two ways to play audio clips. The first mechanism provides a method to load and play an audio file once:

public void play (URL url)

> The `play()` method downloads and plays the audio file located at `url`. `url` must be a complete and valid URL. If `url` is invalid, no sound is played. Some environments throw an exception if the URL is invalid, but not all. Calling `play()` within an applet's `destroy()` method usually has no effect; the applet and its resources will probably be deallocated before `play()` has time to download the audio file.

public void play (URL url, String filename)

> This version of `play()` downloads and plays the audio file located at `url` in the file `filename`. The applet locates the file relative to the specified URL; that is, if the URL ends with a filename, the applet removes the filename and appends the `filename` argument to produce a new URL. If the resulting URL is invalid, no sound is played. Some environments throw an exception if the URL is invalid, but not all.
>
> In most cases, the `url` argument is a call to `getDocumentBase()` or `getCodeBase()`; most often, sound files are located in the same directory as the HTML file or the applet's Java class file. For some reason, you cannot have a double dot (`..`) in the URL of an audio file; you can in the URL of an image file. Putting a double dot in the URL of an audio file raises a security exception in an applet causing `play()` to fail.

The following applet plays an audio file located relative to the HTML file from which the applet was loaded:

```
import java.net.*;
import java.applet.*;
public class audioTest extends Applet {
    public void init () {
        System.out.println ("Before");
        play (getDocumentBase(), "audio/flintstones.au");
        System.out.println ("After");
    }
}
```

The second way to play audio files splits the process into two steps: you get an `AudioClip` object and then play it as necessary. This procedure eliminates a significant drawback to `play()`: if you call `play()` repeatedly, it reloads the audio file each time, making the applet much slower.

public AudioClip getAudioClip (URL url)

The `getAudioClip()` method loads the audio file located at `url`. `url` must be a complete and valid URL. Upon success, `getAudioClip()` returns an instance of a class that implements the `AudioClip` interface. You can then call methods in the `AudioClip` interface (see AudioClip Interface) to play the clip. If an error occurs during loading (e.g., because the file was not found or the URL was invalid), `getAudioClip()` returns `null`.

`getAudioClip()` sounds similar to `getImage()`, and it is. However, Java currently loads audio clips synchronously; it does not start a separate thread as it does for images. You may want to create a helper class that loads audio clips in a separate thread.

The actual class of the `AudioClip` object depends on the platform you are using; you shouldn't need to know it. If you are curious, the *appletviewer* uses the class `sun.applet.AppletAudioClip`; Netscape Navigator uses the class `netscape.applet.AppletAudioClip`.

public AudioClip getAudioClip (URL url , String filename)

This version of the `getAudioClip()` method loads the audio file located at `url` in the file `filename`. The applet locates the file relative to the specified URL; that is, if the URL ends with a filename, the applet removes the filename and appends the `filename` argument to produce a new URL. If the resulting URL is invalid, the file is not loaded. Upon success, `getAudioClip()` returns an instance of a class that implements the `AudioClip` interface. You can then call methods in the `AudioClip` interface (see AudioClip Interface) to play the clip. If an error occurs during loading (e.g., because the file was not found or the URL was invalid), `getAudioClip()` returns `null`.

In most cases, the `url` argument is a call to `getDocumentBase()` or `getCodeBase()`; most often, sound files are located in the same directory as the HTML file or the applet's Java class file.

PREVIOUS | HOME | NEXT
AWTError | BOOK INDEX | AudioClip Interface

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

15. Toolkit and Peers

Contents:

This chapter describes the `Toolkit` class and the purposes it serves. It also describes the `java.awt.peer` package of interfaces, along with how they fit in with the general scheme of things. The most important advice I can give you about the peer interfaces is not to worry about them. Unless you are porting Java to another platform, creating your own `Toolkit`, or adding any native component, you can ignore the peer interfaces.

15.1 Toolkit

The `Toolkit` object is an abstract class that provides an interface to platform-specific details like window size, available fonts, and printing. Every platform that supports Java must provide a concrete class that extends the `Toolkit` class. The Sun JDK provides a `Toolkit` for Windows NT/95 (`sun.awt.win32.MToolkit` [Java1.0] or `sun.awt.windows.MToolkit` [Java1.1]), Solaris/Motif (`sun.awt.motif.MToolkit`), and Macintosh (`sun.awt.macos.MToolkit`). Although the `Toolkit` is used frequently, both directly and behind the scenes, you would never create any of these objects directly. When you need a `Toolkit`, you ask for it with the static method `getDefaultToolkit()` or the `Component.getToolkit()` method.

You might use the `Toolkit` object if you need to fetch an image in an application (`getImage()`), get the font information provided with the `Toolkit` (`getFontList()` or `getFontMetrics()`), get the color model (`getColorModel()`), get the screen metrics (`getScreenResolution()` or `getScreenSize()`), get the system clipboard (`getSystemClipboard()`), get a print job (`getPrintJob()`), or ring the bell (`beep()`). The other methods of `Toolkit` are called for you by the system.

Toolkit Methods

Constructors

public Toolkit()--cannot be called by user

> Because `Toolkit` is an abstract class, it has no usable constructor. To get a `Toolkit` object, ask for your environment's default toolkit by calling the static method `getDefaultToolkit()` or call `Component.getToolkit()` to get the toolkit of a component. When the actual `Toolkit` is created for the native environment, the `awt` package is loaded, the `AWT-Win32` and `AWT--Callback-Win32` or `AWT-Motif` and `AWT-Input` threads (or the appropriate threads for your environment) are created, and the threads go into infinite loops for screen maintenance and event handling.

Pseudo-Constructors

public static synchronized Toolkit getDefaultToolkit ()

The `getDefaultToolkit()` method returns the system's default `Toolkit` object. The default `Toolkit` is identified by the `System` property `awt.toolkit`, which defaults to an instance of the `sun.awt.motif.MToolkit` class. On the Windows NT/95 platforms, this is overridden by the Java environment to be `sun.awt.win32.MToolkit` (Java1.0) or `sun.awt.windows.MToolkit` (Java1.1). On the Macintosh platform, this is overridden by the environment to be `sun.awt.macos.MToolkit`. Most browsers don't let you change the system property `awt.toolkit`. Since this is a static method, you don't need to have a `Toolkit` object to call it; just call `Toolkit.getDefaultToolkit()`.

Currently, only one `Toolkit` can be associated with an environment. You are more than welcome to try to replace the one provided with the JDK. This permits you to create a whole new widget set, outside of Java, while maintaining the standard AWT API.

System information

public abstract ColorModel getColorModel ()

The `getColorModel()` method returns the current `ColorModel` used by the system. The default `ColorModel` is the standard RGB model, with 8 bits for each of red, green, and blue. There are an additional 8 bits for the alpha component, for pixel-level transparency.

public abstract String[] getFontList ()

The `getFontList()` method returns a `String` array of the set Java fonts available with this `Toolkit`. Normally, these fonts will be understood on all the Java platforms. The set provided with Sun's JDK 1.0 (with Netscape Navigator and Internet Explorer, on platforms other than the Macintosh) contains TimesRoman, Dialog, Helvetica, Courier (the only fixed-width font), DialogInput, and ZapfDingbat.

In Java 1.1, `getFont()` reports all the 1.0 font names. It also reports Serif, which is equivalent to TimesRoman; San Serif, which is equivalent to Helvetica; and Monospaced, which is equivalent to Courier. The names TimesRoman, Helvetica, and Courier are still supported but should be avoided. They have been deprecated and may disappear in a future release. Although the JDK 1.1 reports the existence of the ZapfDingbat font, you can't use it. The characters in this font have been remapped to Unicode characters in the range `\u2700` to `\u27ff`.

public abstract FontMetrics getFontMetrics (Font font)

The `getFontMetrics()` method returns the `FontMetrics` for the given `Font` object. You can use this value to compute how much space would be required to display some text using this `font`. You can use this version of `getFontMetrics()` (unlike the similar method in the `Graphics` class) prior to drawing anything on the screen.

public int getMenuShortcutKeyMask() ★

The `getMenuShortcutKeyMask()` method identifies the accelerator key for menu shortcuts for the user's platform. The return value is one of the modifier masks in the `Event` class, like `Event.CTRL_MASK`. This method is used internally by the `MenuBar` class to help in handling menu selection events. See Chapter 10, *Would You Like to Choose from the Menu?* for more information about dealing with menu accelerators.

public abstract PrintJob getPrintJob (Frame frame, String jobtitle, Properties props) ★

The `getPrintJob()` method initiates a print operation, `PrintJob`, on the user's platform. After getting a `PrintJob` object, you can use it to print the current graphics context as follows:

```
// Java 1.1 only
PrintJob p = getToolkit().getPrintJob (aFrame, "hi", aProps);
Graphics pg = p.getGraphics();
printAll (pg);
pg.dispose();
p.end();
```

With somewhat more work, you can print arbitrary content. See Chapter 17, *Printing*, for more information about printing. The `frame` parameter serves as the parent to any print dialog window, `jobtitle` serves as the identification string in the print queue, and `props` serves as a means to provide platform-specific properties (default printer, page order, orientation, etc.). If `props` is `(Properties)null`, no properties will be used. `props` is particularly interesting in that it is used both for input and for output. When the environment creates a print dialog, it can read default values for printing options from the properties sheet and use that to initialize the dialog. After `getPrintJob()` returns, the properties sheet is filled in with the actual printing options that the user requested. You can then use these option settings as the defaults for subsequent print jobs.

The actual property names are `Toolkit` specific and may be defined by the environment outside of Java. Furthermore, the environment is free to ignore the `props` parameter altogether; this appears to be the case with Windows NT/95 platforms. (It is difficult to see how Windows NT/95 would use the properties sheet, since these platforms don't even raise the print dialog until you call the method `getGraphics()`.) Table 15.1 shows some of the properties recognized on UNIX platforms; valid property values are shown in a fixed-width font.

Table 15.1: UNIX Printing Properties

Property Name	**Meaning and Possible Values**
`awt.print.printer`	The name of the printer on your system to send the job to.
`awt.print.fileName`	The name of the file to save the print job to.
`awt.print.numCopies`	The number of copies to be printed.
`awt.print.options`	Other options to be used for the run-time system's print command.
`awt.print.destination`	Whether the print job should be sent to a `printer` or saved in a `file`.
`awt.print.paperSize`	The size of the paper on which you want to print--usually, `letter`.
`awt.print.orientation`	Whether the job should be printed in `portrait` or `landscape` orientation.

public static String getProperty (String key, String defaultValue) ★

The `getProperty()` method retrieves the `key` property from the system's *awt.properties* file (located in the *lib* directory under the *java.home* directory). If `key` is not a valid property, `defaultValue` is returned. This file is used to provide localized names for various system resources.

public abstract int getScreenResolution ()

The `getScreenResolution()` method retrieves the resolution of the screen in dots per inch. The sharper the resolution of the screen, the greater number of dots per inch. Values vary depending on the system and graphics mode.

The PrintJob.getPageResolution() method returns similar information for a printed page.

public abstract Dimension getScreenSize ()

The getScreenSize() method retrieves the dimensions of the user's screen in pixels for the current mode. For instance, a VGA system in standard mode will return 640 for the width and 480 for the height. This information is extremely helpful if you wish to manually size or position objects based upon the physical size of the user's screen. The PrintJob.getPageDimension() method returns similar information for a printed page.

public abstract Clipboard getSystemClipboard() ★

The getSystemClipboard() method returns a reference to the system's clipboard. The clipboard allows your Java programs to use cut and paste operations, either internally or as an interface between your program and objects outside of Java. For instance, the following code copies a String from a Java program to the system's clipboard:

```
// Java 1.1 only
Clipboard clipboard = getToolkit().getSystemClipboard();
StringSelection ss = new StringSelection("Hello");
clipboard.setContents(ss, this);
```

Once you have placed the string "Hello" on the clipboard, you can paste it anywhere. The details of Clipboard, StringSelection, and the rest of the java.awt.datatransfer package are described in Chapter 16, *Data Transfer*.

public final EventQueue getSystemEventQueue() ★

After checking whether the security manager allows access, this method returns a reference to the system's event queue.

protected abstract EventQueue getSystemEventQueueImpl() ★

getSystemEventQueueImpl() does the actual work of fetching the event queue. The toolkit provider implements this method; only subclasses of Toolkit can call it.

Images

The Toolkit provides a set of basic methods for working with images. These methods are similar to methods in the Applet class; Toolkit provides its own implementation for use by programs that don't have access to an AppletContext (i.e., applications or applets that are run as applications). Remember that you need an instance of Toolkit before you can call these methods; for example, to get an image, you might call Toolkit.getDefaultToolkit().getImage(`myImage.gif`).

public abstract Image getImage (String filename)

The getImage() method with a String parameter allows applications to get an image from the local filesystem. Its argument is either a relative or absolute filename for an image in a recognized image file format. The method returns immediately; the Image object that it returns is initially empty. When the image is needed, the system attempts to get filename and convert it to an image. To force the file to load immediately or to check for errors while loading, use the MediaTracker class.

NOTE:

This version of `getImage()` is not usable within browsers since it will throw a security exception because the applet is trying to access the local filesystem.

public abstract Image getImage (URL url)

The `getImage()` method with the `URL` parameter can be used in either applets or applications. It allows you to provide a URL for an image in a recognized image file format. Like the other `getImage()` methods, this method returns immediately; the `Image` object that it returns is initially empty. When the image is needed, the system attempts to load the file specified by `url` and convert it to an image. You can use the `MediaTracker` class to monitor loading and check whether any errors occurred.

public abstract boolean prepareImage (Image image, int width, int height, ImageObserver observer)

The `prepareImage()` method is called by the system or a program to force `image` to start loading. This method can be used to force an image to begin loading before it is actually needed. The `Image image` will be scaled to be `width` x `height`. A `width` and `height` of -1 means `image` will be rendered unscaled (i.e., at the size specified by the image itself). The `observer` is the `Component` on which `image` will be rendered. As the `image` is loaded across the network, the `observer`'s `imageUpdate()` method is called to inform the observer of the image's status.

public abstract int checkImage (Image image, int width, int height, ImageObserver observer)

The `checkImage()` method returns the status of the `image` that is being rendered on `observer`. Calling `checkImage()` only provides information about the image; it does not force the image to start loading. The `image` is being scaled to be `width` x `height`. Passing a `width` and `height` of -1 means the image will be displayed without scaling. The return value of `checkImage()` is some combination of `ImageObserver` flags describing the data that is now available. The `ImageObserver` flags are: `WIDTH`, `HEIGHT`, `PROPERTIES`, `SOMEBITS`, `FRAMEBITS`, `ALLBITS`, `ERROR`, and `ABORT`. Once `ALLBITS` is set, the image is completely loaded, and the return value of `checkImage()` will not change. For more information about these flags, see Chapter 12, *Image Processing*.

The following program loads an image; whenever `paint()` is called, it displays what information about that image is available. When the `ALLBITS` flag is set, `checkingImages` knows that the image is fully loaded, and that a call to `drawImage()` will display the entire image.

```
import java.awt.*;
import java.awt.image.*;
import java.applet.*;
public class checkingImages extends Applet {
    Image i;
    public void init () {
        i = getImage (getDocumentBase(), "ora-icon.gif");
    }
    public void displayChecks (int i) {
        if ((i & ImageObserver.WIDTH) != 0)
            System.out.print ("Width ");
        if ((i & ImageObserver.HEIGHT) != 0)
            System.out.print ("Height ");
        if ((i & ImageObserver.PROPERTIES) != 0)
            System.out.print ("Properties ");
        if ((i & ImageObserver.SOMEBITS) != 0)
```

```
            System.out.print ("Some-bits ");
        if ((i & ImageObserver.FRAMEBITS) != 0)
            System.out.print ("Frame-bits ");
        if ((i & ImageObserver.ALLBITS) != 0)
            System.out.print ("All-bits ");
        if ((i & ImageObserver.ERROR) != 0)
            System.out.print ("Error-loading ");
        if ((i & ImageObserver.ABORT) != 0)
            System.out.print ("Loading-Aborted ");
        System.out.println ();
    }
    public void paint (Graphics g) {
        displayChecks (Toolkit.getDefaultToolkit().checkImage(i, -1, -1, this));
        g.drawImage (i, 0, 0, this);
    }
}
```

Here's the output from running `checkingImages` under Java 1.0; it shows that the width and height of the image are loaded first, followed by the image properties and the image itself. Java 1.1 also displays `Frame-bits` once the image is loaded.

```
Width Height
Width Height Properties Some-bits
Width Height Properties Some-bits All-bits
Width Height Properties Some-bits All-bits
Width Height Properties Some-bits All-bits
... (Repeated Forever More)
```

public abstract Image createImage (ImageProducer producer)

This `createImage()` method creates an `Image` object from an `ImageProducer`. The `producer` parameter must be some class that implements the `ImageProducer` interface. Image producers in the `java.awt.graphics` package are `FilteredImageSource` (which, together with an `ImageFilter`, lets you modify an existing image) and `MemoryImageSource` (which lets you turn an array of pixel information into an image). The image filters provided with `java.awt.image` are `CropImageFilter`, `RGBImageFilter`, `AreaAveragingScaleFilter`, and `ReplicateScaleFilter`. You can also implement your own image producers and image filters. These classes are all covered in detail in Chapter 12, *Image Processing*.

The following code uses this version of `createImage()` to create a modified version of an original image:

```
Image i = Toolkit.getDefaultToolkit().getImage (u);
      TransparentImageFilter tf = new TransparentImageFilter (.5f);
Image j = Toolkit.getDefaultToolkit().createImage (
              new FilteredImageSource (i.getSource(), tf));
```

public Image createImage (byte[] imageData) ★

This `createImage()` method converts the entire byte array in `imageData` into an `Image`. This data must be in one of the formats understood by this AWT `Toolkit` (GIF, JPEG, or XBM) and relies on the "magic number" of the data to determine the image type.

public Image createImage (byte[] imageData, int offset, int length) ★

This `createImage()` method converts a subset of the byte data in `imageData` into an `Image`. Instead of starting at the beginning, this method starts at `offset` and goes to `offset+length-1`, for a total of `length` bytes. If `offset` is 0 and `length` is `imageData.length`, this method is equivalent to the previous method and converts the entire array.

The data in `imageData` must be in one of the formats understood by this AWT `Toolkit` (GIF, JPEG, or XBM) and relies on the "magic number" of the data to determine the image type.

NOTE:

For those unfamiliar with magic numbers, most data files are uniquely identified by the first handful or so of bytes. For instance, the first three bytes of a GIF file are "`GIF`". This is what `createImage()` relies upon to do its magic.

Miscellaneous methods

public abstract void beep () ★

The `beep()` method attempts to play an audio beep. You have no control over pitch, duration, or volume; it is like putting `echo ^G` in a UNIX shell script.

public abstract void sync ()

The `sync()` method flushes the display of the underlying graphics context. Normally, this is done automatically, but there are times (particularly when doing animation) when you need to `sync()` the display yourself.

PREVIOUS | HOME | NEXT
Audio in Applications | BOOK INDEX | The Peer Interfaces

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

16. Data Transfer

Contents:

One feature that was missing from Java 1.0 was the ability to access the system clipboard. It was impossible to cut and paste data from one program into another. Java 1.1 includes a package called `java.awt.datatransfer` that supports clipboard operations. Using this package, you can cut an arbitrary object from one program and paste it into another. In theory, you can cut and paste almost anything; in practice, you usually want to cut and paste text strings, so the package provides special support for string operations. The current version allows only one object to be on the clipboard at a time.

`java.awt.datatransfer` consists of three classes, two interfaces, and one exception. Objects that can be transferred implement the `Transferable` interface. The `Transferable` interface defines methods for working with different *flavors* of an object. The concept of flavors is basic to Java's clipboard model. Essentially, a flavor is a MIME content type. Any object can be represented in several different ways, each corresponding to a different MIME type. For example, a text string could be represented by a Java `String` object, an array of Unicode character data, or some kind of rich text that contains font information. The object putting the string on the clipboard provides whatever flavors it is capable of; an object pasting the string from the clipboard takes whatever flavor it can handle. Flavors are represented by the `DataFlavor` class, and the `UnsupportedFlavorException` is used when an object asks for a `DataFlavor` that is not available.

The `Clipboard` class represents the clipboard itself. There is a single system clipboard, but you can create as many private clipboards as you want. The system clipboard lets you cut and paste between arbitrary applications (for example, Microsoft Word and some Java programs). Private clipboards are useful within a single application, though you could probably figure out some way to export a clipboard to another application using RMI.

To put data on the clipboard, you must implement the `ClipboardOwner` interface, which provides a means for you to be notified when the data you write is removed from the clipboard. (There isn't any `ClipboardReader` interface; any object can read from the clipboard.) The final component of the `datatransfer` package is a special class called `StringSelection` that facilitates cutting and pasting text strings.

Cutting and pasting isn't the whole story; JavaSoft has also promised drag-and-drop capabilities, but this won't be in the initial release of Java 1.1.

16.1 DataFlavor

A `DataFlavor` represents a format in which data can be transferred. The `DataFlavor` class includes two common data flavors; you can create other flavors by extending this class. Flavors are essentially MIME content types and are represented by the standard MIME type strings. An additional content subtype has been added to represent Java classes; the content type of a Java object is:[1]

[1] The type name changed to `x-java-serialized-object` in the 1.1.1 release.

```
application/x-java-serialized-object
<classname>
```

For example, the content type of a `Vector` object would be:

```
application/x-java-serialized-object java.util.Vector
```

In addition to the content type, a `DataFlavor` also contains a *presentable name*. The presentable name is intended to be more comprehensible to humans than the MIME type. For example, the presentable name of a `VectorFlavor` object might just be "Vector", rather than the complex and lengthy MIME type given previously. Presentable names are useful when a program needs to ask the user which data flavor to use.

DataFlavor Methods

Variables

The `DataFlavor` class includes two public variables that hold "prebuilt" flavors representing different kinds of text objects. These flavors are used in conjunction with the `StringSelection` class.

Although these flavors are variables for all practical purposes, they are used as constants.

public static DataFlavor stringFlavor ★

The `stringFlavor` variable is the data flavor for textual data represented as a Java `String` object. Its MIME type is `application/x-javaserializedobject String`.

public static DataFlavor plainTextFlavor ★

The `plainTextFlavor` variable is the data flavor for standard, Unicode-encoded text. Its MIME type is `text/plain; charset=unicode`.

Constructors

The `DataFlavor` class has two constructors. One creates a `DataFlavor` given a MIME content type; the other creates a `DataFlavor` given a Java class and builds the MIME type from the class name.

public DataFlavor(String mimeType, String humanPresentableName) ★

The first constructor creates an instance of `DataFlavor` for the `mimeType` flavor of data. The `humanPresentableName` parameter should be a more user-friendly name. It might be used in a menu to let the user select a flavor from several possibilities. It might also be used to generate an error message when the `UnsupportedFlavorException` occurs. The `plainTextFlavor` uses "Plain Text" as its presentable name.

To read data from the clipboard, a program calls the `Transferable.getTransferData()` method. If the data is represented by a `DataFlavor` that doesn't correspond to a Java class (for example, `plainTextFlavor`), `getTransferData()` returns an `InputStream` for you to read the data from.

public DataFlavor(Class representationClass, String humanPresentableName) ★

The other constructor creates an instance of `DataFlavor` for the specific Java class `representationClass`. Again, the `humanPresentableName` provides a more user-friendly name for use in menus, error messages, or other interactions with users. The `stringFlavor` uses "Unicode String" as its presentable name.

A program calls `Transferable.getTransferData()` to read data from the clipboard. If the data is represented by a Java class, `getTransferData()` returns an instance of the representation class itself. It does not return a `Class` object. For example, if the data flavor is `stringFlavor`, `getTransferData()` returns a `String`.

Presentations

public String getHumanPresentableName() ★

The `getHumanPresentableName()` method returns the data flavor's presentable name; for example, `stringFlavor.getHumanPresentableName()` returns the string "Unicode String".

public void setHumanPresentableName(String humanPresentableName) ★

The `setHumanPresentableName()` method changes the data flavor's presentable name to a new `humanPresentableName`. It is hard to imagine why you would want to change a flavor's name.

public String getMimeType() ★

The `getMimeType()` method gets the MIME content type for the `DataFlavor` as a `String`.

public Class getRepresentationClass() ★

The `getRepresentationClass()` method returns the Java type that is used to represent data of this flavor (i.e., the type that would be returned by the `getTransferData()`method). It returns the type as a `Class` object, not an instance of the class itself. Note that all data flavors have a representation class, not just those for which the class is specified explicitly in the constructor. For example, the `plainTextFlavor.getRepresentationClass()` method returns the class `java.io.StringReader`.

public boolean isMimeTypeEqual(String mimeType) ★

The `isMimeTypeEqual()` method checks for string equality between `mimeType` and the data flavor's MIME type string. For some MIME types, this comparison may be too simplistic because character sets may not be present on types like `text/plain`. Therefore, this method would tell you that the MIME type `text/plain; charset=unicode` is different from `text/plain`.

public final boolean isMimeTypeEqual(DataFlavor dataFlavor) ★

The `isMimeTypeEqual()` method checks whether the MIME type of the `dataFlavor` parameter equals the current data flavor's MIME type. It calls the previous method, and therefore has the same weaknesses.

Protected methods

protected String normalizeMimeType(String mimeType) ★

The `normalizeMimeType()` method is used to convert a MIME type string into a standard form. Its argument is a MIME type, as a `String`; it returns the new normalized MIME type. You would never call `normalizeMimeType()` directly, but you might want to override this method if you are creating a subclass of `DataFlavor` and want to change the default normalization process. For example, one thing you might do with this is add the string `charset=US-ASCII` to the `text/plain` MIME type if it appears without a character set.

protected String normalizeMimeTypeParameter(String parameterName, String parameterValue) ★

The `normalizeMimeTypeParameter()` method is used to convert any parameters associated with MIME types into a standard form. Its arguments are a parameter name (for example, `charset`) and the parameter's value (for example, `unicode`). It returns `parameterValue` normalized. You would never call `normalizeMimeTypeParameter()` directly, but you might want to override this method if you are creating a subclass of `DataFlavor` and want to change the default normalization process. For example, parameter values may be case sensitive. You could write a method that would convert the value `Unicode` to the more appropriate form `unicode`.

While it may be more trouble than it's worth, carefully overriding these normalization methods might help you to get more predictable results from methods like `isMimeTypeEqual()`.

Miscellaneous methods

public boolean equals(DataFlavor dataFlavor) ★

The `equals()` method defines equality for flavors. Two `DataFlavor` objects are equal if their MIME type and representation class are equal.

PREVIOUS

NEXT

18. java.applet Reference

Contents:

18.1 Introduction to the Reference Chapters

The preceding seventeen chapters cover just about all there is to know about AWT. We have tried to organize them logically, and provide all the information that you would expect in a reference manual--plus much more in the way of examples and practical information about how to do things effectively. However, there are many times when you just need a reference book, pure and simple: one that's organized alphabetically, and where you can find any method if you know the class and package that it belongs to, without having to second guess the author's organizational approach. That's what the rest of this book provides. It's designed to help you if you need to look something up quickly, and find a brief but accurate summary of what it does. In these sections, the emphasis is on *brief*; if you want a longer description, look in the body of the book.

The reference sections describe the following packages:

- `java.applet` (Chapter 18, *java.applet Reference*)
- `java.awt` (Chapter 19, *java.awt Reference*)
- `java.awt.datatransfer` (Chapter 20, *java.awt.datatransfer Reference*)

- `java.awt.event` (Chapter 21, *java.awt.event Reference*)
- `java.awt.image` (Chapter 22, *java.awt.image Reference*)
- `java.awt.peer` (Chapter 23, *java.awt.peer Reference*)

Within each package, classes and interfaces are listed alphabetically. There is a description and a pseudo-code definition for each class or interface. Each variable and method is listed and described. New Java 1.1 classes are marked with a black star (★), as are new methods and new variables. Of course, if a class is new, all its methods are new. We didn't mark individual methods in new classes. Methods that are deprecated in Java 1.1 are marked with a white star (☆).

Inheritance presents a significant problem with documenting object-oriented libraries, because the bulk of a class's methods tend to be hiding in the superclasses. Even if you're very familiar with object-oriented software development, when you're trying to look up a method under the pressure of some deadline, it's easy to forget that you need to look at the superclasses in addition to the class you're interested in itself. Nowhere is this problem worse than in AWT, where some classes (in particular, components and containers) inherit well over 100 methods, and provide few methods of their own. For example, the `Button` class contains seven public methods, none of which happens to be `setFont()`. The font used to display a button's label is certainly settable--but to find it, you have to look in the superclass `Component`.

So far, we haven't found a way around this problem. The description of each class has an abbreviated class hierarchy diagram, showing superclasses (all the way back to Object), immediate subclasses, and the interfaces that the class implements. Ideally, it would be nice to have a list of all the inherited methods--and in other parts of Java, that's possible. For AWT, the lists would be longer than the rest of this book, much too long to be practical, or even genuinely useful. Someday, electronic documentation may be able to solve this problem, but we're not there yet.

PREVIOUS | HOME | NEXT
Printing Arbitrary Content | BOOK INDEX | Package diagrams

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

19. java.awt Reference

Contents:

Graphics

GridBagConstraints

GridBagLayout

GridLayout

IllegalComponentStateException ★

Image

Insets

ItemSelectable ★

Label

LayoutManager

LayoutManager2 ★

List

MediaTracker

Menu

MenuBar

MenuComponent

MenuContainer

MenuItem

MenuShortcut ★

Panel

Point

Polygon

PopupMenu ★

PrintGraphics ★

PrintJob ★

Rectangle

ScrollPane ★

Scrollbar

Shape ★

SystemColor ★

TextArea

TextComponent

TextField

Toolkit

Window

AWTError

Name

AWTError

[Graphic: Figure from the text]

Description

An `AWTError`; thrown to indicate a serious runtime error.

Class Definition

```
public class java.awt.AWTError
    extends java.lang.Error {

  // Constructors
  public AWTError (String message);
}
```

Constructors

AWTError

public AWTError (String message)

Parameters

message

Detail message

See Also

`Error`, `String`

PREVIOUS | HOME | NEXT

Window | BOOK INDEX | AWTEvent ★

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

20. java.awt.datatransfer Reference

Contents:

Clipboard ★

Name

Clipboard ★

[Graphic: Figure from the text]

Description

The `Clipboard` class is a repository for a `Transferable` object and can be used for cut, copy, and paste operations. The system clipboard can be accessed by calling `Toolkit.getDefaultToolkit().getSystemClipboard()`. You can use this technique if you are interested in exchanging data between your application and other applications (Java or non-Java) running on the system. In addition, `Clipboard` can be instantiated directly, if "private" clipboards are needed.

Class Definition

```
public class java.awt.datatransfer.Clipboard
   extends java.lang.Object {
  // Variables
  protected Transferable contents;
  protected ClipboardOwner owner;
  // Constructors
  public Clipboard (String name);
  // Instance Methods
  public synchronized Transferable getContents (Object requestor);
  public String getName();
  public synchronized void setContents (Transferable contents, ClipboardOwner owner);
}
```

Variables

contents

protected Transferable contents

The object that the `Clipboard` contains, i.e., the object that has been cut or copied.

owner

protected ClipboardOwner owner

The object that owns the `contents`. When something else is placed on the clipboard, owner is notified via lostOwnership().

Constructors

Clipboard

public Clipboard (String name)

Parameters

> *name*
>
> > The name for this `Clipboard`.

Description

> Constructs a `Clipboard` object with the given `name`.

Instance Methods

getContents

public synchronized Transferable getContents (Object requestor)

Parameters

> *requestor*

The object asking for the contents.

Returns

An object that implements the `Transferable` interface.

Description

Returns the current contents of the `Clipboard`. You could use this method to paste data from the clipboard into your application.

getName

public String getName()

Returns

`Clipboard`'s name.

Description

Returns the name used when this clipboard was constructed. `Toolkit.getSystemClipboard()` returns a `Clipboard` named "System".

setContents

public synchronized void setContents (Transferable contents, ClipboardOwner owner)

Parameters

contents

New contents.

owner

Owner of the new contents.

Description

Changes the contents of the `Clipboard`. You could use this method to cut or copy data from your application to the clipboard.

See Also

`ClipboardOwner`, `Toolkit`, `Transferable`

PREVIOUS HOME NEXT
Window BOOK INDEX ClipboardOwner ★

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

21. java.awt.event Reference

Contents:

[TextEvent](#) ★

[TextListener](#) ★

[WindowAdapter](#) ★

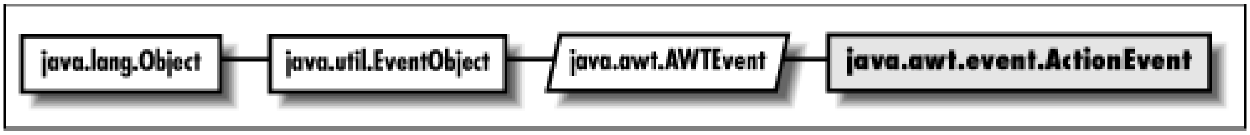
[WindowEvent](#) ★

[WindowListener](#) ★

ActionEvent ★

Name

ActionEvent ★



Description

Action events are fired off when the user performs an action on a component, such as pushing a button, double-clicking on a list item, or selecting a menu item. There is only one action event type, `ACTION_PERFORMED`.

Class Definition

```
public class java.awt.event.ActionEvent
   extends java.awt.AWTEvent {
  // Constants
  public final static int ACTION_FIRST;
  public final static int ACTION_LAST;
  public final static int ACTION_PERFORMED;
  public final static int ALT_MASK;
  public final static int CTRL_MASK;
  public final static int META_MASK;
  public final static int SHIFT_MASK;
  // Constructors
  public ActionEvent (Object source, int id, String command);
  public ActionEvent (Object source, int id, String command, int modifiers);
  // Instance Methods
  public String getActionCommand();
  public int getModifiers();
  public String paramString();
}
```

Constants

ACTION_FIRST

public final static int ACTION_FIRST

Specifies the beginning range of action event ID values.

ACTION_LAST

public final static int ACTION_LAST

Specifies the ending range of action event ID values.

ACTION_PERFORMED

public final static int ACTION_PERFORMED

The only action event type; it indicates that the user has performed an action.

ALT_MASK

public final static int ALT_MASK

A constant representing the ALT key. ORed with other masks to form `modifiers` setting of an `AWTEvent`.

CTRL_MASK

public final static int CTRL_MASK

A constant representing the Control key. ORed with other masks to form `modifiers` setting of an `AWTEvent`.

META_MASK

public final static int META_MASK

A constant representing the META key. ORed with other masks to form `modifiers` setting of an `AWTEvent`.

SHIFT_MASK

public final static int SHIFT_MASK

A constant representing the Shift key. ORed with other masks to form `modifiers` setting of an `AWTEvent`.

Constructors

ActionEvent

public ActionEvent (Object source, int id, String command)

Parameters

source

The object that generated the event.

id

The type ID of the event.

command

The action command string.

Description

Constructs an `ActionEvent` with the given characteristics.

public ActionEvent (Object source, int id, String command, int modifiers)

Parameters

source

The object that generated the event.

id

The type ID of the event.

command

The action command string.

modifiers

A combination of the key mask constants.

Description

Constructs an `ActionEvent` with the given characteristics.

Instance Methods

getActionCommand

public String getActionCommand()

Returns

The action command string for this `ActionEvent`.

Description

Generally the action command string is the label of the component that generated the event. Also, when localization is necessary, the action command string can provide a setting that does not get localized.

getModifiers

public int getModifiers()

Returns

A combination of the key mask constants.

Description

Returns the modifier keys that were held down when this action was performed. This enables you to perform special processing if, for example, the user holds down Shift while pushing a button.

paramString

public String paramString()

Returns

String with current settings of `ActionEvent`.

Overrides

`AWTEvent.paramString()`

Description

Helper method for `toString()` to generate string of current settings.

See Also

`ActionListener`, `AWTEvent`, `String`

PREVIOUS | HOME | NEXT

WindowListener ★ | BOOK INDEX | ActionListener ★

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

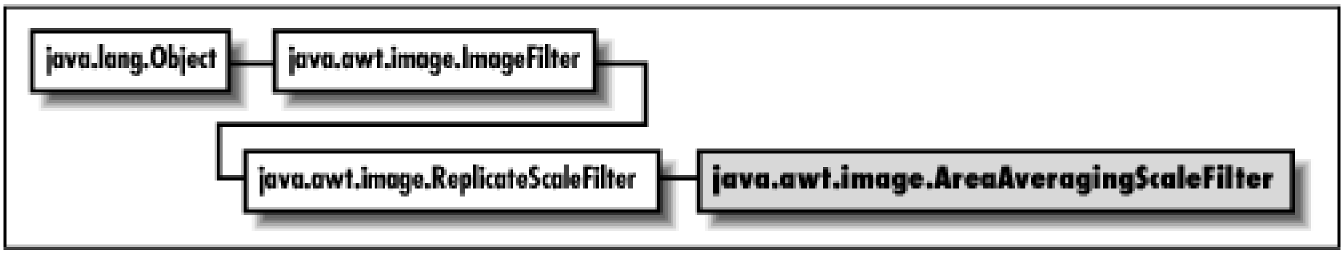
22. java.awt.image Reference

Contents:

AreaAveragingScaleFilter ★

Name

AreaAveragingScaleFilter ★


java.lang.Object
java.awt.image.ImageFilter
java.awt.image.ReplicateScaleFilter
java.awt.image.AreaAveragingScaleFilter

Description

The `AreaAveragingScaleFilter` class scales an image using a simple smoothing algorithm.

Class Definition

```
public class java.awt.image.AreaAveragingScaleFilter
    extends java.awt.image.ReplicateScaleFilter {
  // Constructor
  public AreaAveragingScaleFilter (int width, int height);
  // Instance Methods
  public void setHints (int hints);
  public void setPixels (int x, int y, int w, int h, ColorModel model,
    byte[] pixels, int off, int scansize);
  public void setPixels (int x, int y, int w, int h, ColorModel model,
    int[] pixels, int off, int scansize);
}
```

Constructor

AreaAveragingScaleFilter

public AreaAveragingScaleFilter (int width, int height)

Parameters

width

Width of scaled image.

height

Height of scaled image.

Description

Constructs an `AverageScaleFilter` that scales the original image to the specified size.

Instance Methods

setHints

public void setHints (int hints)

Parameters

hints

Flags indicating how data will be delivered.

Overrides

`ImageFilter.setHints(int)`

Description

Gives this filter hints about how data will be delivered.

setPixels

public void setPixels (int x, int y, int w, int h, ColorModel model, byte[] pixels, int off, int scansize)

Parameters

x

x-coordinate of top-left corner of pixel data delivered with this method call.

y

y-coordinate of top-left corner of pixel data delivered with this method call.

w

Width of the rectangle of pixel data delivered with this method call.

h

Height of the rectangle of pixel data delivered with this method call.

model

Color model of image data.

pixels

Image data.

off

Offset from beginning of the pixels array.

scansize

Size of each line of data in pixels array.

Overrides

`ReplicateScaleFilter.setPixels(int, int, int, int, ColorModel, byte[], int, int)`

Description

Receives a rectangle of image data from the `ImageProducer`; scales these pixels and delivers them to any `ImageConsumer`s.

public void setPixels (int x, int y, int w, int h, ColorModel model, int[] pixels, int off, int scansize)

Parameters

x

x-coordinate of top-left corner of pixel data delivered with this method call.

y

y-coordinate of top-left corner of pixel data delivered with this method call.

w

Width of the rectangle of pixel data delivered with this method call.

h

Height of the rectangle of pixel data delivered with this method call.

model

Color model of image data.

pixels

Image data.

off

Offset from beginning of the pixels array.

scansize

Size of each line of data in pixels array.

Overrides

`ReplicateScaleFilter.setPixels(int, int, int, int, ColorModel, int[], int, int)`

Description

Receives a rectangle of image data from the `ImageProducer`; scales these pixels and delivers them to any `ImageConsumer`s.

See Also

ColorModel, ReplicateScaleFilter

PREVIOUS | HOME | NEXT
WindowListener ★ | BOOK INDEX | ColorModel

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

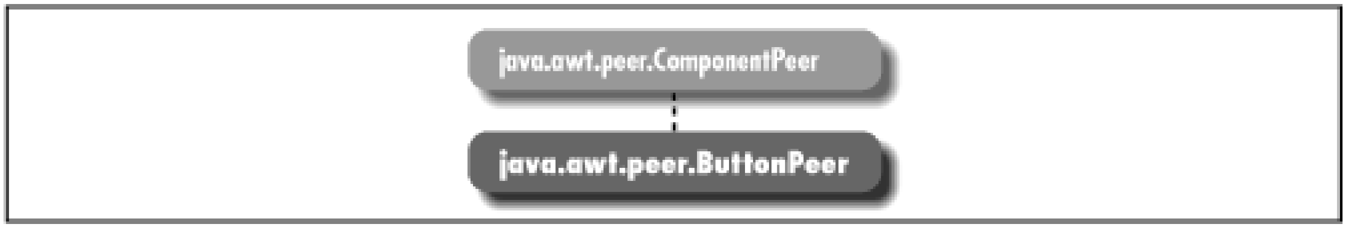
23. java.awt.peer Reference

Contents:

ButtonPeer

Name

ButtonPeer



Description

`ButtonPeer` is an interface that defines the basis for buttons.

Interface Definition

```
public abstract interface java.awt.peer.ButtonPeer
extends java.awt.peer.ComponentPeer {
  // Interface Methods
  public abstract void setLabel (String label);
}
```

Interface Methods

setLabel

public abstract void setLabel (String label)

Parameters

label

New text for label of button's peer.

Description

Changes the text of the label of button's peer.

See Also

`ComponentPeer`, `String`

PREVIOUS
WindowPeer

HOME

BOOK INDEX

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

B. HTML Markup For Applets

Contents:

B.1 The Applet Tag

The introduction of Java created the need for additional HTML tags. In the alpha release of Java, the HotJava browser used the `<APP>` tag to include applets within HTML files. However, `<APP>` was unacceptable to the standards committee because it could have an infinite number of parameters. It was replaced by the `<APPLET>` tag, used in conjunction with the `<PARAM>` tag. Apparently, the standards folks did not like the `<APPLET>` tag either, so you can expect it to be replaced eventually, although at this point, there is no agreement about its successor, and it is highly unlikely that any production browser would stop supporting `<APPLET>`.

The syntax of the `<APPLET>` tag is shown below; the order of the parameters does not matter:

```
<APPLET
    [ALIGN = alignment]
    [ALT = alternate-text]
    CODE = applet-filename or OBJECT = serialized-applet
    [CODEBASE = applet-directory-url]
    [ARCHIVE = filename.zip/filename.jar]
    HEIGHT = applet-pixel-height
    [HSPACE = horizontal-pixel-margin]
    [MAYSCRIPT = true/false]
    [NAME = applet-name]
    [VSPACE = vertical-pixel-margin]
    WIDTH = applet-pixel-width
>
<PARAM NAME=parameter1 VALUE=value1>
```

```
<PARAM NAME=parameter2 VALUE=value2>
<PARAM NAME=parameter3 VALUE=value3>
...
[alternate-html]
</APPLET>
```

`<APPLET>`

The `<APPLET>` tag specifies where and how to display an applet within the HTML document. If the browser does not understand the `<APPLET>` and `<PARAM>` tags, it displays the `alternate-html`. (It displays the `alternate-html` because it doesn't understand the surrounding tags and ignores them. There's no magic to the `alternate-html` itself.) If a browser does understand `<APPLET>` but cannot run Java (for example, a browser on Windows 3.1) or Java has been disabled, the browser displays the `alternate-html` or the `alternate-text` specified by the optional `ALT` parameter. The `CODE`, `WIDTH`, and `HEIGHT` parameters are required. Parameters within the `<APPLET>` tag are separated by spaces, not by commas.

`</APPLET>`

Closes the `<APPLET>` tag. Anything prior to `</APPLET>` is considered `alternate-html` if it is not a `<PARAM>` tag. The `alternate-html` is displayed when Java is disabled, when Java cannot be run in the current browser, or when the browser does not understand the `<APPLET>` tag.

The following parameters may appear inside the `<APPLET>` tag.

`ALIGN`

`alignment`, optional. Specifies the applet's alignment on the Web page. Valid values are: `left`, `right`, `top`, `texttop`, `middle`, `absmiddle`, `baseline`, `bottom`, `absbottom`. Default: `left`. The alignment values have the same meanings as they do in the `<IMG>` tag.

`ALT`

`alternate-text`, optional. The alternate text is displayed when the browser understands the `<APPLET>` tag but is incapable of executing applets, either because Java is disabled or not supported on the platform. Support of this tag is browser dependent; most browsers just display the `alternate-html` since that is not restricted to text.

`ARCHIVE`

`filename.zip/filename.jar`, optional. Points to a comma-separated list of uncompressed ZIP or JAR files that contain one or more Java classes. Each file is downloaded once to the user's disk and searched for the class named in the CODE parameter, and any helper classes required to execute that class. JAR files may be signed to grant additional access. (JAR files are Java archives, a new archive format defined in Java 1.1. JAR files support features like digital signatures and compression. While they are not yet in wide use, they should become an important way of distributing sets of Java classes.)

CODE

applet-filename. This parameter or the OBJECT parameter is required. Name of applet *.class* file. The *.class* extension is not required in the <APPLET> tag but is required in the class's actual filename. The filename has to be a quoted string only if it includes whitespace.

CODEBASE

applet-directory-url, optional. Relative or absolute URL specifying the directory in which to locate the *.class* file or ZIP archive for the applet. Default: html directory.

HEIGHT

applet-pixel-height, required. Initial height of applet in pixels. Many browsers do not allow applets to change their height.

HSPACE

horizontal-pixel-margin, optional. Horizontal margin left and right of the applet, in pixels.

MAYSCRIPT

Required for applets that wish to use LiveConnect and the netscape.javascript classes to interact with JavaScript. Set to true to communicate with JavaScript. Set to false, or omit this parameter to disable communication with JavaScript. Both Java and JavaScript must be enabled in the browser.

NAME

applet-name, optional. Allows simultaneously running applets to communicate by this name. Default: the applet's class name.

OBJECT

serialized-applet. This parameter or the CODE parameter is required. Name of applet saved to a file as a serialized object. When loaded, `init()` is not called again but `start()` is. Parameters for running the applet are taken from this `<APPLET>` tag, not the original.

`VSPACE`

`vertical-pixel-margin`, optional. Vertical margin above and below the applet, in pixels.

`WIDTH`

`applet-pixel-width`, required. Initial width of applet in pixels. Many browsers do not allow applets to change their width.

The `<PARAM>` tag may appear between the `<APPLET>` and `</APPLET>` tags:

`<PARAM>`

The `<PARAM>` tag allows the HTML author to provide run-time parameters to the applet as a series of `NAME` and `VALUE` pairs. The `NAME` is case insensitive, a `String`. See Chapter 14, *And Then There Were Applets* for a discussion of how to read parameters in an applet. Quotes are required around the parameter name or its value if there are any embedded spaces. There can be an infinite number of `<PARAM>` tags, and they all must appear between `<APPLET>` and `</APPLET>`

The special parameter name `CABBASE` is used for sending CAB files with Internet Explorer 3.0. CAB files are similar to ZIP files but are compressed into a CABinet file and can store audio and image files, in addition to classes. (For a full explanation see: http://207.68.137.43/workshop/java/overview.htm.) When *.class* files are placed within a CAB file, they are decompressed at the local end. Here's an example:

```
<APPLET CODE="oreilly.class" WIDTH=400 HEIGHT=400>
<PARAM NAME="cabbase" VALUE="ora.cab>
</APPLET>
```

The special parameter name `ARCHIVES` is reserved for sending JAR files. JAR files can also be specified using the `ARCHIVES` parameter to the `<APPLET>` tag.[1] Here's an example:

[1] For a full explanation see http://www.javasoft.com/products/JDK/1.1/docs/guide/jar/index.html.

```
<APPLET CODE="oreilly.class" WIDTH=400 HEIGHT=400>
<PARAM NAME="archives" VALUE="ora.jar>
```

```
</APPLET>
```

PREVIOUS | HOME | NEXT
Resource Bundles | BOOK INDEX | Platform-Specific Event Handling

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

D. Image Loading

Contents:

D.1 How Images are Loaded

You have seen how easy it is to display an image on screen and have probably guessed that there's more going on behind the scenes. The `getImage()` and `drawImage()` methods trigger a series of events that result in the image being available for display on the `ImageObserver`. The image is fetched asynchronously in another thread. The entire process[1] goes as follows:

> [1] This summary covers Sun's implementation (JDK). Implementations that don't derive from the JDK may behave completely differently.

1. The call to `getImage()` triggers `Toolkit` to call `createImage()` for the image's `InputStreamImageSource` (which is a `URLImageSource` in this case; it would be a `FileImageSource` if we were loading the image from a local file).

2. The `Toolkit` registers the image as being "desired." Desired just means that something will eventually want the image loaded. The system then waits until an `ImageObserver` registers its interest in the image.

3. The `drawImage()` method (use of `MediaTracker` or `prepareImage()`) registers an `ImageObserver` as interested.

4. Registering an `ImageObserver` kicks the image's `ImageRepresentation` into action; this is the start of the loading process, although image data isn't actually transferred until step 9. `ImageRepresentation` implements the `ImageConsumer` interface.

5. The start of production registers the image source (`ImageProducer` `URLImageSource`) with the `ImageFetcher` and also registers the `ImageRepresentation` as an `ImageConsumer` for the image.

6. The `ImageFetcher` creates a thread to get the image from its source.

7. The `ImageFetcher` reads data and passes it along to the `InputStreamImageSource`, which is a `URLImageSource`.

8. The `URLImageSource` determines that `JPEGImageDecoder` is the proper `ImageDecoder` for converting the input stream into an `Image`. (Other `ImageDecoders` are used for other image types, like GIF.)

9. The `ImageProducer` starts reading the image data from the source; it calls the `ImageConsumer` (i.e., the `ImageRepresentation`) as it processes the image. The most important method in the `ImageConsumer` interface is `setPixels()`, which delivers pixel data to the consumer for rendering onscreen.

10. As the `ImageConsumer` (i.e., the `ImageRepresentation`) gets additional information, it notifies the `ImageObserver` via `imageUpdate()` calls.

11. When the image is fully acquired across the network, the thread started by the `ImageFetcher` stops.

As you see, there are a lot of unfamiliar moving pieces. Many of them are from the `java.awt.image` package and are discussed in Chapter 12, *Image Processing*. Others are from the `sun.awt.image` package; they are hidden in that you don't need to know anything about them to do image processing in Java. However, if you're curious, we'll briefly summarize these classes in the next section.

PREVIOUS | HOME | NEXT
Test Program | BOOK INDEX | A Brief Tour of sun.awt.image

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Index

A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X

A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X

HOME

1. Introduction

Contents:

Java is a relatively new programming language. However, many of the features that make up the language are not new at all. Java's designers borrowed features from a variety of older languages, such as Smalltalk and Lisp, in order to achieve their design goals.

Java is designed to be both robust and secure, so that it can be used to write small, hosted programs, or *applets*, that can be run safely by hosting programs such as Web browsers and cellular phones. Java also needs to be portable, so that these programs can run on many different kinds of systems. What follows is a list of the important features that Java's designers included to create a robust, secure, and portable language.

- Java is a simple language. It borrows most of its syntax from C/C++, so it is easy for C/C++ programmers to understand the syntax of Java code. But that is where the similarities end. Java does not support troublesome features from C/C++, so it is much simpler than either of those languages. In fact, if you examine the features of Java, you'll see that it has more in common with languages like Smalltalk and Lisp.

- Java is a statically typed language, like C/C++. This means that the Java compiler can perform static type checking and enforce a number of usage rules.

- Java is fully runtime-typed as well. The Java runtime system keeps track of all the objects in the system, which makes it possible to determine their types at runtime. For example, casts from one object type to another are verified at runtime. Runtime typing also makes it possible to use completely new, dynamically loaded objects with some amount of type safety.

- Java is a late-binding language, like Smalltalk, which means that it binds method calls to their definitions at runtime. Runtime binding is essential for an object-oriented language, where a subclass can override methods in its superclass, and only the runtime system can determine which method should be invoked. However, Java also supports the performance benefits of early binding. When the compiler can determine that a method cannot be overridden by subclassing, the method definition is bound to the method call at compile-time.

- Java takes care of memory management for applications, which is unlike C/C++, where the programmer is responsible for explicit memory management. Java supports the dynamic allocation of arrays and objects, and then takes care of reclaiming the storage for objects and arrays when it is safe to do so, using a technique called *garbage collection*. This eliminates one of the largest sources of bugs in C/C++ programs.

- Java supports object references, which are like pointers in C/C++. However, Java does not allow any manipulation of references. For example, there is no way that a programmer can explicitly dereference a reference or use pointer arithmetic. Java implicitly handles dereferencing references, which means that they can be used to do most of the legitimate things that C/C++ pointers can do.

- Java uses a single-inheritance class model, rather than the error-prone multiple-inheritance model used by C++. Instead, Java provides a feature called an *interface* (borrowed from Objective C) that specifies the behavior of an object without defining its implementation. Java supports multiple inheritance of interfaces, which provides many of the benefits of multiple inheritance, without the associated problems.

- Java has support for multiple threads of execution built into the language, so there are mechanisms for thread synchronization and explicit waiting and signaling between threads.

- Java has a powerful exception-handling mechanism, somewhat like that in newer implementations of C++. Exception handling provides a way to separate error-handling code from normal code, which leads to cleaner, more robust applications.

- Java is both a compiled and an interpreted language. Java code is compiled to Java byte-codes, which are then executed by a Java runtime environment, called the Java virtual machine. The specifications of the Java language and the virtual machine are fully defined; there are no implementation-dependent details. This architecture makes Java an extremely portable language.

- Java uses a three-layer security model to protect a system from untrusted Java code. The byte-code verifier reads byte-codes before they are run and makes sure that they obey the basic rules of the Java language. The class loader takes care of bringing compiled Java classes into the runtime interpreter. The security manager handles application-level security, by controlling whether or not a program can access resources like the filesystem, network ports, external processes, and the windowing system.

As you can see, Java has quite a list of interesting features. If you are a C/C++ programmer, many of the constructs of the Java language that are covered in this book should look familiar to you. Just be warned that you shouldn't take all of these constructs at face value, since many of them are different in Java than they are in C/C++.

1.1 A "Hello World" Program

Before diving into the various constructs provided by the Java language, you should have at least a general understanding of the Java programming environment. In the fine tradition of all language reference manuals, here is a short Java program that outputs "Hello world!" and then exits:

```
/*
 * Sample program to print "Hello World"
 */
class HelloWorld {             // Declare class HelloWorld
    public static void main(String argv[]) {
        System.out.println("Hello World!");
    }
}
```

This example begins with a comment that starts with `/*` and ends with `*/`. This type of comment is called a *C-style comment*. The example also uses another kind of comment that begins with `//` and ends at the end of the line. This kind of comment is called a *single-line comment*; it is identical to that style of comment in C++. Java supports a third type of comment, called a *documentation comment*, that provides for the extraction of comment text into a machine-generated document.

Comments aside, the example consists of a single class declaration for the class called `HelloWorld`. If you are unfamiliar with classes, you can think of a class as a collection of variables and pieces of executable code called *methods* for the purposes of this discussion. In Java, most executable code is part of a method. Methods are identical to virtual member functions in C++, except that they can exist only as part of a class. Methods are also similar to functions, procedures, and subroutines in other programming languages.

The `HelloWorld` class contains a single method named `main()`. When you ask the Java interpreter to run a Java program, you tell it what code to run by giving it the name of a class. The Java interpreter then loads the class and searches it for a method named `main()` that has the same attributes and parameters as shown in the example. The interpreter then calls that `main()` method.

In the declaration of `main()`, the name `main` is preceded by the three keywords: `public`, `static`, and `void`. The `public` modifier makes the `main()` method accessible from any class. The `static` modifier, when applied to a method, means that the method can be called independently of an instance of a class. The `void` keyword means that the method returns no value. The `main()` method of an application should always be declared with these three keywords. Although the meanings of these keywords is similar to their meanings in C++, there are some differences in the meaning of the keyword `static` as used in Java and C++.

The `main()` method contains a single line of executable code that calls the `println()` method of the object `System.out`. Passing the argument `"Hello World!"` to the `println()` method results in "Hello World!" being output. `System.out` is an object that encapsulates an application's standard output. It is similar in purpose to `stdout` in C and `cout` in C++. Java also has `System.in` and `System.err` objects that are similar in purpose to `stdin` and `stderr` in C and `cin` and `cerr` in C++, respectively.

2. Lexical Analysis

When the Java compiler compiles a program, the first thing it does is determine the structure of the program. The compiler reads the characters in the program source and then applies rules to recognize progressively larger chunks of the file, such as identifiers, expressions, statements, and classes. The process of discovering the organization of the program is divided into two components:

- The *lexical analyzer*. This component looks for sequences of characters called *tokens* that form identifiers, literals, operators, and the like.
- The *parser*. This component is responsible for discovering higher levels of organization in the sequences of tokens discovered by lexical analysis.

This chapter describes the rules governing the lexical analysis of Java programs. The rules governing the parsing of Java programs are described over the course of subsequent chapters.

The lexical analysis rules for Java can appear slightly ambiguous. Where ambiguity occurs, the rules for interpreting character sequences specify that conflicts are resolved in favor of the interpretation that matches the most characters. That's a bit confusing, so an example should help. Take the character sequence:

```
+++
```

The ambiguity is that the sequence could potentially be interpreted as either + followed by ++ or the other way around; both are valid tokens. But according to the lexical analysis rules that insist that tokenization favor the longest character match, Java interprets the character sequence as:

```
++ +
```

Because ++ is longer than +, Java first recognizes the token ++ and then the +.

These rules can produce undesired results when character sequences are not separated by white space. For example, the following sequence is ambiguous:

```
x++y
```

The programmer probably intended this sequence to mean "`x + (+y)`", but the lexical analyzer always produces the token sequence "`x ++ y`". This sequence is syntactically incorrect.

Java lexical analysis consists of two phases: pre-processing and tokenization. The pre-processing phase is discussed in the following section. The tokenization phase is responsible for recognizing the tokens in the pre-processed input and is discussed later in this chapter.

2.1 Pre-Processing

A Java program is a sequence of characters. These characters are represented using 16-bit numeric codes defined by the Unicode standard.[1] Unicode is a 16-bit character encoding standard that includes representations for all of the characters needed to write all major natural languages, as well as special symbols for mathematics. Unicode defines the codes 0 through 127 to be consistent with ASCII. Because of that consistency, Java programs can be written in ASCII without any need for programmers to be aware of Unicode.

> [1] Unicode is defined by an organization called the Unicode Consortium. The defining document for Unicode is *The Unicode Standard, Version 2.0* (published by Addison-Wesley, ISBN 0-201-48345-9). More recent information about Unicode is available at *http://unicode.org/*.

Java is based on Unicode to allow Java programs to be useful in as many parts of the world as possible. Internally, Java programs store characters as 16-bit Unicode characters. The benefits of using Unicode are currently difficult to realize, however, because most operating environments do not support Unicode. And those environments that do support Unicode generally do not include fonts that cover more than a small subset of the Unicode character set.

Since most operating environments do not support Unicode, Java uses a pre-processing phase to make sure that all of the characters of a program are in Unicode. This pre-processing comprises two steps:

- Translate the program source into Unicode characters if it is in an encoding other than Unicode. Java defines escape sequences that allow all characters that can be represented in Unicode to be represented in other character encodings, such as ASCII or EBCDIC. The escape sequences are recognized by the compiler, even if the program is already represented in Unicode.

- Divide the stream of Unicode characters into lines.

Conversion to Unicode

The first thing a Java compiler does is translate its input from the source character encoding (e.g., ASCII or EBCDIC) into Unicode. During the conversion process, Java translates escape sequences of the form `\u` followed by four hexadecimal digits into the Unicode characters indicated by the given hexadecimal values. These escape sequences let you represent Unicode characters in whatever character set you are using for your source code, even if it is not Unicode. For example, `\u0000` is a way of representing the NUL character.

More formally, the compiler input is converted from a stream of *EscapedSourceCharacters* into a stream of Unicode characters. *EscapedSourceCharacter* is defined as:

[Graphic: Figure from the text]

HexDigit is either a *Digit* or one of the following letters: `A`, `a`, `B`, `b`, `C`, `c`, `D`, `d`, `E`, `e`, `F`, or `f`.

A *Digit* is one of the following characters: `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, or `9`.

As you can see, the definition of *EscapedSourceCharacter* specifies that the `` `u' `` in the escape sequence can occur multiple times. Multiple occurrences have the same meaning as a single occurrence of `` `u' ``.

If the program source is already in Unicode, this conversion step is still performed in order to process these `\u` escapes.

The Java language specification recommends, but does not require, that the classes that come with Java use the `\u`*xxxx* escapes when called upon to display a character that would not otherwise be displayable.

Division of the Input Stream into Lines

The second step of pre-processing is responsible for recognizing sequences of characters that terminate lines. The character sequence that indicates the end of a line varies with the operating environment. By recognizing end-of-line character sequences during pre-processing, Java makes sure that subsequent compilation steps do not need to be concerned with multiple representations for the end of a line.

In this step, the lexical analyzer recognizes the combinations of carriage return (`\u000D`) and line feed (`\u000A`) characters that are in widespread use as end-of-line indicators:

[Graphic: Figure from the text]

As always, ambiguities in lexical rules are resolved by matching the longest possible sequence of characters. That means that the sequence of a carriage return character followed by a linefeed character is always recognized as a one-line terminator, never as two.

PREVIOUS HOME NEXT
Notational Conventions BOOK INDEX Tokenization

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

3. Data Types

Contents:

A *data type* defines the set of values that an expression can produce or a variable can contain. The data type of a variable or expression also defines the operations that can be performed on the variable or expression. The type of a variable is established by the variable's declaration, while the type of an expression is determined by the definitions of its operators and the types of their operands.

Conceptually, there are two types of data in Java programs: primitive types and reference types. The primitive types are self-contained values that can be contained in a variable. The primitive types are comprised of integer types, floating-point types, and the `boolean` type. Of these, the integer types and floating-point types are considered arithmetic types, since arithmetic can be performed on them. Reference types contain values that point to or identify arrays or objects. The syntax for specifying a type is:

[Graphic: Figure from the text]

References Arithmetic Types; Boolean Type; Floating-point types; Integer types; Interface method return type; Interface Variables; Local Variables; Method return type; Primitive Types; Reference Types; Variables

3.1 Primitive Types

A primitive data type represents a single value, such as a number, a character, or a Boolean value. Java has primitive types for arithmetic and Boolean data:

[Graphic: Figure from the text]

References Arithmetic Types; Boolean Type

Arithmetic Types

Unlike in C/C++, all of the arithmetic data types in Java are specified to have representations that are independent of the particular computer running a Java program. This guarantees that numeric computations made by Java programs produce the same results on all platforms.

There are two kinds of arithmetic types: integer and floating-point.

The integer types are: `byte`, `short`, `int`, `long`, and `char`. Like C/C++, character data is considered an integer type because of its representation and because arithmetic operations can be performed on `char` data. Unlike C/C++, however, `short int` and `long int` are not valid data types in Java. In addition, `signed` and `unsigned` do not have any special meaning in Java.

The floating-point data types are `float` and `double`.

The formal definition of an arithmetic type is:

[Graphic: Figure from the text]

References Integer types; Floating-point types

Integer types

Java provides integer data types in a variety of sizes. Unlike C/C++, however, the sizes of these types are part of the language specification; they are not platform-dependent. Formally:

[Graphic: Figure from the text]

The values represented by these types are specified in Table 3-1. The representation shown is used on all platforms and is independent of the native platform architecture.

Table 3.1: Integer Types and Their Representations

Type	**Representation**	**Range**
`byte`	8-bit, signed, two's complement	-128 to 127
`short`	16-bit, signed, two's complement	-32768 to 32767
`int`	32-bit, signed, two's complement	-2147483648 to 2147483647
`long`	64-bit, signed, two's complement	-9223372036854775808 to 9223372036854775807
`char`	16-bit, unsigned, Unicode	`'\u0000'` to `'\uffff'`

All of the signed integer types in Java use a two's complement representation. Two's complement is a binary encoding for integers, which has the following properties:

- The leftmost bit is the sign bit. If the sign bit is 1, the number is negative.
- Positive numbers have the usual binary representation.
- Negating a number involves complementing all of the bits in the number and then adding 1 to the result.
- The most negative value does not have a positive equivalent.

The `java.lang` package includes the `Byte`, `Short`, `Integer`, `Long`, and `Character` classes. These classes provide object wrappers for `byte`, `short`, `int`, `long`, and `char` values, respectively. Each of these classes defines `static` `MIN_VALUE` and `MAX_VALUE` variables for its minimum and maximum values.

Java performs all integer arithmetic using `int` or `long` operations. A value that is of type `byte`, `short`, or `char` is widened to an `int` or a `long` before the arithmetic operation is performed.

A value of any integer type can be cast (i.e., converted) to a value of any other integer type. Integer types, however, cannot be cast to a `boolean` value, nor can the `boolean` type be cast to an integer-type value. A value of a signed integer type can be assigned to a value of the same or wider type without a cast. In this case, the value is automatically widened to the appropriate type. Table 3-2 shows whether an assignment from a particular integer type to another integer type can be done directly or if it requires a type cast.

Table 3.2: Assignment Compatibility Between Integer Types

To/From	**`byte`**	**`char`**	**`short`**	**`int`**	**`long`**
`byte`	Assignable	Cast needed	Cast needed	Cast needed	Cast needed
`char`	Cast needed	Assignable	Cast needed	Cast needed	Cast needed
`short`	Assignable	Cast needed	Assignable	Cast needed	Cast needed
`int`	Assignable	Assignable	Assignable	Assignable	Cast needed
`long`	Assignable	Assignable	Assignable	Assignable	Assignable

The principle underlying the above table is that assignments that do not lose information do not require a type cast. Assigning a `short` value to an `int` without a cast is allowed because all of the values that can be represented by a `short` can also be represented by `int`. However, assigning an `int` value to a `short` is not allowed without a cast because it involves going from a 32-bit signed quantity to a 16-bit signed quantity. Similarly, a `byte` value cannot be assigned to `char` without a cast. `byte` is an 8-bit signed quantity, so it can represent negative numbers. However, `char` is a 16-bit unsigned quantity, so it cannot represent negative numbers.

Java provides the following kinds of operators for integer values:

- Comparison operators
- Arithmetic operators
- Increment and decrement operators
- Bitwise logical operators

If all of the operands of an operator are of an integer type, the operation is performed as an integer operation. Normally, integer operations are performed with a precision of 32 bits. If at least one of the operands of an integer operation is a `long`, however, the operation is performed with a precision of 64 bits.

When an integer operation overflows or underflows, there is no indication given that the overflow or underflow occurred.

If the right-hand operand (the divisor) of a division or remainder operation is 0, Java throws an `ArithmeticException`. Division by zero is the only circumstance that can cause an integer operation to throw an exception.

References Additive Operators; Assignment Operators; Bitwise/Logical Operators; Byte; Character; Conditional Operator; Equality Comparison Operators; Increment/Decrement Operators; Integer; Integer literals; Long; Multiplicative Operators; Relational Comparison Operators; Runtime exceptions; Shift Operators; Short; Unary Operators

Floating-point types

Like C/C++, Java provides two sizes of floating-point numbers: single precision and double precision.

Formally:

[Graphic: Figure from the text]

Java uses the single precision 32-bit IEEE 754 format to represent `float` data and the double precision 64-bit IEEE 754 format to represent `double` data.[1] These representations are used on all platforms, whether or not there is native support for the formats. The values represented by these types are shown in Table 3-3.

> [1] The IEEE 754 floating-point data representation and operations on it are defined in *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std. 754-1985 (IEEE, New York). The standard can be ordered by calling (908) 981-0060 or writing to IEEE, 445 Hoes Lane, PO Box 1331, Piscataway, NJ 08855-1331, USA.

Table 3.3: Floating-Point Types and Their Representations

Type	**Representation**	**Range**
`float`	32-bit, IEEE 754	1.40239846e-45 to 3.40282347e+38
`double`	64-bit, IEEE 754	4.94065645841246544e-324 to 1.79769313486231570e+308

Normally, non-zero `float` values are represented as:

```
sign*mantissa*2^exponent
```

where `sign` is +1 or -1, `mantissa` is a positive integer less than 2^24, and `exponent` is an integer in the inclusive range -149 to 104.

Non-zero `double` values are represented as:

```
sign*mantissa*2^exponent
```

where `sign` is +1 or -1, `mantissa` is a positive integer less than 2^53, and `exponent` is an integer in the inclusive range -1045 to 1000.

In addition, the IEEE 754 standard defines three special values:

Positive infinity

> This value is produced when a `float` or `double` operation overflows, or a positive value is divided by zero. Positive infinity is by definition greater than any other `float` or `double` value.

Negative infinity

> This value is produced when a `float` or `double` operation overflows, or a negative value is divided by zero. Negative infinity is by definition less than any other `float` or `double` value.

Not-a-number (NaN)

> This value is produced by the `float` and `double` operations such as the division of zero by zero. When NaN is one of the operands for an operation, most arithmetic operations return NaN as the result. Since NaN is unordered, most comparison operators (e.g., `<`, `<=`, `==`, `>=`, `>`) return `false` when one of their arguments is NaN. The exception is `!=`, which returns `true` when one of its arguments is NaN.

The `java.lang` package includes `Float` and `Double` classes that provide object wrappers for `float` and `double` values. Each class defines the three special values as symbolic constants: `POSITIVE_INFINITY`, `NEGATIVE_INFINITY`, and `NaN`. Each class also defines `MIN_VALUE` and `MAX_VALUE` constants for its minimum and maximum values.

Floating-point operations never throw exceptions. Operations that overflow produce positive or negative infinity. Operations that underflow produce positive or negative zero. Operations that have no defined result produce not-a-number.

Both `float` and `double` data types have distinct representations for positive and negative zero. These values compare as equal `(0.0 == -0.0)`. Positive and negative zero do produce different results for some arithmetic operations, however: `1.0/0.0` produces positive infinity, while `1.0/-0.0` produces negative infinity.

A `float` value can be assigned to a `double` variable without using a type cast, but assigning a `double` value to a `float` variable does require a cast. Conversion from a `float` or `double` value to any other data type also requires a cast. Either of the floating-point data types can be cast to any other arithmetic type, but they cannot be cast to `boolean`. When a floating-point number is cast to an integer type, it is truncated (i.e., rounded toward zero).

Java provides the following kinds of operators for floating-point values:

- Comparison operators
- Arithmetic operators
- Increment and decrement operators

If any of the arguments of an operation are of a floating-point type, the operation is performed as a floating-point operation. In other words, any of the integer operands are converted to floating point before the operation takes place. Floating-point operations are normally performed with a precision of 32 bits. However, if at least one of the operands of the operation is a `double`, the operation is performed with a precision of 64 bits.

References Additive Operators; Assignment Operators; Conditional Operator; Equality Comparison Operators; Double; Float; Floating-point literals; Increment/Decrement Operators; Multiplicative Operators; Relational Comparison Operators; Unary Operators

Boolean Type

The `boolean` data type represents two values: `true` and `false`. These values are keywords in Java. The `java.lang` package includes a `Boolean` class that provides an object wrapper for `boolean` values. This `Boolean` class defines the constant objects `Boolean.TRUE` and `Boolean.FALSE`.

Java provides the following kinds of operators for `boolean` values:

- Equality and inequality operators
- Boolean logical operators

The following Java constructs require a `boolean` value to specify a condition:

- `if`
- `while`
- `for`
- `do`
- The conditional operator `? :`

Unlike C/C++, any attempt to substitute a different type for `boolean` in these constructs is treated as an error by Java.

No other data type can be cast to or from `boolean`. In particular, using the integer 1 to represent `true` and 0 to represent `false` does not work in Java. Though Java does not provide conversions between `boolean` and other types, it is possible to provide explicit logic to accomplish the same thing:

```
int i;
i != 0             // This is true if i is not equal to zero
boolean b;
b ? 1 : 0          // If b is true produce 1; otherwise 0
```

References Boolean; Bitwise/Logical Operators; Boolean literals; Boolean Negation Operator !; Boolean Operators; Conditional Operator; Equality Comparison Operators; The do Statement; The for Statement; The if Statement; The while Statement

PREVIOUS | HOME | NEXT
Tokenization | BOOK INDEX | Reference Types

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

4. Expressions

Contents:

Expressions in Java are used to fetch, compute, and store values. To fetch a value, you use a type of expression called a primary expression. To compute and store values, you use the various operators described in this chapter. In Java, expressions are most often used in methods and constructors; they can also appear in field variable initializers and static initializers.

Most expressions, when they are evaluated, produce values that can be used by other expressions. The one exception is an expression that calls a method declared with the `void` return type. An expression that invokes a `void` method cannot be embedded in another expression. The evaluation of an expression can also produce side effects, such as variable assignments and increment and decrement operations.

The value produced by an expression can be either a pure value or a variable or array element. The distinction is that a *pure value* cannot be used to store a value, while a variable or array element can.[1] An expression that produces a variable or an array element can be used anywhere that an expression that produces a pure value can be used.

This chapter refers to values as being either pure values or variables. Saying that a value is a pure value means that it is a value such as 24 that contains information, but cannot be used as the target of an assignment expression. Saying that a value is a variable means that it is something like `var` or `ar[i]` that can be used as the target of an assignment. The generic term "value" is used to mean either a variable or a pure value.

The formal definition of an expression is:

[Graphic: Figure from the text]

The above diagram may seem deceptively simple; why is such a definition even necessary? Expressions in Java are defined with a number of mutually recursive railroad diagrams. You can think of the *Expression* definition as being both the lowest-level definition and the highest-level definition of these mutually recursive diagrams. In other words, `a=b[i]+c[i]` is an expression, as are `b[i]`, `c[i]`, `a`, `b`, `c`, and `i`. This first diagram defines an expression to be an *AssignmentExpression*, which is the final definition used to describe Java expressions.

References Assignment Operators

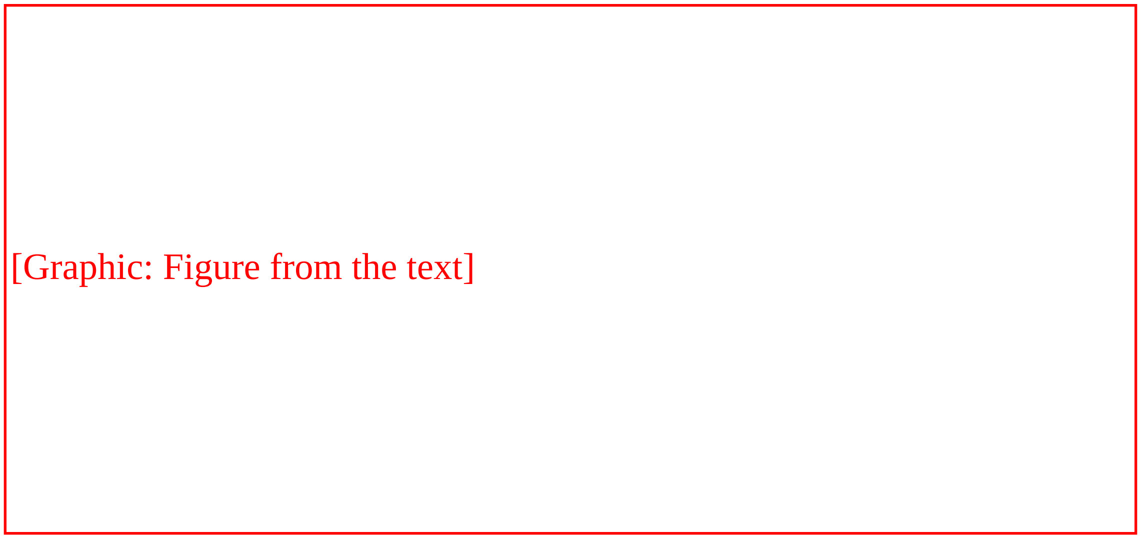
4.1 Primary Expressions

A primary expression is the most elementary type of expression. Primary expressions are constructs that fetch or create values, but do not directly perform computations on them:

[Graphic: Figure from the text]

Terms are those primary expressions that produce values, by doing such things as accessing fields in classes, invoking methods, and accessing array elements:

[1] Note that Java's distinction between pure values and variable and array elements is similar to the distinction in C and C++ between rvalues and lvalues.



References Allocation Expressions; *Expression* 4; *FieldExpression* 4.1.6; Identifiers; Index Expressions; Literals; Method Call Expression; Class Literals

this

The keyword `this` can be used only in the body of a constructor or an instance method (i.e., a method that is not declared `static`), or in the initializer for an instance variable. A `static` method is not associated with an object, so `this` makes no sense in such a method and is treated as an undefined variable. If `this` is used in an inappropriate place, a compile-time error occurs.

The value produced by `this` is a reference to the object associated with the expression that is being evaluated. The type of the primary expression `this` is a reference to the class in which it appears.

One common use for `this` is to allow access to a field variable when there is a local variable with the same name. For example:

```
int foo;
void setFoo(int foo) {
    this.foo = foo;
}
```

Another common usage is to implement a callback mechanism. Passing `this` to another object allows that object to call methods in the object associated with the calling code. For example, to allow an object inside of an applet to be able to access parameters passed to the applet in the HTML `<applet>` tag, you can use code like the following:

```
public class MyApplet extends Applet {
    ...
    Foo foo;
    public void init() {
        foo = new Foo(this);
    ...
    }
}
class Foo {
    Applet app;
    ...
    Foo(Applet app) {
        this.app = app;
    }
    ...
    void doIt() {
        String dir = app.getParameter("direction");
        ...
    }
    ...
}
```

Another use for the keyword `this` is in a special kind of *FieldExpression* that refers to an enclosing instance of this object. A reference to an enclosing instance is written as the class name of the enclosing instance followed by a dot and the keyword `this` (as described in 5.3.7.2 Member classes). Consider the following code:

```
public class ImageButton extends Canvas {
    ...
    private class MyImage extends Image {
        Image fileImage;
        MyImage(String fileName) throws IOException {
            URL url = new URL(fileName);
            ImageProducer src = (ImageProducer)url.getContent();
            Image fileImage = createImage(src);
            prepareImage(this, ImageButton.this);
        }
    ...
```

The call to `prepareImage()` takes two arguments. The first argument is a reference to `this` instance of the `MyImage` class. The second argument is a reference to this object's enclosing instance, which is an instance of the `ImageButton` class.

References Constructors; Constructor implementation; *FieldExpression* 4.1.6; Inner Classes; Methods

super

The keyword `super` can be used only in the body of a constructor or an instance method (i.e., a method that is not declared `static`), or in the initializer for an instance variable. In addition, `super` cannot appear in the class `Object` because `Object` has no superclass. If `super` is used in an inappropriate place, a compile-time error occurs.

In most cases, the primary expression `super` has a value that is equivalent to casting the value of the primary expression `this` to the superclass of the class in which it appears. In other words, `super` causes the object to be treated as an instance of its superclass. The type of the primary expression `super` is a reference to the superclass of the class in which it appears.

There are two situations in which `super` produces a result that is different than what would be produced by casting `this` to its superclass:

- When `super` is used to explicitly call a constructor in the superclass from a constructor in the class, the field variables for the class are initialized when the superclass's constructor returns.

- If a class contains a method that overrides a method declared in its superclass, calling the method by casting `this` to the superclass results in a call to the overriding method. However, calling the method with the special reference provided by `super` calls the overridden method in the superclass.

The main purpose of `super` is to allow the behavior of methods and constructors to be extended, rather than having to be totally replaced. Consider the following example:

```
class A {
    public int foo(int x) {
        return x*x;
    }
    public int bar(int x) {
        return x*8;
    }
}
class B extends A{
    public int foo(int x) {
        return super.foo(x)+x;
    }
    public int bar(int x){
        return x*5;
    }
}
```

The `foo()` method in class `B` extends the behavior of the `foo()` method in class `A` by calling that method and performing further computations on its result. On the other hand, the `bar()` method in class `B` totally replaces the behavior of the `bar()` method in class `A`.

References Constructors; Constructor implementation; Methods

null

The primary expression `null` produces a special object reference value that does not refer to any object, but is assignment-compatible with all object reference types.

An operation on an object reference that does not attempt to access the referenced object works the same way for `null` as it does for other object reference values. For example:

```
foo == null
```

However, any operation that attempts to access an object through a `null` reference throws a `NullPointerException`. The one exception to this rule is the string concatenation operator (`+`), which converts a `null` operand to the string literal `"null"`.

References Runtime exceptions; String Concatenation Operator +

Literal Expressions

A primary expression that consists of a literal produces a pure value. The data type of this pure value is the data type of the literal.

References Literals

Parenthetical Expressions

A primary expression that consists of a parenthesized expression produces the same pure value as the expression in parentheses. The data type of this pure value is also the same as the data type of the expression in parentheses. A parenthetical expression can only produce a pure value. Thus, the following code produces an error:

```
(x) = 5;          // Illegal
```

References *Expression* 4

Field Expressions

A *field expression* is a primary expression that fetches such things as local variables, formal parameters, and field variables. A field expression can evaluate to a pure value or a variable. The data type of a field expression is the data type of the pure value, variable, or array element produced by the following expression.

[Graphic: Figure from the text]

[Graphic: Figure from the text]

Essentially, a field expression can be a simple identifier, a primary expression followed by an identifier, or a class or interface name followed by an identifier. Here's an example of a field expression that consists of a simple *Identifier* :

```
myVar
```

Before the Java compiler can decide what to do with a lone identifier such as this, it must first match it with a declaration. The compiler first looks in the method where the identifier appears for a local variable or formal parameter with the same name as the identifier. If the compiler finds a matching local variable or formal parameter, the field expression produces the matching variable or parameter.

If the identifier does not match a local variable or a formal parameter, it is expected to match the name of a field variable in the class in which it occurs.

If the matching variable is declared `final`, the field expression produces the pure value specified by the variable's initializer. Otherwise, the field expression produces the matching field variable. If the method that the identifier appears in is declared `static`, the matching variable must also be declared `static` or the compiler declares an error.

A lone identifier that matches the name of a field variable is equivalent to:

```
this.Identifier
```

This form of a field expression can be used to access a field variable that is shadowed by a local variable or a formal parameter. For example:

```
class Shadow {
    int value;
    Shadow(int value) {
        this.value=value;
    }
}
```

In the above example, the identifier `value` is used as both the name of a field variable and the name of a formal parameter in the constructor. Within the constructor, the unqualified identifier `value` refers to the formal parameter, not to the field variable. In order to access the field variable, you have to qualify `value` with `this`.

In addition to allowing an object to refer to itself, the keyword `this` has another use in field expressions. The construct *ClassOrInterfaceName*`.this` identifies the enclosing instance of an object that is an instance of an inner class.[2] Consider the following example:

> [2] Since this construct fetches an object reference, you might expect it to be a primary expression. However, due to the way in which inner classes are implemented, this construct is actually a field expression.

```
public class ImageButton extends Canvas {
    ...
    private class MyImage extends Image {
        Image fileImage;
        MyImage(String fileName) throws IOException {
            URL url = new URL(fileName);
            ImageProducer src = (ImageProducer)url.getContent();
            Image fileImage = createImage(src);
            prepareImage(this, ImageButton.this);
        }
    ...
```

The call to `prepareImage()` takes two arguments. The first argument is a reference to `this` instance of the `MyImage` class. The second argument is a reference to this object's enclosing instance, which is an instance of the `ImageButton` class.

Here are some examples of field expressions that consist of a *PrimaryExpression* and an *Identifier*:

```
this.myVar
size().height
```

```
(new Foo()).bar
```

A primary expression that appears at the beginning of a field expression must produce a reference to an object. The identifier to the right of the dot must be the name of a field variable in the object referred to by the primary expression. If, at runtime, the primary expression in a field expression produces the value `null`, a `NullPointerException` is thrown.

Here's an example of a field expression that consists of a *ClassOrInterfaceName* and an *Identifier*:

```
Double.POSITIVE_INFINITY
```

A field expression that begins with *ClassOrInterfaceName* produces a field variable of the specified class. If the field variable is not declared `static`, the specified class must either be the same class in which the field expression appears or a superclass of that class.

Such a field expression is approximately equivalent to:

```
((ClassOrInterfaceName)this).Identifier
```

If *ClassOrInterfaceName* specifies a class or interface defined in a different package than the package in which the field expression appears, the class name must be qualified by the package in which the class is defined. For example:

```
java.awt.Event.MOUSE_UP
```

However, if an import statement imports the specified class, the package name is not necessary.

ClassOrInterfaceName can refer to an inner class or a nested top-level class or interface by qualifying the name of the class or interface with the name of the enclosing class. For example, consider the following declaration:

```
public class A {
    public class B {
    }
}
```

Based on this declaration, you can create a new instance of `B` as follows:

```
new A.B()
```

Most field expressions produce variables when they are evaluated. This means that the field expression can be used as the left operand of an assignment operator. A field expression produces a pure value, rather than a variable, if the identifier at the end of the field expression is a field variable that is declared `final`. Such a field expression returns a pure value because the value of a `final` variable cannot be modified. A field expression that produces a pure value cannot be the left operand of an assignment operator, or the operand of the `++` or `- -` operators. Here's an erroneous example:

```
final int four=4
four++
```

This is equivalent to:

```
4++
```

As such, it causes the Java compiler to issue an error message.

When the Java compiler detects an expression that uses the value of a local variable that may not have been initialized, it issues an error message. For example:

```
{
    int x;
    if (testForSomething())
        x = 4;
    System.out.println(x);    // Compiler complains
}
```

The compiler complains about the use of `x` in the `println()` method because `x` may not have been given an explicit value when the program reaches that statement. Even though there is an assignment to `x` in the preceding statement, the compiler recognizes that the assignment may not have been performed, since it is enclosed within a conditional statement. The Java language specification requires that a compiler issue an error message when it detects an uninitialized local variable.

References Identifiers; Inheritance; Inner Classes; Interface Variables; Local Variables; Packages; Primary Expressions; Runtime exceptions; Variables

Index Expressions

An *index expression* is a primary expression that produces an array element when it is evaluated. The value produced by an index expression is a variable; it can be used as the left operand of an assignment expression. The data type of an index expression is the data type of the array element produced by the expression:

[Graphic: Figure from the text]

When the compiler evaluates an index expression, the term to the left of the square brackets is evaluated before the expression inside of the square brackets. If the term produces the value `null`, a `NullPointerException` is thrown.

Array indexing uses an `int`-valued expression between the square brackets. If the type of the expression is `byte`, `char`, or `short`, the value is automatically converted to `int`. An array index cannot be of type `long`. The value of the array index must be in the range zero to one less than the length of the array. An array object detects an out-of-range index value and throws an `ArrayIndexOutOfBoundsException`.

Because of the precedence of Java expressions, an array allocation expression can only be indexed when the expression is enclosed in parentheses. For example:

```
(new int[6])[3]
```

This expression refers to the fourth element of the newly created array. Leaving out the parentheses results in the following:

```
new int[6][3]
```

This is not an index expression, but an array allocation expression that allocates an array of 3 arrays of 6 integers.

References Array Allocation Expressions; Array Types; *Expression* 4; *Term* 4.1; Exceptions

Method Call Expression

A *method call expression* is a primary expression that invokes a method:

[Graphic: Figure from the text]

[Graphic: Figure from the text]

A method call expression produces the pure value returned by the method; the type of this value is specified by the return type in the method declaration. But if the method has the return type `void`, the expression does not produce a value.

The *PrimaryExpression*, if present, is evaluated first. Then expressions provided as method arguments are evaluated from left to right. Finally, the method is invoked.

When a method call is made to a method that is not `static`, the call is made through an object reference:

- If the method call expression does not contain a *PrimaryExpression* or *ClassOrInterfaceName* before the method name, the method call is made implicitly through the object referenced by the keyword `this`. This form of a method call expression is treated as if it were written:

  ```
  this.Identifier(...)
  ```

- If the method call expression contains a *PrimaryExpression* before the method name, the call is made through the object reference produced by the *PrimaryExpression*.

- If the method call expression contains a *ClassOrInterfaceName* before the method name, then the specified class must either be the same class in which the method call expression appears or a superclass of that class. In this case, the method call is made through the object referenced by the keyword `this`. This form of a method call expression is treated as if it were written:

  ```
  ((ClassOrInterfaceName)this).Identifier(...)
  ```

When a method call is made to a `static` method, the call is made through a class or interface type:

- If the method call expression does not contain a *PrimaryExpression* or *ClassOrInterfaceName* before the method name, the method call is made implicitly through the class that contains the call.

- If the method call expression contains a *PrimaryExpression* before the method name, the call is made through the class of the object reference produced by the *PrimaryExpression*.

- If the method call expression contains a *ClassOrInterfaceName* before the method name, the method call is made through the specified class or interface type.

The rules for supplying actual values for the formal parameters of a method are similar to the rules for assignment. A particular value can be specified as the actual value of a formal parameter if and only if it is assignment-compatible with the type of the formal parameter. You can use a type cast to make a value assignment compatible with a formal parameter.

The process that the Java compiler uses to select the actual method that will be invoked at runtime is rather involved. The compiler begins by finding any methods that have the specified name. If the method call has been made through an object reference, the compiler searches in the class of that object reference. If the call has been made through a specific class or interface name, the compiler searches in that class or interface. The compiler searches all of the methods defined in the particular class or interface, as well as any methods that are inherited from superclasses or super-interfaces. At this point, the compiler is searching for both `static` and non-`static` methods, since it does not know which type of method is being called.

If the compiler finds more than one method, that means the method is overloaded. Consider this example:

```
public class MyMath {
    public int square(int x) { return x*x; }
    public long square(long x) { return x*x; }
    public float square(float x) { return x*x; }
    public double square(double x) { return x*x; }
    public double hypotenuse(double x, double y) {
        return Math.sqrt(x*x + y*y);
    }
}
```

In the above example, the `square()` method is overloaded, while `hypotenuse()` is not.

If the method is overloaded, the compiler then determines which of the methods has formal parameters that are compatible with the given arguments. If more than one method is compatible with the given arguments, the method that most closely matches the given parameters is selected. If the compiler cannot select one of the methods as a better match than the others, the method selection process fails and the compiler issues an error message. Note that the return types of overloaded methods play no part in selecting which method is to be invoked.

After the compiler successfully selects the method that most closely matches the specified arguments, it knows the name and signature of the method that will be invoked at runtime. It does not, however, know for certain what class that method will come from. Although the compiler may have selected a method from `MyMath`, it is possible that a subclass of `MyMath` could define a method that has the same name and the same number and types of parameters as the selected method. In this case, the method in the subclass overrides the method in `MyMath`. The compiler cannot know about overriding methods, so it generates runtime code that dynamically selects the appropriate method.

Here are the details of the three-step method selection process:

Step One

> The method definitions are searched for methods that, taken in isolation, could be called by the method call expression. If the method call expression uses an object reference, the search takes place in the class of that object reference. If the expression uses a specific class or interface name, the search takes place in that class or interface. The search includes all of the methods defined in the particular class or interface, as well as any methods inherited from superclasses or super-interfaces. The search also includes both `static` and non-`static` methods.
>
> A method is a candidate if it meets the following criteria:
>
> - The name of the method is the same as the name specified in the method call expression.
> - The method is accessible to the method call expression, based on any access modifiers specified in the method's declaration.
> - The number of formal parameters declared for the method is the same as the number of actual arguments provided in the method call expression.
> - The data type of each actual parameter is assignment-compatible with the corresponding formal parameter.
>
> Consider the following expression that calls a method defined in the preceding example:
>
> ```
> MyMath m;
> m.square(3.4F)
> ```
>
> Here is how the Java compiler uses the above criteria to decide which method the expression actually calls:
>
> - The name `square` matches four methods defined in the `MyMath` class, so the compiler must decide which one of those methods to invoke.
> - All four methods are declared `public`, so they are all accessible to the above expression and are thus all still viable candidates.

- The method call expression provides one argument. Since the four methods under consideration each take one argument, there are still four possible choices.

- The method call expression is passing a `float` argument. Because a `float` value cannot be assigned to an `int` or a `long` variable, the compiler can eliminate the versions of `square()` that take these types of arguments. That still leaves two possible methods for the above expression: the version of `square()` that takes a `float` argument and the one that takes a `double` argument.

Step Two

If more than one method meets the criteria in Step One, the compiler tries to determine if one method is a more specific match than the others. If there is no method that matches more specifically, the selection process fails and the compiler issues an error message.

Given two methods, `A()` and `B()`, that are both candidates to be invoked by the same method call expression, `A()` is more specific than `B()` if:

- The class in which the method `A()` is declared is the same class or a subclass of the class in which the method `B()` is declared.

- Each parameter of `A()` is assignment-compatible with the corresponding parameter of `B()`.

Let's go back to our previous example. We concluded by narrowing the possible methods that the expression `m.square(3.4F)` might match to the methods in `MyMath` named `square()` that take either a `float` or a `double` argument. Using the criteria of this step, we can further narrow the possibilities. These methods are declared in the same class, but the version of `square()` that takes a `float` value is more specific than the one that takes a `double` value. It is more specific because a `float` value can be assigned to a `double` variable, but a `double` value cannot be assigned to a `float` variable without a type cast.

There are some cases in which it is not possible to choose one method that is more specific than others. When this happens, the Java compiler treats the situation as an error and issues an appropriate error message.

For example, consider a situation where the compiler needs to choose between two methods declared as follows:

```
double foo(float x, double y)
double foo(double x, float y)
```

Neither method is more specific than the other. The first method is not more specific because the type of its second parameter is `double` and `double` values cannot be assigned to `float` variables. The second method is not more specific because of a similar problem with its first parameter.

Step Three

After successfully completing the previous two steps, the Java compiler knows that the expression in our example will call a method named `square()` and that the method will take one `float` argument. However, the compiler does not know if the method called at runtime will be the one defined in the `MyMath` class. It is possible that a subclass of `MyMath` could define a method that is also called `square()` and takes a single `float` argument. This method in a subclass would override the method in `MyMath`. If the variable `m` in the expression `m.square(3.4F)` refers to such a subclass, the method defined in the subclass is called instead of the one defined in `MyMath`.

The Java compiler generates code to determine at runtime which method named `square()` that takes a single `float` argument it should call. The Java compiler must always generate such runtime code for method call expressions, unless it is able to determine at compile time the exact method to be invoked at runtime.

There are four cases in which the compiler can know exactly which method is to be called at runtime:

- The method is called through an object reference, and the type of the reference is a `final` class. Since the type of the reference is a `final` class, Java does not allow any subclasses of that class to be defined. Therefore, the object reference will always refer to an object of the class declared `final`. The Java compiler knows the actual class that the reference will refer to, so it can know the actual method to be called at runtime.

- The method is invoked through an object reference, and the type of the reference is a class that defines or inherits a `final` method that has the method name, number of parameters, and types of parameters determined by the preceding steps. In this case, the compiler knows the actual method to be called at runtime because `final` methods cannot be overridden.

- The method is a `static` method. When a method is declared static, it is also implicitly declared final. Thus, the compiler can be sure that the method to be called at runtime is the one defined in or inherited by the specified class that has the method name, number of parameters, and types of parameters determined by the preceding steps.

- The compiler is able to deduce that a method is invoked through an object reference that will always refer to the same class of object at runtime. One way the compiler might deduce this is through data flow analysis.

If none of the above cases applies to a method call expression, the Java compiler must generate runtime code to determine the actual method to be invoked. The runtime selection process begins by getting the class of the object through which the method is being invoked. This class is searched for a method that has the same name and the same number and types of parameters as the method selected in Step Two. If this class does not contain such a definition, its immediate superclass is searched. If the immediate superclass does not contain an appropriate definition, its superclasses are searched, and so on up the inheritance hierarchy. This search process is called *dynamic method lookup*.

Dynamic method lookup always begins with the class of the actual object being referenced. The type of the reference being used to access the object does not influence where the search for a method begins. The one exception to this rule occurs when the keyword `super` is used as part of the method call expression. The form of this type of method call expression is:

```
super.Identifier(...)
```

In this case, dynamic method lookup begins by searching the superclass of the class that the calling code appears in.

Now that we've gone through the entire method selection process, let's consider an example that illustrates the process:

```
class A {}
class B extends A {}
class C extends B {}
class D extends C {}
class W {
    void foo(D d) {System.out.println("C");}
}
class X extends W {
    void foo(A a) {System.out.println("A");}
    void foo(B b) {System.out.println("X.B");}
}
class Y extends X {
    void foo(B b) {System.out.println("Y.B");}
}
class Z extends Y {
    void foo(C c) {System.out.println("D");}
}
```

```
public class CallSelection {
    public static void main(String [] argv) {
        Z z = new Z();
        ((X) z).foo(new C());
    }
}
```

In the class CallSelection, the method main() contains a call to a method named foo(). This method is called through an object reference. Although the object refers to an instance of the class Z, it is treated as an instance of the class X because the reference is type cast to the class X. The process of selecting which method to call proceeds as follows:

1. The compiler finds all of the methods named foo() that are accessible through an object of class X: foo(A), foo(B), and foo(D). However, because a reference to an object of class C cannot be assigned to a variable of class D, foo(D) is not a candidate to be invoked by the method call expression.

2. Now the compiler must choose one of the two remaining foo() methods as more specific than the other. Both methods are defined in the same class, but foo(B) is more specific than foo(A) because a reference to an object of class B can be assigned to a variable declared with a type of class A.

3. At runtime, the dynamic method lookup process finds that it has a reference to an object of class Z. The fact that the reference is cast to class X is not significant, since dynamic lookup is concerned with the class of an object, not the type of the reference used to access the object. The definition of class Z is searched for a method named foo() that takes one parameter that is a reference to an object of class B. No such method is found in the definition of class Z, so its immediate superclass, class Y, is searched. Such a method is found in class Y, so that method is invoked.

Here is another example that shows some ambiguous and erroneous method call expressions:

```
class A {}
class B extends A {}
class AmbiguousCall {
    void foo(B b, double x){}
    void foo(A a, int i){}
    void doit() {
        foo(new A(), 8);   // Matches foo(A, int)
        foo(new A(), 8.0); // Error: doesn't match anything
        foo(new B(), 8);   // Error: ambiguous, matches both
        foo(new B(), 8.0); // Matches foo(B, double)
    }
}
```

References Assignment Compatibility; *ClassOrInterfaceName* 4.1.6; Casts; *Expression* 4; Identifiers; Inheritance; Interface Methods; Methods; Primary Expressions

Class Literals

A *class literal* is an expression that produces a reference to a `Class` object that identifies a specified data type. Class literals are not supported prior to Java 1.1. Here's the syntax for a class literal:

[Graphic: Figure from the text]

If the type in a class literal is a reference type, the class literal produces a reference to the `Class` object that defines the specified reference type. The following are some examples of this type of class literal:

```
String.class
java.util.Stack.class
myNewClass.class
```

Such a class literal can throw a `NoClassDefFoundError` if the specified class is not available.

You can also call `Class.forName()` with the name of a specified reference type to retrieve the `Class` object for that type. For example:

```
Class.forName("java.util.Stack")
```

A class literal and a call to `Class.forName()` for the same reference type return the same `Class` object. There are certain situations when it makes sense to use a class literal, while in other cases a call to `Class.forName()` is more appropriate. Here are the differences between the two techniques for retrieving a `Class` object:

- A class literal cannot contain an expression, so it always refers to the same type. However, the argument passed to `Class.forName()` can be an expression that produces different strings that name different classes.

- The class or interface name passed to `Class.forName()` must be fully qualified by its package name. The class or interface name in a class literal, however, does not typically need to include a package name because the Java compiler can use information provided in package and import directives to deduce the package name.

- The name of an inner class can be used directly with a class literal. Because of the way that inner-class names are encoded, however, when an inner-class name is passed to `Class.forName()`, the name must contain dollar signs ($) in place of dots (`.`).

- The efficiency of a class literal is comparable to a field reference; it is more efficient than the method call required by `Class.forName()`.

If the type in a class literal is `void` or a primitive type, the class literal produces a reference to a unique `Class` object that represents `void` or the specified type. The special `Class` object that represents `void` or a primitive type can be distinguished from a `Class` object that represents a reference type by calling its `isPrimitive()` method. This method only returns `true` if the `Class` object represents `void` or a primitive type. The `getName()` method of a special `Class` object returns a string that contains the name of the primitive type represented by the object. The easiest way to determine the primitive type of a special `Class` object is to compare it to the `TYPE` variables of the primitive wrapper classes. The following comparisons always produce `true`:

```
boolean.class == Boolean.TYPE
byte.class == Byte.TYPE
short.class == Short.TYPE
int.class == Integer.TYPE
long.class == Long.TYPE
char.class == Character.TYPE
float.class == Float.TYPE
double.class == Double.TYPE
void.class == Void.TYPE
```

References Boolean; Byte; Character; Class; Double; Errors; Float; Inner Classes; Integer; Long; Short; Void; *Type* 3

5. Declarations

Contents:

A declaration is a construct that associates a name with storage that contains specified data or a specified type of data. More specifically, declarations associate names with classes, interfaces, methods, and variables. In addition, the declaration of a class, interface, or method defines the actual class, interface, or method that is associated with the name. Methods and variables can only be declared within classes and interfaces, so this chapter covers method and variable declarations in the context of class and interface declarations.

Every name has a *lexical scope*. The scope of a declaration determines the portions of a program in which the declaration is applicable.

A declaration can be preceded by modifiers that specify attributes of the name or of the data associated with the name. One such attribute for a name is its accessibility. The accessibility modifiers specify the other classes that can access the data associated with the name. The `static` modifier specifies an attribute for data; it indicates whether the data is associated with a class or with individual instances of a class.

Because Java is an object-oriented programming language, this chapter also describes the object-oriented model used by the language. An understanding of this model is necessary for a complete understanding of class and interface declarations.

5.1 Naming Conventions

The Java language has no requirements for choosing names, aside from the lexical requirements for identifiers stated in Identifiers. However, there are certain conventions that you should follow when choosing names; these conventions are the ones used by Sun in much of the Java API. Following these conventions makes your programs easier to read, as many programmers are already accustomed to reading programs that use them:

- If an identifier is logically made up of multiple words, the first letter of each word other than the first is uppercase and the rest of the letters are lowercase (e.g., `aSimpleExample`). Sun is consistent about following this convention.
- The first letter of the name of a class or interface is uppercase, while the first letter of all other names is lowercase. Sun is also consistent about following this convention.
- The names of final variables that are intended to represent symbolic constants are all uppercase; logical words contained in the name are separated by underscore characters (e.g., `MAX_LEGAL_VALUE`). Sun uses this convention quite often, but is not entirely consistent.
- Some Java programmers have adopted the additional convention of beginning the names of instance variables with an underscore (e.g., `_value`).
- Avoid the use of `$` in names to prevent confusion with compiler-generated names. Sun is consistent about following this convention.

References Class Name; Identifiers; Interface Name; Interface Variables; Variables

PREVIOUS | HOME | NEXT
Constant Expressions | BOOK INDEX | Lexical Scope of Declarations

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

6. Statements and Control Structures

Contents:

A statement is the construct used to control the flow of program execution in Java:

[Graphic: Figure from the text]

Statements are executed in sequence, unless a statement alters the flow of control. Statements usually correspond to executable code.

References Blocks; The break Statement; The continue Statement; The Empty Statement; Expression Statements; The if Statement; Iteration Statements; Labeled Statements; The return Statement; The switch Statement; The synchronized Statement; The throw Statement; The try Statement

6.1 Blocks

A block is a sequence of zero or more statements, local variable declarations, or local class declarations enclosed in curly braces:

[Graphic: Figure from the text]

The bodies of methods, constructors, static initializers, and instance initializers are blocks. A variable declaration in a block causes a local variable to be defined, while a class declaration in a block causes a local class to be defined. A block is itself a kind of statement, so a block can contain other blocks. Here is an example of a block:

```
{
    int tmp = x;
    x = y;
    y = tmp;
}
```

The statements in a block are executed in the sequence in which they occur, unless a statement that alters the sequence of execution is executed. If, as a result of such a statement, the Java compiler can determine that a statement will never be executed, the compiler is required to produce an error message about the unreachable statement.

The one exception to this rule allows `if` statements that have constant Boolean expressions. The compiler recognizes `if` statements that have constant Boolean expressions and does not generate code for the portion of the statement that can never be executed. This mechanism can be used for conditional compilation; it is similar to the C/C++ preprocessor features that are used for this purpose.

References Constant Expressions; Constructors; Instance Initializers; Local Classes; Local Variables; Methods; *Statement* 6; Static Initializers; The switch Statement

Local Variables

Local variables declared in a block exist only from their declaration to the end of the block. A local variable declaration cannot include any modifiers except the `final` modifier. In other words, a variable declaration in a block cannot include any of the following keywords: `public`, `protected`, `private`, `static`, `transient`, or `volatile`. The syntax that permits the use of the `final` modifier with local variables is new as of Java 1.1; the usage is not permitted with earlier versions of the language.

The syntax of a local variable declaration is:

[Graphic: Figure from the text]

A local variable declaration is really made up of two distinct things:

- A type, which can be either a primitive type or a reference type.
- Any number of identifiers that name variables. Each name can be followed by pairs of square brackets to indicate an array variable, as well as an optional initializer for the variable.

A local variable declared within a block that has an initializer is initialized when its declaration is executed. Within the body of a method or constructor, its formal parameters are treated as local variables. Formal parameters are initialized when a method is called. A local variable can also be declared in the header of a `for` statement.

The following are some examples of local variable declarations:

```
int x;
double[] k, m[];
```

References Variable initializers; *Expression* 4; Identifiers; Interface Variables; *Type* 3; The for Statement; Variables

Final local variables

If a local variable is declared with the `final` modifier, the variable is a named constant value. As such, it must be assigned an initial value. Any assignment to a `final` local variable, other than the one that provides its initial value, is a compile-time error. The initial value for a `final` local variable is typically provided by an initializer that is part of the variable's declaration. For example:

```
final int X = 4;
```

A `final` local variable that is not initialized in its declaration is called a *blank final*. A blank final must be assigned a value exactly once. The compiler uses flow analysis that takes `if` statements and iteration statements into account to ensure that a blank final is assigned a value exactly once. Thus, it is possible to have multiple assignments to a blank final, so long as exactly one of them can be executed. For example, here is an instance initializer that sets the value of a blank final:

```
{
    final int DAYS_IN_YEAR;
    if (isLeapYear(new Date()))
        DAYS_IN_YEAR = 366;
    else
        DAYS_IN_YEAR = 365;
    ...
}
```

Local variables that are declared `final` are not supported prior to Java 1.1.

References Instance Initializers; The do Statement; The for Statement; The if Statement; The while Statement; Variable modifiers

Local variable type

A local variable declaration must always specify the type of the variable. If the declaration of a local variable uses a primitive type, the variable contains a value of the specified primitive type. If the declaration uses a reference type, the variable contains a reference to the specified type of object.

The presence of square brackets in a variable declaration, after either the type or the variable name, indicates that the variable contains a reference to an array. For example:

```
int a[];        // a is an array of int
int[] b;        // b is also an array of int
```

It is also possible to declare a variable to contain an array of arrays, or more generally, arrays nested to any level. Each pair of square brackets in the declaration corresponds to a dimension of the array; it makes no difference whether the brackets appear after the type or the variable name. For example:

```
int[][][] d3;      // Each of these is an array of
int[][] f3[];      // arrays of arrays of integers
int[] g3[][];
int h3[][][];
int[] j3, k3[];    // An array and an array of arrays
```

References Array Types; Primitive Types; Reference Types

Local variable name

The identifier that follows the variable type is the name of the local variable. When a local variable definition is in effect, all occurrences of that name are taken to mean the local variable. If a local variable is declared with the same name as a class, an interface, or a field of the class in which the local variable is declared, the definition of the class, interface, or field is hidden. Fields that are hidden by a local variable can be referenced using the keyword `this`. For example:

```
class myClass {
    int value;
    void doit(int x) {
        int value;
        value = x*4;                // Set local variable
        this.value = value + 1;     // Set field variable
}
```

A block cannot have multiple local variables with the same name. This means that a local variable cannot be declared at a point in a block where a local variable with the same name is already defined. For example, consider the following code:

```
myMethod(char c){
    int j;          // Okay
    char c;         // Error
    int j;          // Error
    {
        int j;      // Error
    }
    {
        int x;      // Okay
    }
    {
        int x;      // Okay
    }
    int x;          // Okay
}
```

In the above example, the declaration of `c` as a local variable is an error because it occurs in a method that has a formal parameter with that name. The second declaration of `j` is an error because there is already a local variable defined with that name. The third declaration of `j` as a local variable is also an error for the same reason; the nested block sees all of the declarations that are visible in the enclosing block, including the declaration of `j` in the outer block.

The first declaration of `x` is fine because there is no previous declaration of `x` for it to conflict with. The second declaration of `x` is also fine because there is no previous declaration of `x` in the enclosing block for it to conflict with. The first declaration of `x` occurs in a nested block, so it is not visible in the enclosing block. The third declaration of `x` is also fine because the preceding declarations occurred in nested blocks; they are not visible in the enclosing block.

References Identifiers; this

Local variable initializers

A local variable declaration can contain an initializer. If the variable is of a non-array type, the expression in the initializer is evaluated and the variable is set to the result of the expression, as long as the result is assignment-compatible with the variable. If the variable is of an array type, the initializer must be an array initializer, as described in Variable initializers. If the variable is declared `final`, the initializer sets the value of the named constant.

A local variable declaration with an initializer is similar in effect to a local variable declaration without an initializer immediately followed by an assignment statement that sets the declared variable. Take the following example:

```
int a = 4;
```

This is equivalent to:

```
int a;
a = 4;
```

If a local variable has an initializer, the value of the variable is set to the value of the initializer when the declaration is executed.

Any attempt to access the value of a local variable before its value is set by an assignment statement or an initializer is treated as an error by the Java compiler. For example:

```
int foo(int x) {
    int a = x + 1;
    int b, c;
    if (a > 4)
        b = 3;
    a = a * c;        // Error: c not initialized
    a = b * 8 + a;    // Error: b might not be initialized
```

This example contains two errors. First, the compiler complains about the expression `a*c` because `c` is not initialized. The compiler also complains about the expression `b* 8+a` because the preceding assignment to `b` may not executed, depending on the value of `a`. If the compiler cannot guarantee that a local variable will be initialized, it generates an error message when the variable is used.

References Variable initializers; Assignment Operators

Local Classes

Local classes declared in a block exist only in the scope of that block. Local classes are not supported prior to Java 1.1 Here's the syntax of a local class declaration:

[Graphic: Figure from the text]

A local class can access local variables in the enclosing block that are declared `final`. A local class can also access instance variables of the enclosing class if they are not declared inside of a `static` method or static initializer.

There is an alternate syntax for a local class that allows an anonymous local class to be defined. This syntax is available as part of an allocation expression.

References Allocation Expressions; Anonymous classes; Local classes; Local Variables; Class Declarations; Variables

Local class modifiers

The keywords `abstract` and `final` can be used in the declaration of a local class. These modifiers have the following meanings:

`abstract`

> If a local class is declared `abstract`, no instances of the class may be created. A local class declared `abstract` may contain `abstract` methods. Classes not declared `abstract` may not contain `abstract` methods and must override any `abstract` methods they inherit with methods that are not `abstract`. Classes that implement an interface and are not declared `abstract` must contain or inherit methods that are not `abstract` that have the same name, have the same number of parameters, and have corresponding parameter types as the methods declared in the interfaces that the class implements.

`final`

> If a local class is declared `final`, it cannot be subclassed. In other words, it cannot appear in the `extends` clause of another class.

References Class Modifiers; Inner class modifiers

Local class members

The body of a local class cannot declare any `static` variables, `static` methods, `static` classes, or static initializers. Beyond those restrictions, the remainder of the declaration is the same as that for a top-level class declaration, which is described in Class Declarations.

References Class Declarations; Constructors; Instance Initializers; Methods; Nested Top-Level and Member Classes; Static Initializers; Variables

PREVIOUS
Interface Declarations

HOME
BOOK INDEX

NEXT
Labeled Statements

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

7. Program Structure

Contents:

This chapter discusses the higher levels of program structure for Java programs. The two levels of organization discussed in this chapter are compilation units and packages. A *compilation unit* contains the source code for one or more classes or interfaces. A *package* is a collection of related compilation units.

This chapter also discusses the two most common top-level Java program architectures: applications and applets. An *application* is a stand-alone Java program that can be run directly from the command line (or other operating system environment). An *applet* is a Java program that must be run from within another program, such as a Web browser. In the future, applets will even be hosted by other environments, such as cellular phones and personal digital assistants.

7.1 Compilation Units

A compilation unit is the highest-level syntactic structure that Java recognizes:

[Graphic: Figure from the text]

Only one of the classes or interfaces declared in a compilation unit can be declared `public`.

A compilation unit usually corresponds to a single source code file. However, the Java language specification allows compilation units to be stored in a database. If compilation units are stored in a database, the limit of one `public` class or interface per compilation unit does not apply, as long as there is a way to extract the compilation units from the database and place them in individual files that contain no more than one `public` class or interface per file. This exception to the one `public` class or interface per compilation unit rule is useful if you are implementing a Java development environment.

Every compilation unit is part of exactly one package. That package is specified by the `package` directive that appears at the beginning of the compilation unit. If there is no `package` directive, the compilation unit is part of the default package.

References *ClassDeclaration* 5.4; Class Modifiers; The import Directive; Interface Declarations; Interface Modifiers; *PackageDirective* 7.2

8. Threads

Contents:

Threads provide a way for a Java program to do multiple tasks concurrently. A thread is essentially a flow of control in a program and is similar to the more familiar concept of a process. An operating system that can run more than one program at the same time uses processes to keep track of the various programs that it is running. However, processes generally do not share any state, while multiple threads within the same application share much of the same state. In particular, all of the threads in an application run in the same address space, sharing all resources except the stack. In concrete terms, this means that threads share field variables, but not local variables.

When multiple processes share a single processor, there are times when the operating system must stop the processor from running one process and start it running another process. The operating system must execute a sequence of events called a *context switch* to transfer control from one process to another. When a context switch occurs, the operating system has to save a lot of information for the process that is being paused and load the comparable information for the process being resumed. A context switch between two processes can require the execution of thousands of machine instructions. The Java virtual machine is responsible for handling context switches between threads in a Java program. Because threads share much of the same state, a context switch between two threads typically requires the execution of less than 100 machine instructions.

There are a number of situations where it makes sense to use threads in a Java program. Some programs must be able to engage in multiple activities and still be able to respond to additional input from the user. For example, a web browser should be able to respond to user input while fetching an image or playing a sound. Because threads can be suspended and resumed, they can make it easier to control multiple activities, even if the activities do not need to be concurrent. If a program models real world objects that display independent, autonomous behavior, it makes sense to use a separate thread for each object. Threads can also implement asynchronous methods, so that a calling method does not have to wait for the method it calls to complete before continuing with its own activity.

Java applets make considerable use of threads. For example, an animation is generally implemented with a separate thread. If an applet has to download extensive information, such as an image or a sound, to initialize itself, the initialization can take a long time. This initialization can be done in a separate thread to prevent the initialization from interfering with the display of the applet. If an applet needs to process messages from the network, that work generally is done in a separate thread so that the applet can continue painting itself on the screen and responding to mouse and keyboard events. In addition, if each message is processed separately, the applet uses a separate thread for each message.

For all of the reasons there are to use threads, there are also some compelling reasons not to use them. If a program uses inherently sequential logic, where one operation starts another operation and then must wait for the other operation to complete before continuing, one thread can implement the entire sequence. Using multiple threads in such a case results in a more complex program with no accompanying benefits. There is considerable overhead in creating and starting a thread, so if an operation involves only a few primitive statements, it is faster to handle it with a single thread. This can even be true when the operation is conceptually asynchronous. When multiple threads share objects, the objects must use synchronization mechanisms to coordinate thread access and maintain consistent state. Synchronization mechanisms add complexity to a program, can be difficult to tune for optimal performance, and can be a source of bugs.

8.1 Using Thread Objects

The `Thread` class in the `java.lang` package creates and controls threads in Java programs. The execution of Java code is always under the control of a `Thread` object. The `Thread` class provides a `static` method called `currentThread()` that provides a reference to the `Thread` object that controls the current thread of execution.

References Thread

Associating a Method with a Thread

The first thing you need to do to make a `Thread` object useful is to associate it with a method you want it to run. Java provides two ways of associating a method with a `Thread`:

- Declare a subclass of `Thread` that defines a `run()` method.

- Pass a reference to an object that implements the `Runnable` interface to a `Thread` constructor.

For example, if you need to load the contents of a URL as part of an applet's initialization, but the applet can provide other functionality before the content is loaded, you might want to load the content in a separate thread. Here is a class that does just that:

```
import java.net.URL;
class UrlData extends Thread   {
    private Object data;
    private URL url
    public UrlData(String urlName) throws MalformedURLException {
        url = new URL(urlName);
        start();
    }
    public void run(){
        try {
            data = url.getContent();
        } catch (java.io.IOException  e) {
        }
    }
    public Object getUrlData(){
        return data;
    }
}
```

The `UrlData` class is declared as a subclass of `Thread` so that it can get the contents of the URL in a separate thread. The constructor creates a `java.net.URL` object to fetch the contents of the URL, and then calls the `start()` method to start the thread. Once the thread is started, the constructor returns; it does not wait for the contents of the URL to be fetched. The `run()` method is executed after the thread is started; it does the real work of fetching the data. The `getUrlData()` method is an access method that returns the value of the `data` variable. The value of this variable is `null` until the contents of the URL have been fetched, at which time it contains a reference to the actual data.

Subclassing the `Thread` class is convenient when the method you want to run in a separate thread does not need to belong to a particular class. Sometimes, however, you need the method to be part of a particular class that is a subclass of a class other than `Thread`. Say, for example, you want a graphical object that is displayed in a window to alternate its background color between red and blue once a second. The object that implements this behavior needs to be a subclass of the `java.awt.Canvas` class. However, at the same time, you need a separate thread to alternate the color of the object once a second.

In this situation, you want to tell a `Thread` object to run code in another object that is not a subclass of the `Thread` class. You can accomplish this by passing a reference to an object that implements the `Runnable` interface to the constructor of the `Thread` class. The `Runnable` interface requires that an object has a `public` method called `run()` that takes no arguments. When a `Runnable` object is passed to the constructor of the `Thread` class, it creates a `Thread` object that calls the `Runnable` object's `run()` method when the thread is started. The following example shows part of the code that implements an object that alternates its background color between red and blue once a second:

```
class AutoColorChange extends java.awt.Canvas implements Runnable {
    private Thread myThread;
    AutoColorChange () {
        myThread = new Thread(this);
        myThread.start();
        ...
    }
    public void run() {
        while (true) {
            setBackground(java.awt.Color.red);
            repaint();
            try {
                myThread.sleep(1000);
            } catch (InterruptedException e) {}
            setBackground(java.awt.Color.blue);
            repaint();
            try {
                myThread.sleep(1000);
            } catch (InterruptedException e) {}
        }
    }
}
```

The `AutoChangeColor` class extends `java.awt.Canvas`, alternating the background color between red and blue once a second. The constructor creates a new `Thread` by passing the current object to the `Thread` constructor, which tells the `Thread` to call the `run()` method in the `AutoChangeColor` class. The constructor then starts the new thread by calling its `start()` method, so that the color change happens asynchronously of whatever else is going on. The class has an instance variable called `myThread` that contains a reference to the `Thread` object, so that can control the thread. The `run()` method takes care of changing the background color, using the `sleep()` method of the `Thread` class to temporarily suspend the thread and calling `repaint()` to redisplay the object after each color change.

References Runnable; Thread

Controlling a Thread

As shown in the previous section, you start a `Thread` by calling its `start()` method. Before the `start()` method is called, the `isAlive()` method of the `Thread` object always returns `false`. When the `start()` method is called, the `Thread` object becomes associated with a scheduled thread in the underlying environment. After the `start()` method has returned, the `isAlive()` method always returns `true`. The `Thread` is now scheduled to run until it dies, unless it is suspended or in another unrunnable state.

It is actually possible for `isAlive()` to return `true` before `start()` returns, but not before `start()` is called. This can happen because the `start()` method can return either before the started `Thread` begins to run or after it begins to run. In other words, the method that called `start()` and the new thread are now running concurrently. On a multiprocessor system, the `start()` method can even return at the same time the started `Thread` begins to run.

`Thread` objects have a parent-child relationship. The first thread created in a Java environment does not have a parent `Thread`. However, after the first `Thread` object is created, the `Thread` object that controls the thread used to create another `Thread` object is considered to be the parent of the newly created `Thread`. This parent-child relationship is used to supply some default values when a `Thread` object is created, but it has no further significance after a `Thread` has been created.

References Thread

Stopping a thread

A thread dies when one of the following things happens:

- The `run()` method called by the `Thread` returns.
- An exception is thrown that causes the `run()` method to be exited.
- The `stop()` method of the `Thread` is called.

The `stop()` method of the `Thread` class works by throwing a `ThreadDeath` object in the `run()` method of the thread. Normally, you should not catch `ThreadDeath` objects in a `try` statement. If you need to catch `ThreadDeath` objects to detect that a `Thread` is about to die, the `try` statement that catches `ThreadDeath` objects should rethrow them.

When an object (`ThreadDeath` or otherwise) is thrown out of the `run()` method for the `Thread`, the `uncaughtException()` method of the `ThreadGroup` for that `Thread` is called. If the thrown object is an instance of the `ThreadDeath` class, the thread dies, and the thrown object is ignored. Otherwise, if the thrown object is of any other class, `uncaughtException()` calls the thrown object's `printStackTrace()` method, the thread dies, and the thrown object is ignored. In either case, if there are other nondaemon threads running in the system, the current program continues to run.

References Errors; The try Statement; Thread; ThreadGroup

Interrupting a thread

In some situations, you need to kill a thread in a way that allows it to complete what it is currently doing before dying. For example, if a thread is in the middle of processing a transaction, you might want the transaction to complete before the thread dies. The `Thread` class provides support for this in the form of the `interrupt()` method.

There are a number of methods in the Java API, such as `wait()` and `join()`, that are declared as throwing an `InterruptedException`. Both of these methods temporarily suspend the execution of a thread. In Java 1.1, if a thread is waiting for one of these methods to return and another thread calls `interrupt()` on the waiting thread, the method that is waiting throws an `InterruptedException`.

The `interrupt()` method sets an internal flag in a `Thread` object. Before the `interrupt()` method is called, the `isInterrupted()` method of the `Thread` object always returns `false`. After the `interrupt()` method is called, `isInterrupted()` returns `true`.

Prior to version 1.1, the methods in the Java API that are declared as throwing an `InterruptedException` do not actually do so. However, the `isInterrupted()` method does return `True` if the thread has been interrupted. Thus, if the code in the `run()` method for a thread periodically calls `isInterrupted()`, the thread can respond to a call to `interrupt()` by shutting down in an orderly fashion.

References Other exceptions; Thread

Thread priority

One of the attributes that controls the behavior of a thread is its priority. Although Java does not guarantee much about how threads are scheduled, it does guarantee that a thread with a priority that is higher than that of another thread will be scheduled to run at least as often, and possibly more often, than the thread with the lower priority. The priority of a thread is set when the `Thread` object is created, by passing an argument to the constructor that creates the `Thread` object. If an explicit priority is not specified, the `Thread` inherits the priority of its parent `Thread` object.

You can query the priority of a `Thread` object by calling its `getPriority()` method. Similarly, you can set the priority of a `Thread` using its `setPriority()` method. The priority you specify must be greater than or equal to `Thread.MIN_PRIORITY` and less than or equal to `Thread.MAX_PRIORITY`.

Before actually setting the priority of a `Thread` object, the `setPriority()` method checks the maximum allowable priority for the `ThreadGroup` that contains the `Thread` by calling `getMaxPriority()` on the `ThreadGroup`. If the call to `setPriority()` tries to set the priority to a value that is higher than the maximum allowable priority for the `ThreadGroup`, the priority is instead set to the maximum priority. It is possible for the current priority of a `Thread` to be greater than the maximum allowable priority for the `ThreadGroup`. In this case, an attempt to raise the priority of the `Thread` results in its priority being lowered to the maximum priority.

References Thread; ThreadGroup

Daemon threads

A daemon thread is a thread that runs continuously to perform a service, without having any connection with the overall state of the program. For example, the thread that runs the garbage collector in Java is a daemon thread. The thread that processes mouse events for a Java program is also a daemon thread. In general, threads that run application code are not daemon threads, and threads that run system code are daemon threads. If a thread dies and there are no other threads except daemon threads alive, the Java virtual machine stops.

A `Thread` object has a `boolean` attribute that specifies whether or not a thread is a daemon thread. The daemon attribute of a thread is set when the `Thread` object is created, by passing an argument to the constructor that creates the `Thread` object. If the daemon attribute is not explicitly specified, the `Thread` inherits the daemon attribute of its parent `Thread` object.

The daemon attribute is queried using the `isDaemon()` method; it is set using the `setDaemon()` method.

References Thread

Yielding

When a thread has nothing to do, it can call the `yield()` method of its `Thread` object. This method tells the scheduler to run a different thread. The value of calling `yield()` depends largely on whether the scheduling mechanism for the platform on which the program is running is preemptive or nonpreemptive.

By choosing a maximum length of time a thread can continuously, a *preemptive* scheduling mechanism guarantees that no single thread uses more than its fair share of the processor. If a thread runs for that amount of time without yielding control to another thread, the scheduler preempts the thread and causes it to stop running so that another thread can run.

A *nonpreemptive* scheduling mechanism cannot preempt threads. A nonpreemptive scheduler relies on the individual threads to yield control of the processor frequently, so that it can provide reasonable performance. A thread explicitly yields control by calling the `Thread` object's `yield()` method. More often, however, a thread implicitly yields control when it is forced to wait for something to happen elsewhere.

Calling a Thread object's yield() method during a lengthy computation can be quite valuable on a platform that uses a nonpreemptive scheduling mechanism, as it allows other threads to run. Otherwise, the lengthy computation can prevent other threads from running. On a platform that uses a preemptive scheduling mechanism, calling yield() does not usually make any noticeable difference in the responsiveness of threads.

Regardless of the scheduling algorithm that is being used, you should not make any assumptions about when a thread will be scheduled to run again after it has called yield(). If you want to prevent a thread from being scheduled to run until a specified amount of time has elapsed, you should call the sleep() method of the Thread object. The sleep() method takes an argument that specifies a minimum number of milliseconds that must elapse before the thread can be scheduled to run again.

References Thread

Controlling groups of threads

Sometimes it is necessary to control multiple threads at the same time. Java provides the ThreadGroup class for this purpose. Every Thread object belongs to a ThreadGroup object. By passing an argument to the constructor that creates the Thread object, the ThreadGroup of a thread can be set when the Thread object is created. If an explicit ThreadGroup is not specified, the Thread belongs to the same ThreadGroup as its parent Thread object.

References Thread; ThreadGroup

9. Exception Handling

Contents:

Exception handling is a mechanism that allows Java programs to handle various exceptional conditions, such as semantic violations of the language and program-defined errors, in a robust way. When an exceptional condition occurs, an *exception* is thrown. If the Java virtual machine or run-time environment detects a semantic violation, the virtual machine or run-time environment implicitly throws an exception. Alternately, a program can throw an exception explicitly using the `throw` statement. After an exception is thrown, control is transferred from the current point of execution to an appropriate `catch` clause of an enclosing `try` statement. The `catch` clause is called an exception handler because it handles the exception by taking whatever actions are necessary to recover from it.

9.1 Handling Exceptions

The `try` statement provides Java's exception-handling mechanism. A `try` statement contains a block of code to be executed. Putting a block in a `try` statement indicates that any exceptions or other abnormal exits in the block are going to be handled appropriately. A `try` statement can have any number of optional `catch` clauses that act as exception handlers for the `try` block. A `try` statement can also have a `finally` clause. The `finally` block is always executed before control leaves the `try` statement; it cleans up after the `try` block. Note that a `try` statement must have either a `catch` clause or a `finally` clause.

Here is an example of a `try` statement that includes a `catch` clause and a `finally` clause:

```
try {
    out.write(b);
} catch (IOException e) {
    System.out.println("Output Error");
} finally {
    out.close();
}
```

If `out.write()` throws an `IOException`, the exception is caught by the `catch` clause. Regardless of whether `out.write()` returns normally or throws an exception, the `finally` block is executed, which ensures that `out.close()` is always called.

A `try` statement executes the block that follows the keyword `try`. If an exception is thrown from within the `try` block and the `try` statement has any `catch` clauses, those clauses are searched, in order, for one that can handle the exception. If a `catch` clause handles an exception, that `catch` block is executed.

However, if the `try` statement does not have any `catch` clauses that can handle the exception (or does not have any `catch` clauses at all), the exception propagates up through enclosing statements in the current method. If the current method does not contain a `try` statement that can handle the exception, the exception propagates up to the invoking method. If this method does not contain an appropriate `try` statement, the exception propagates up again, and so on. Finally, if no `try` statement is found to handle the exception, the currently running thread terminates.

A `catch` clause is declared with a parameter that specifies the type of exception it can handle. The parameter in a `catch` clause must be of type `Throwable` or one of its subclasses. When an exception occurs, the `catch` clauses are searched for the first one with a parameter that matches the type of the exception thrown or is a superclass of the thrown exception. When the appropriate `catch` block is executed, the actual exception object is passed as an argument to the `catch` block. The code within a `catch` block should do whatever is necessary to handle the exceptional condition.

The `finally` clause of a `try` statement is always executed, no matter how control leaves the `try` statement. Thus it is a good place to handle clean-up operations, such as closing files, freeing resources, and closing network connections.

References The throw Statement; The try Statement; Throwable

10. The java.lang Package

Contents:

Select a new section and then Go

Byte
Character
Class
ClassLoader
Cloneable
Compiler
Double
Float
Integer
Long
Math

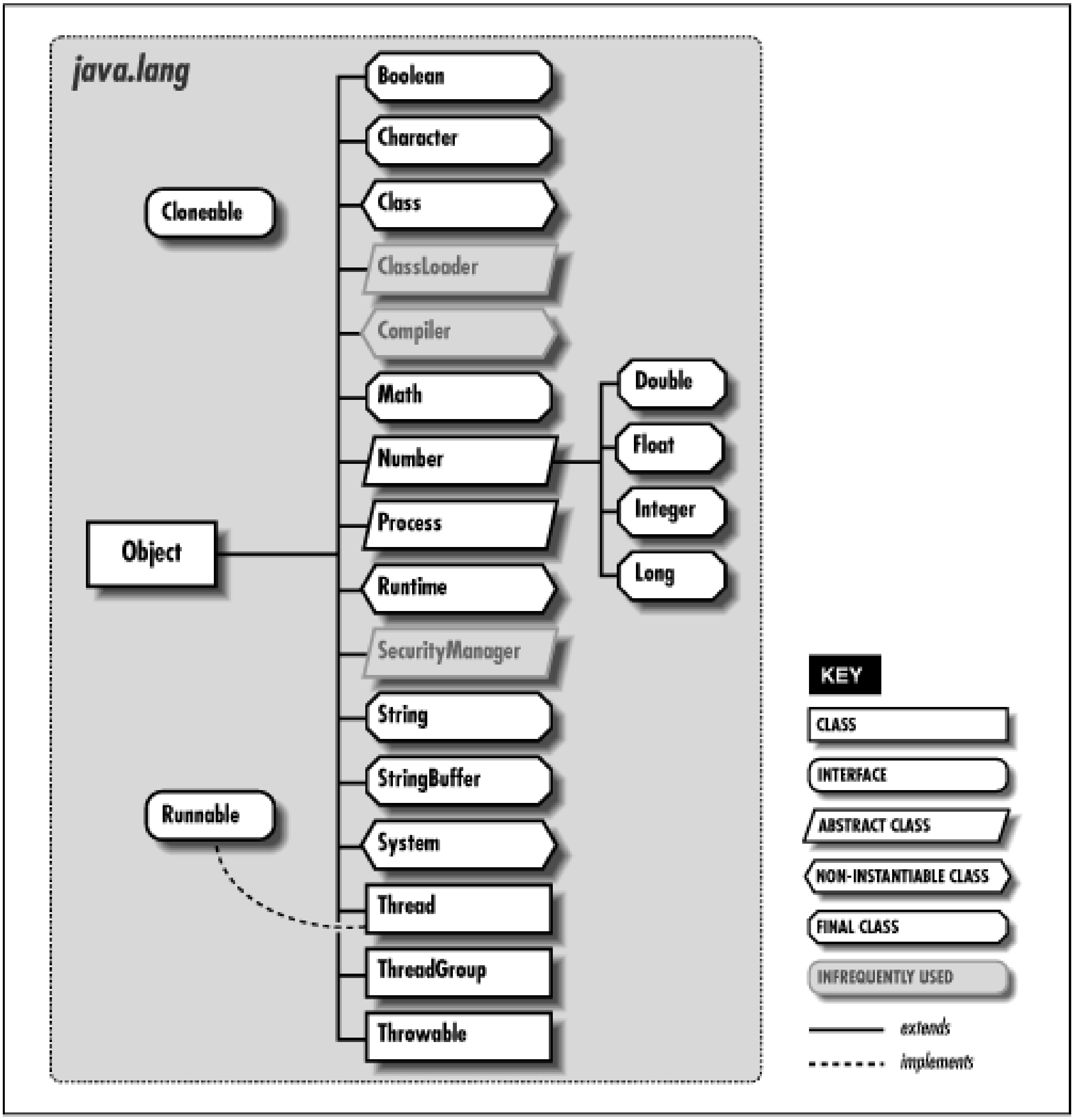
The package `java.lang` contains classes and interfaces that are essential to the Java language. These include:

- `Object`, the ultimate superclass of all classes in Java
- `Thread`, the class that controls each thread in a multithreaded program
- `Throwable`, the superclass of all error and exception classes in Java
- Classes that encapsulate the primitive data types in Java
- Classes for accessing system resources and other low-level entities
- `Math`, a class that provides standard mathematical methods
- `String`, the class that is used to represent strings

Because the classes in the `java.lang` package are so essential, the `java.lang` package is implicitly imported by every Java source file. In other words, you can refer to all of the classes and interfaces in `java.lang` using their simple names.

Figure 10.1 shows the class hierarchy for the `java.lang` package.

Figure 10.1: The java.lang package



Boolean

Name

Boolean

Synopsis

Class Name:

`java.lang.Boolean`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

`java.io.Serializable`

Availability:

JDK 1.0 or later

Description

The `Boolean` class provides an object wrapper for a `boolean` value. This is useful when you need to treat a `boolean` value as an object. For example, there are a number of utility methods that take a reference to an `Object` as one of their arguments. You cannot specify a `boolean` value for one of these arguments, but you can provide a reference to a `Boolean` object that encapsulates the `boolean` value. Furthermore, as of JDK 1.1, the `Boolean` class is necessary to support the Reflection API and class literals.

Class Summary

```
public final class java.lang.Boolean {
    // Constants
    public final static Boolean FALSE;
    public final static Boolean TRUE;
    public final static Class TYPE;                    // New in 1.1
    // Constructors
    public Boolean(boolean value);
    public Boolean(String s);
    // Class Methods
    public static boolean getBoolean(String name);
    public static Boolean valueOf(String s);
    // Instance Methods
    public boolean booleanValue();
    public boolean equals(Object obj);
    public int hashCode();
    public String toString();
}
```

Constants

TRUE

public static final Boolean TRUE

Description

> A constant `Boolean` object that has the value `true`.

FALSE

public static final Boolean FALSE

Description

> A constant `Boolean` object that has the value `false`.

TYPE

`public static final Class TYPE`

Availability

New as of JDK 1.1

Description

The `Class` object that represents the type `boolean`. It is always true that `Boolean.TYPE == boolean.class`.

Constructors

Boolean

`public Boolean(boolean value)`

Parameters

`value`

The `boolean` value to be made into a `Boolean` object.

Description

Constructs a `Boolean` object with the given value.

`public Boolean(String s)`

Parameters

`s`

The string to be made into a `Boolean` object.

Description

Constructs a `Boolean` object with the value specified by the given string. If the string equals `'true'` (ignoring case), the value of the object is `true`; otherwise it is `false`.

Class Methods

getBoolean

`public static boolean getBoolean(String name)`

Parameters

`name`

The name of a system property.

Returns

The `boolean` value of the system property.

Description

This methods retrieves the `boolean` value of a named system property.

valueOf

`public static Boolean valueOf(String s)`

Parameters

`s`

The string to be made into a `Boolean` object.

Returns

A `Boolean` object with the value specified by the given string.

Description

This method returns a `Boolean` object with the value `true` if the string equals `"true"` (ignoring case); otherwise the value of the object is `false`.

Instance Methods

booleanValue

`public boolean booleanValue()`

Returns

> The `boolean` value contained by the object.

equals

`public boolean equals(Object obj)`

Parameters

> `obj`
>
> > The object to be compared with this object.

Returns

> `true` if the objects are equal; `false` if they are not.

Overrides

> `Object.equals()`

Description

> This method returns `true` if `obj` is an instance of `Boolean`, and it contains the same value as the object this method is associated with.

hashCode

`public int hashCode()`

Returns

A hashcode based on the boolean value of the object.

Overrides

Object.hashCode()

toString

public String toString()

Returns

"true" if the value of the object is true; "false" otherwise.

Overrides

Object.toString()

Description

This method returns a string representation of the Boolean object.

Inherited Methods

Method	Inherited From	Method	Inherited From
clone()	Object	finalize()	Object
getClass()	Object	notify()	Object
notifyAll()	Object	wait()	Object
wait(long)	Object	wait(long, int)	Object

See Also

Boolean Type; Boolean literals; Class; Object; System

A. The Unicode 2.0 Character Set

Characters	Description
\u0000 - \u1FFF	Alphabets
\u0020 - \u007F	Basic Latin
\u0080 - \u00FF	Latin-1 supplement
\u0100 - \u017F	Latin extended-A
\u0180 - \u024F	Latin extended-B
\u0250 - \u02AF	IPA extensions
\u02B0 - \u02FF	Spacing modifier letters
\u0300 - \u036F	Combining diacritical marks
\u0370 - \u03FF	Greek
\u0400 - \u04FF	Cyrillic
\u0530 - \u058F	Armenian
\u0590 - \u05FF	Hebrew
\u0600 - \u06FF	Arabic
\u0900 - \u097F	Devanagari
\u0980 - \u09FF	Bengali
\u0A00 - \u0A7F	Gurmukhi
\u0A80 - \u0AFF	Gujarati
\u0B00 - \u0B7F	Oriya
\u0B80 - \u0BFF	Tamil
\u0C00 - \u0C7F	Telugu
\u0C80 - \u0CFF	Kannada
\u0D00 - \u0D7F	Malayalam

\u0E00 - \u0E7F	Thai
\u0E80 - \u0EFF	Lao
\u0F00 - \u0FBF	Tibetan
\u10A0 - \u10FF	Georgian
\u1100 - \u11FF	Hangul Jamo
\u1E00 - \u1EFF	Latin extended additional
\u1F00 - \u1FFF	Greek extended
\u2000 - \u2FFF	Symbols and punctuation
\u2000 - \u206F	General punctuation
\u2070 - \u209F	Superscripts and subscripts
\u20A0 - \u20CF	Currency symbols
\u20D0 - \u20FF	Combining diacritical marks for symbols
\u2100 - \u214F	Letterlike symbols
\u2150 - \u218F	Number forms
\u2190 - \u21FF	Arrows
\u2200 - \u22FF	Mathematical operators
\u2300 - \u23FF	Miscellaneous technical
\u2400 - \u243F	Control pictures
\u2440 - \u245F	Optical character recognition
\u2460 - \u24FF	Enclosed alphanumerics
\u2500 - \u257F	Box drawing
\u2580 - \u259F	Block elements
\u25A0 - \u25FF	Geometric shapes
\u2600 - \u26FF	Miscellaneous symbols
\u2700 - \u27BF	Dingbats
\u3000 - \u33FF	CJK auxiliary
\u3000 - \u303F	CJK symbols and punctuation
\u3040 - \u309F	Hiragana
\u30A0 - \u30FF	Katakana
\u3100 - \u312F	Bopomofo
\u3130 - \u318F	Hangul compatibility Jamo
\u3190 - \u319F	Kanbun
\u3200 - \u32FF	Enclosed CJK letters and months
\u3300 - \u33FF	CJK compatibility

\u4E00 - \u9FFF	CJK unified ideographs: Han characters used in China, Japan, Korea, Taiwan, and Vietnam
\uAC00 - \uD7A3	Hangul syllables
\uD800 - \uDFFF	Surrogates
\uD800 - \uDB7F	High surrogates
\uDB80 - \uDBFF	High private use surrogates
\uDC00 - \uDFFF	Low surrogates
\uE000 - \uF8FF	Private use
\uF900 - \uFFFF	Miscellaneous
\uF900 - \uFAFF	CJK compatibility ideographs
\uFB00 - \uFB4F	Alphabetic presentation forms
\uFB50 - \uFDFF	Arabic presentation forms-A
\uFE20 - \uFE2F	Combing half marks
\uFE30 - \uFE4F	CJK compatibility forms
\uFE50 - \uFE6F	Small form variants
\uFE70 - \uFEFE	Arabic presentation forms-B
\uFEFF	Specials
\uFF00 - \uFFEF	Halfwidth and fullwidth forms
\uFFF0 - \uFFFF	Specials

PREVIOUS
Void

HOME
BOOK INDEX

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

6.5 Reading Applet Parameters

Example 6.4 shows an extension to our `Scribble` applet. The `ColorScribble` class is a subclass of `Scribble` that adds the ability to scribble in a configurable foreground color over a configurable background color. (The `ColorScribble` applet looks a lot like the `Scribble` applet of Figure 6.3 and is not pictured here.)

`ColorScribble` has an `init()` method that reads the value of two "applet parameters" that can be optionally specified with the `<PARAM>` tag in the applet's HTML file. The returned string values are converted to colors and specified as the default foreground and background colors for the applet. Note that the `init()` method invokes its superclass's `init()` method, just in case a future version of `Scribble` defines that method to perform initialization.

This example also introduces the `getAppletInfo()` and `getParameterInfo()` methods. These methods provide textual information about the applet (its author, its version, its copyright, etc.) and the parameters that it can accept (the parameter names, their types, and their meanings). An applet should generally define these methods, although the current generation of Web browsers do not actually ever make use of them. (The `appletviewer` application in the JDK does call these methods, however.)

Example 6.4: Reading Applet Parameters

```
import java.applet.*;
import java.awt.*;
public class ColorScribble extends Scribble {
  // Read in two color parameters and set the colors.
  public void init() {
    super.init();
    Color foreground = getColorParameter("foreground");
    Color background = getColorParameter("background");
    if (foreground != null) this.setForeground(foreground);
    if (background != null) this.setBackground(background);
  }
  // Read the specified parameter.  Interpret it as a hexadecimal
  // number of the form RRGGBB and convert it to a color.
  protected Color getColorParameter(String name) {
    String value = this.getParameter(name);
    try { return new Color(Integer.parseInt(value, 16)); }
    catch (Exception e) { return null; }
  }
  // Return information suitable for display in an About dialog box.
  public String getAppletInfo() {
    return "ColorScribble v. 0.02.  Written by David Flanagan.";
  }
  // Return info about the supported parameters.  Web browsers and applet
  // viewers should display this information, and may also allow users to
  // set the parameter values.
  public String[][] getParameterInfo() { return info; }
  // Here's the information that getParameterInfo() returns.
  // It is an array of arrays of strings describing each parameter.
  // Format: parameter name, parameter type, parameter description
  private String[][] info = {
    {"foreground", "hexadecimal color value", "foreground color"},
    {"background", "hexadecimal color value", "background color"}
  };
}
```

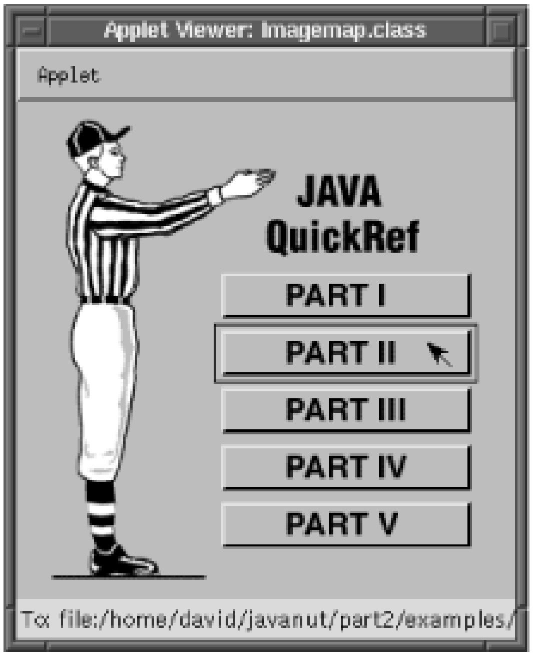
The following HTML fragment references the applet, and demonstrates how parameter values can be set with the <PARAM> tag:

```
<APPLET code="ColorScribble.class" width=300 height=300>
<PARAM name="foreground" value="0000FF">
<PARAM name="background" value="FFCCCC">
</APPLET>
```

6.6 Images and Sounds

Example 6.5 shows a Java applet that implements a simple client-side imagemap, which has the capability to highlight the "hot spots" in the image and play a sound clip when the user clicks on the image. Figure 6.4 shows what this applet might look like, when configured with an appropriate image.

Figure 6.4: An imagemap applet


Applet Viewer: Imagemap.class
Applet
JAVA
QuickRef
PART I
PART II
PART III
PART IV
PART V
To: file:/home/david/javanut/part2/examples/

This applet demonstrates quite a few important applet techniques:

- `getParameter()` looks up the name of the image to display and the audio clip to play when the user clicks, and it also reads a list of rectangles and URLs that define the hot spots and hyperlinks of the imagemap.
- The `getImage()` and `getDocumentBase()` methods load the image in the `init()` method, and `Graphics.drawImage()` displays the image in the `paint()` method.
- `getAudioClip()` loads a sound file in the `init()` method, and `AudioClip.play()` plays the sound in the `mousePressed()` method.

- Events are handled by an "event listener" object, which is defined by an inner class (see Chapter 5, *Inner Classes and Other New Language Features*). This is an example of the Java 1.1 event handling model (see Chapter 7, *Events*). Therefore, this applet only runs in Web browsers that support Java 1.1.

- The `showStatus()` method displays the destination URL when the user presses the mouse button over a hot spot, while the `AppletContext.showDocument()` method makes the browser display that URL when the user releases the mouse button.

- The applet uses "XOR mode" of the `Graphics` class to highlight an area of the image in a way that can be easily "un-highlighted" by redrawing.

- The individual hot spots are represented by instances of a nested top-level class. The `java.util.Vector` class stores the list of hot spot objects, and `java.util.StringTokenizer` parses the descriptions of those hot spots.

The following HTML fragment shows an example of the properties read by this applet:

```
<APPLET code="Soundmap.class" width=288 height=288>
<PARAM name="image" value="image.gif">
<PARAM name="sound" value="sound.au">
<PARAM name="rect0" value="114,95,151,33,#part1">
<PARAM name="rect1" value="114,128,151,33,#part2">
<PARAM name="rect2" value="114,161,151,33,#part3">
<PARAM name="rect3" value="114,194,151,33,#part4">
<PARAM name="rect4" value="114,227,151,33,#part5">
</APPLET>
```

Example 6.5: An Imagemap Applet

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.util.*;
/**
 * A Java applet that simulates a client-side imagemap.
 * Plays a sound whenever the user clicks on one of the hyperlinks.
 */
public class Soundmap extends Applet {
  protected Image image;      // The image to display.
  protected Vector rects;     // A list of rectangles in it.
  protected AudioClip sound;  // A sound to play on user clicks in a rectangle.
  /** Initialize the applet. */
  public void init() {
    // Look up the name of the image, relative to a base URL, and load it.
    // Note the use of three Applet methods in this one line.
    image = this.getImage(this.getDocumentBase(), this.getParameter("image"));
    // Lookup and parse a list of rectangular areas and the URLs they map to.
    // The convenience routine getRectangleParameter() is defined below.
    rects = new Vector();
    ImagemapRectangle r;
    for(int i = 0; (r = getRectangleParameter("rect" + i)) != null; i++)
      rects.addElement(r);
    // Look up a sound to play when the user clicks one of those areas.
    sound = this.getAudioClip(this.getDocumentBase(),
                  this.getParameter("sound"));
    // Specify an "event listener" object to respond to mouse button
    // presses and releases.  Note that this is the Java 1.1 event model.
    // Note that it also uses a Java 1.1 inner class, defined below.
    this.addMouseListener(new Listener());
  }
  /** Called when the applet is being unloaded from the system.
   * We use it here to "flush" the image we no longer need. This may
   * result in memory and other resources being freed more quickly. */
  public void destroy() { image.flush(); }
  /** To display the applet, we simply draw the image. */
  public void paint(Graphics g) { g.drawImage(image, 0, 0, this); }
  /** We override this method so that it doesn't clear the background
   * before calling paint().  No clear is necessary, since paint() overwrites
   * everything with an image.  Causes less flickering this way. */
  public void update(Graphics g) { paint(g); }
  /** Parse a comma-separated list of rectangle coordinates and a URL.
   * Used to read the imagemap rectangle definitions from applet parameters. */
  protected ImagemapRectangle getRectangleParameter(String name) {
    int x, y, w, h;
    URL url;
    String value = this.getParameter(name);
    if (value == null) return null;
    try {
      StringTokenizer st = new StringTokenizer(value, ",");
      x = Integer.parseInt(st.nextToken());
      y = Integer.parseInt(st.nextToken());
      w = Integer.parseInt(st.nextToken());
      h = Integer.parseInt(st.nextToken());
      url = new URL(this.getDocumentBase(), st.nextToken());
    }
    catch (NoSuchElementException e) { return null; }
    catch (NumberFormatException e) { return null; }
    catch (MalformedURLException e) { return null; }
    return new ImagemapRectangle(x, y, w, h, url);
  }
  /**
   * An instance of this inner class is used to respond to mouse events.
   */
  class Listener extends MouseAdapter {
```

```
  /** The rectangle that the mouse was pressed in. */
  private ImagemapRectangle lastrect;
  /** Called when a mouse button is pressed. */
  public void mousePressed(MouseEvent e) {
    // On button down, check if we're inside one of the specified rectangles.
    // If so, highlight the rectangle, display a message, and play a sound.
    // The utility routine findrect() is defined below.
    ImagemapRectangle r = findrect(e);
    if (r == null) return;
    Graphics g = Applet.this.getGraphics();
    g.setXORMode(Color.red);
    g.drawRect(r.x, r.y, r.width, r.height);  // highlight rectangle
    Applet.this.showStatus("To: " + r.url);   // display URL
    sound.play();                             // play the sound
    lastrect = r;   // Remember the rectangle so it can be un-highlighted.
  }
  /** Called when a mouse button is released. */
  public void mouseReleased(MouseEvent e) {
    // When the button is released, un-highlight the rectangle.  If the
    // mouse is still inside it, ask the browser to go to the URL.
    if (lastrect != null) {
  Graphics g = Applet.this.getGraphics();
  g.setXORMode(Color.red);
  g.drawRect(lastrect.x, lastrect.y, lastrect.width, lastrect.height);
  Applet.this.showStatus("");   // Clear the message.
  ImagemapRectangle r = findrect(e);
  if ((r != null) && (r == lastrect))  // If still in the same rectangle
    Applet.this.getAppletContext().showDocument(r.url); // Go to the URL
  lastrect = null;
    }
  }
  /** Find the rectangle we're inside. */
  protected ImagemapRectangle findrect(MouseEvent e) {
    int i, x = e.getX(), y = e.getY();
    for(i = 0; i < rects.size(); i++)  {
  ImagemapRectangle r = (ImagemapRectangle) rects.elementAt(i);
  if (r.contains(x, y)) return r;
    }
    return null;
  }
}
/**
 * A helper class.  Just like java.awt.Rectangle, but with a URL field.
 * Note the use of a nested top-level class for neatness.
 */
static class ImagemapRectangle extends Rectangle {
  URL url;
  public ImagemapRectangle(int x, int y, int w, int h, URL url) {
    super(x, y, w, h);
```

```
      this.url = url;
    }
  }
}
```

6.7 JAR Files

The Soundmap applet defined in the previous section requires five files to operate: the class file for the applet itself, the class files for the two nested classes it contains, the image file, and the sound clip file. It can be loaded using an <APPLET> tag like this:

```
<APPLET code="Soundmap.class" width=288 height=288>
   ...
</APPLET>
```

When the applet is loaded in this manner, however, each of the five files is transferred in uncompressed form using a separate HTML request. As you might imagine, this is quite inefficient.

In Java 1.1, you can instead combine the five files into a single JAR (Java ARchive) file. This single, compressed file (it is a ZIP file) can be transferred from Web server to browser much more efficiently. To create a JAR file, use the *jar* tool, which has a syntax reminiscent of the UNIX *tar* command:

```
% jar cf soundmap.jar *.class image.gif sound.au
```

This command creates a new file, *soundmap.jar*, that contains all the *.class* files in the current directory, and also contains the files *image.gif* and *sound.au*. *jar* can also be used to list and extract the contents of a JAR file. See Chapter 16, *JDK Tools* for complete documentation.

To use a JAR file, you specify it as the value of the archive attribute of the <APPLET> tag:

```
<APPLET archive="soundmap.jar" code="Soundmap.class" width=288 height=288>
   ...
</APPLET>
```

Note that the archive attribute does not replace the code attribute. archive specifies where to look for files, but code is still required to tell the browser which of the files in the archive is the applet class file to be executed. The archive attribute may actually specify a comma-separated list of JAR files. The Web browser or applet viewer searches these archives for any files the applet requires. If a file is not found in an archive, however, the browser falls back upon its old behavior and attempts to load the file from the Web server using a new HTTP request.

PREVIOUS	HOME	NEXT
Images and Sounds	BOOK INDEX	Applet Security Restrictions

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

7. Events

Contents:

The heart of any applet or graphical user interface is the event processing code. Graphical applications are event-driven: they do nothing until the user moves the mouse or clicks a button or types a key. An event-driven program is structured around its event-processing model, so a solid understanding of event handling mechanisms is crucial for good programming.

Unfortunately, the Java event handling model has changed between Java 1.0 and Java 1.1. The Java 1.0 model is a simple one, well suited to writing basic applets. It has a number of shortcomings, however, and does not scale well to complicated interfaces. Although the 1.0 event model is deprecated in Java 1.1, you'll still need to use it for any applets that you want to run on Web browsers based on Java 1.0. The Java 1.1 event model solves many of the shortcomings of the 1.0 model it replaces, but would be quite cumbersome to use if it were not for the new inner class features also introduced in Java 1.1. This chapter covers both event models and provides examples of each.

7.1 The Java 1.0 Event Model

In Java 1.0, all events are represented by the `Event` class. This class has a number of instance variables that describe the event. One of these variables, `id`, specifies the type of the event. `Event` defines a number of constants that are the possible values for the `id` field. The `target` field specifies the object (typically a `Component`) that generated the event, or on which the event occurred (i.e., the source of the event). The other fields may or may not be used, depending on the type of the event. For example, the `x` and `y` fields are defined when `id` is `BUTTON_EVENT`, but not when it is `ACTION_EVENT`. The `arg` field can provide additional type-dependent data.

Java 1.0 events are dispatched first to the `handleEvent()` method of the `Component` on which they occurred. The default implementation of this method checks the `id` field of the `Event` object and dispatches the most commonly used types of events to various type-specific methods, listed in Table 7.1.

Table 7.1: Java 1.0 Event Processing Methods of Component

`action()`	`lostFocus()`	`mouseExit()`
`gotFocus()`	`mouseDown()`	`mouseMove()`
`keyDown()`	`mouseDrag()`	`mouseUp()`
`keyUp()`	`mouseEnter()`	

The methods listed in Table 7.1 are defined by the `Component` class. One of the primary characteristics of the Java 1.0 event model is that you must override these methods in order to process events. This means that you must create a subclass to define custom event-handling behavior, which is exactly what we do when we write an applet, for example. Notice, however, that not all of the event types are dispatched by `handleEvent()` to more specific methods. So, if you are interested in `LIST_SELECT` or `WINDOW_ICONIFY` events, for example, you have to override `handleEvent()` itself, rather than one of the more specific methods. If you do this, you should usually invoke `super.handleEvent()` to continue dispatching events of other types in the default way.

The `handleEvent()` method, and all of the type-specific methods, return `boolean` values. If an event-handling method returns `false`, as they all do by default, it means that the event was not handled, so it should be passed to the container of the current component to see if that container is interested in processing it. If a method returns `true`, on the other hand, it is a signal that the event *has* been handled and no further processing is needed.

The fact that unhandled events are passed up the containment hierarchy is important. It means that we can override the `action()` method (for example) in an applet in order to handle the `ACTION_EVENT` events that are generated by the buttons within the applet. If they were not propagated up as they are, we would have to create a custom subclass of `Button` for every button we wanted to add to an interface!

In programs that use the Java 1.0 event model, it is typical to handle events at the top-level component. In an applet, for example, you override the `handleEvent()` method, or some of the other type-specific methods, of the `Applet` subclass you create. Or, in a stand-alone program that creates its own window, you subclass `Frame` to provide definitions of the event-handling methods. When a program displays a dialog box, it subclasses `Dialog` to define the methods. With complex interfaces, the event-handling methods of the containers at the top of the hierarchy can become long and somewhat convoluted, so you need to be careful when writing them.

Components and Their Events

In the Java 1.0 model, there is no defacto way to know what types of events are generated by what GUI components. You simply have to look this information up in the documentation. Additionally, different components use different fields of the `Event` object, and pass different values in the `arg` field of that object. Table 7.2 lists each of the AWT components, and for each one, lists the type of events it generates. The first column of the table specifies both the type of the component and the type of the event. The event type is the constant stored in the `id` field of the `Event`.

The second through sixth columns indicate whether the `when` (timestamp), `x` (mouse x coordinate), `y` (mouse y coordinate), `key` (the key that was pressed), and `modifiers` (modifier keys that were down) fields are set for a given event. If a dot appears in this column, the event sets a value for the corresponding field. The seventh column explains what occurred to trigger the event, and what the value of the `arg` field of the `Event` object is.

Events listed for the `Component` component type apply to all `java.awt Component` subclasses. The events listed for the `Window` component type also apply to the `Window` subclasses, `Dialog` and `Frame`.

Table 7.2: AWT Components and the Java 1.0 Events They Generate

Component	**when**	**x**	**y**	**key**	**mods**	**Event Meaning**
Event Type (`id`)						**`arg` (Type: value)**
`Button`						User clicked on the button
`ACTION_EVENT`						`String`: the button label
`Checkbox`						User clicked on checkbox
`ACTION_EVENT`						`Boolean`: new checkbox state
`Choice`						User selected an item
`ACTION_EVENT`						`String`: label of selected item
`Component`						Got input focus
`GOT_FOCUS`						*unused*
`Component`	*	*	*	*	*	User pressed a function key
`KEY_ACTION`						*unused*--`key` contains key
						constant

Component	*	*	*	*	*	User released a function key
KEY_ACTION_RELEASE						*unused*--key contains key
						constant
Component	*	*	*	*	*	User pressed a key
KEY_PRESS						*unused*--key contains ASCII
						key value
Component	*	*	*	*	*	User released a key
KEY_RELEASE						*unused*--key contains ASCII
						key value
Component						Lost input focus
LOST_FOCUS						*unused*
Component	*	*	*			Mouse entered the Component
MOUSE_ENTER						*unused*
Component	*	*	*			Mouse left the Component
MOUSE_EXIT						*unused*
Component	*	*	*		*	User pressed mouse button
MOUSE_DOWN						*unused*
Component	*	*	*		*	User released mouse button
MOUSE_UP						*unused*
Component	*	*	*		*	User moved mouse
MOUSE_MOVE						*unused*
Component	*	*	*		*	User dragged mouse
MOUSE_DRAG						*unused*
List						User double-clicked on an item
ACTION_EVENT						String: label of activated item
List						User selected an item
LIST_SELECT						Integer: index of selected item
List						User deselected an item
LIST_DESELECT						Integer: index of deselected item
MenuItem						User selected an item
ACTION_EVENT						String: label of selected item

Scrollbar						User requested scroll
SCROLL_LINE_UP						Integer: position to scroll to
Scrollbar						User requested scroll
SCROLL_LINE_DOWN						Integer: position to scroll to
Scrollbar						User requested scroll
SCROLL_PAGE_UP						Integer: position to scroll to
Scrollbar						User requested scroll
SCROLL_PAGE_DOWN						Integer: position to scroll to
Scrollbar						User requested scroll
SCROLL_ABSOLUTE						Integer: position to scroll to
TextField						User struck <Return>
ACTION_EVENT						String: user's input text
Window						Window was destroyed
WINDOW_DESTROY						*unused*
Window						Window was iconified
WINDOW_ICONIFY						*unused*
Window						Window was deiconified
WINDOW_DEICONIFY						*unused*
Window		*	*			Window was moved
WINDOW_MOVED						*unused*

Key and Modifier Constants

The java.awt.Event class contains the field key, which is filled in when a keyboard event has occurred, and the field modifiers, which list the keyboard modifier keys currently in effect for key and mouse events.

Four modifier constants are defined by the java.awt.Event class; they are listed in Table 7.3. They are mask values that are ORed into the modifiers field. You can test for them using AND. You can also check a given event for the first three of the modifiers with the Event methods shiftDown(), controlDown(), and metaDown().

Table 7.3: Java Keyboard Modifiers

Modifier Constant	**Meaning**

`Event.SHIFT_MASK`	SHIFT key is held down (or CAPS LOCK on)
`Event.CTRL_MASK`	CONTROL key is held down
`Event.META_MASK`	META key is held down
`Event.ALT_MASK`	ALT key is held down

When a `KEY_PRESS` or `KEY_RELEASE` event occurs, it means that the user pressed a key that is a normal printing character, a control character, or a non-printing character with a standard ASCII value--one of RETURN (ASCII 10 or '`\n`'), TAB (ASCII 9 or '`\t`'), ESCAPE (ASCII 27), BACKSPACE (ASCII 8), or DELETE (ASCII 127). In this case, the value of the `key` field in the event is simply the ASCII value of the key that was pressed or released.

When a `KEY_ACTION` or `KEY_ACTION_RELEASE` event occurs, it means that the user pressed some sort of function key, one which does not have an ASCII representation.

`java.awt.Event` defines constants for each of these function keys, which are listed in Table 7.4.

Table 7.4: Java Function Key Constants

Key Constant	**Meaning**
`Event.HOME`	HOME key
`Event.END`	END key
`Event.PGUP`	PAGE UP key
`Event.PGDN`	PAGE DOWN key
`Event.UP`	UP arrow key
`Event.DOWN`	DOWN arrow key
`Event.LEFT`	LEFT arrow key
`Event.RIGHT`	RIGHT arrow key
`Event.F1 to Event.F12`	Function keys 1 through 12

Mouse Buttons

In order to maintain platform independence, Java only recognizes a single mouse button--the `Event` class does not have any kind of `mouseButton` field to indicate which button has been pressed on a multi-button mouse.

On platforms that support two- or three-button mouses, the right and center buttons generate mouse down, mouse drag, and mouse up events as if the user were holding down modifier keys, as shown in

Table 7.5.

Table 7.5: Mouse Button Modifiers

Mouse Button	Flag Set in Event.modifiers Field
Left button	*none*
Right button	`Event.META_MASK`
Middle button	`Event.ALT_MASK`

Using keyboard modifiers to indicate the mouse button that has been pressed maintains compatibility with platforms that only have one-button mouses, but still allows programs to use the right and middle buttons on platforms that support them. Suppose, for example, you want to write a program that allows the user to draw lines with the mouse using two different colors. You might draw in the primary color if there are no modifier flags set, and draw in the secondary color when the `META_MASK` modifier is set. In this way, users with a two- or three-button mouse can simply use the left and right mouse buttons to draw in the two colors; and users with a one-button mouse can use the META key, in conjunction with the mouse, to draw in the secondary color.

6.8 Applet Security Restrictions

Applets loaded over the network are usually considered to be untrusted code. (The exception, as we'll see in the next section, is when the applet bears the digital signature of an entity that you've specified you trust.) The only way to be sure that an untrusted applet cannot perform any malicious actions (e.g., deleting your files, sending out fake email that looks like it came from you, using your computer as a remote file server) is to run it in a tightly controlled environment. For this reason, Web browsers and applet viewers carefully restrict what an applet is allowed to do. When designing an applet, you must be aware of a fairly long list of things that an applet is not allowed to do. The following list details the security restrictions imposed by the *appletviewer* application in Java 1.1. Different Web browsers and applet viewers may impose somewhat different restrictions on applets, and some (including *appletviewer*) may allow the user to relax or customize the restrictions. In general, however, you should assume that your applets will be restricted in the following ways:

- Untrusted code cannot read from or write to the local filesystem. This means that untrusted code cannot:
    - Read files.
    - List directories.
    - Check for the existence of files.
    - Obtain the size or modification date of files.
    - Obtain the read and write permissions of a file.
    - Test whether a filename is a file or directory.
    - Write files.

    - Delete files.

    - Create directories.

    - Rename files.

    - Read or write from `FileDescriptor` objects.

- *appletviewer* allows a system administrator to set properties that allow applets to read and write files within a specified list of directories.

  Untrusted code cannot perform networking operations, except in certain restricted ways. Untrusted code cannot:

    - Create a network connection to any computer other than the one from which the code was itself loaded.

    - Listen for network connections on any of the privileged ports with numbers less than or equal to 1024.

    - Accept network connections on ports less than or equal to 1024 or from any host other than the one from which the code itself was loaded.

    - Use multicast sockets.

    - Create or register a `SocketImplFactory`, `URLStreamHandlerFactory`, or `ContentHandlerFactory`.

- *appletviewer* uses the "host-of-origin" policy described above by default, but can also be configured to disallow all networking or to allow all networking.

  Untrusted code cannot make use of certain system facilities. It cannot:

    - Exit the Java interpreter by calling `System.exit()` or `Runtime.exit()`.

    - Spawn new processes by calling any of the `Runtime.exec()` methods.

    - Dynamically load native code libraries with the `load()` or `loadLibrary()` methods of `Runtime` or `System`.

- Untrusted code cannot make use of certain AWT facilities. One major restriction is that all windows created by untrusted code will display a prominent visual indication that they have been created by untrusted code and are "insecure." This is to prevent untrusted code from spoofing the on-screen appearance of trusted code. Additionally, untrusted code cannot:

  - Initiate a print job.
  - Access the system clipboard.
  - Access the system event queue.

- Untrusted code has restricted access to system properties. It cannot call `System.getProperties()`, and so cannot modify or insert properties into the system properties list. It can call `System.getProperty()` to read individual properties, but can only read system properties to which it has been explicitly granted access. By default, *appletviewer* grants access to only the following ten properties. Note that `user.home` and `user.dir` are excluded:

  - `java.version`
  - `java.class.version`
  - `java.vendor`
  - `java.vendor.url`
  - `os.name`
  - `os.version`
  - `os.arch`
  - `file.separator`
  - `path.separator`
  - `line.separator`

- Untrusted code cannot create or access threads or thread groups outside of the thread group in which the untrusted code is running.

- Untrusted code has restrictions on the classes it can load and define. It cannot:

    - Explicitly load classes from the `sun.*` packages.
    - Define classes in any of the `java.*` or `sun.*` packages.
    - Create a `ClassLoader` object or call any `ClassLoader` methods.

- Untrusted code cannot use the `java.lang.Class` reflection methods to obtain information about non-public members of a class, unless the class was loaded from the same host as the untrusted code.
- Untrusted code has restrictions on its use of the `java.security` package. It cannot:
    - Manipulate security identities in any way.
    - Set or read security properties.
    - List, lookup, insert, or remove security providers.
    - Finally, to prevent untrusted code from circumventing all of these restrictions, it is not allowed to create or register a `SecurityManager` object.

Local Applet Restrictions

When an applet is loaded from the local file system, instead of through a network protocol, Web browsers and applet viewers may relax some, or even many, of the above restrictions. The reason for this is that local applets are assumed to be more trustworthy than anonymous applets from the network.

Intermediate applet security policies are also possible. For example, an applet viewer could be written that would place fewer restrictions on applets loaded from an internal corporate network than on those loaded from the Internet.

Applet Security Implementation

Implementing the security restrictions described above is the responsibility of the `java.lang.SecurityManager` class. This class defines a number of methods that the system calls to check whether a certain operation (such as reading a file) is permitted in the current environment. Applet viewers create a subclass of `SecurityManager` to implement a particular security policy. A security policy is put in place by instantiating a `SecurityManager` object and registering it with `System.setSecurityManager()`. (One of the obvious security restrictions that must be enforced is that untrusted code may not register its own `SecurityManager` object!)

Loading Classes Securely

Another component of Java security is the way Java classes are loaded over the network. The `java.lang.ClassLoader` class defines how this is done. Applet viewers and Web browsers create subclasses of this class that implement security policies and define how class files are loaded via various protocols.

One important function of the class loader is to ensure that loaded classes reside in a separate namespace than classes loaded from the local system. This prevents naming conflicts, and also prevents a malicious applet from replacing standard Java classes with its own versions.

Byte-Code Verification

Another important function of the class loader is to ensure that all untrusted Java code (generally code loaded over the network) is passed through the Java byte-code verification process. This process ensures that the loaded code does not violate Java namespace restrictions or type conversion restrictions. It also checks that the code:

- Is valid Java Virtual Machine code.
- Does not overflow or underflow the stack.
- Does not use registers incorrectly.
- Does not convert data types illegally.

The purpose of these checks is to verify that the loaded code cannot forge pointers or do memory arithmetic, which could give it access to the underlying machine. The checks also ensure that the code cannot crash the Java interpreter or leave it in an undefined state, which might allow malicious code to take advantage of security flaws that could exist in some interpreter implementations. Essentially, the byte-code verification process protects against code from an "untrusted" Java compiler.

Denial of Service Attacks

The one "security hole" that remains when running an untrusted applet is that the applet can perform a "denial of service attack" on your computer. For example, it could frivolously allocate lots of memory, run many threads, or download lots of data. This sort of attack consumes system resources and can slow your computer (or your network connection) considerably. While this sort of attack by an applet is inconvenient, fortunately it cannot usually do any significant damage.

PREVIOUS
JAR Files

HOME
BOOK INDEX

NEXT
Signed Applets

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

16. JDK Tools

Contents:

appletviewer

Name

appletviewer---The Java Applet Viewer

Availability

JDK 1.0 and later.

Synopsis

```
appletviewer [-debug]  [-Jjavaoption] [-encoding enc] url|file...
```

Description

appletviewer reads or downloads one or more HTML documents specified by filename or URL on the command line. It reads or downloads all the applets referenced in each document and displays them, each in their own window. If none of the named documents has an `<APPLET>` tag, *appletviewer* does nothing.

Options

`-debug`

> If this option is specified, the *appletviewer* is started within *jdb* (the Java debugger). This allows you to debug the applets referenced by the document or documents.

`-J`*`javaoption`*

> This option passes the following *`javaoption`* as a command-line argument to the Java interpreter. The specified *`javaoption`* should not contain spaces. If a multi-word option must be passed to the Java interpreter, multiple `-J` options should be used. See *java* for a list of valid Java interpreter options. Available in JDK 1.1 and later.

`-encoding`*`enc`*

> This option specifies the character encoding that *appletviewer* should use when reading the contents of the specified files or URLs. It is used in the conversion of applet parameter values to Unicode. Available in JDK 1.1 and later.

Commands

The window displayed by *appletviewer* contains a single **Applet** menu, with the following commands available:

Restart

> Stops and destroys the current applet, then re-initializes and restarts it.

Reload

Stops, destroys, and unloads the applet, then reloads, reinitializes, and restarts it.

Stop

Stops the current applet. Available in Java 1.1 and later.

Save

Serializes the applet and saves the serialized applet in the file *Applet.ser* in the user's home directory. The applet should be stopped before selecting this option. Available in Java 1.1 and later.

Start

Restarts a stopped applet. Available in Java 1.1 and later.

Clone

Creates a new copy of the applet in a new *appletviewer* window.

Tag

Pops up a dialog box that displays the `<APPLET>` tag and all associated `<PARAM>` tags that created the current applet.

Info

Pops up a dialog box that contains information about the applet. This information is provided by the `getAppletInfo()` and `getParameterInfo()` methods implemented by the applet.

Edit

This command is not implemented. The **Edit** menu item is disabled.

Character Encoding

Displays the current character encoding in the status line. Available in Java 1.1 and later.

Print

Prints the applet. Available in Java 1.1 and later.

Properties

Displays a dialog that allows the user to set *appletviewer* preferences, including settings for firewall and caching proxy servers.

Close

Closes the current *appletviewer* window.

Quit

Quits *appletviewer*, closing all open windows.

Properties

When it starts up, *appletviewer* reads property definitions from the file *~/.hotjava/properties* (UNIX) or the *.hotjava\properties* file relative to the `HOME` environment variable (Windows). These properties are stored in the system properties list and are used to specify the various error and status messages the applet viewer displays, as well as its security policies and use of proxy servers. The properties that affect security and proxies are described below.

Security

The following properties specify the security restrictions that *appletviewer* places on untrusted applets:

`acl.read`

This is a list of files and directories that an untrusted applet is allowed to read. The elements of the list should be separated with colons on UNIX systems and semicolons on Windows systems. On UNIX systems, the ~ character is replaced with the home directory of the current user. If the plus character appears as an element in the list, it is replaced by the value of the `acl.read.default` property. This provides an easy way to enable read access--by simply setting `acl.read` to "+". By default, untrusted applets are not allowed to read any files or directories.

`acl.read.default`

This is a list of files and directories that are readable by untrusted applets if the `acl.read` property contains a plus character.

acl.write

This is a list of files and directories that an untrusted applet is allowed to write to. The elements of the list should be separated with colons on UNIX systems and semicolons on Windows systems. On UNIX systems, the ~ character is replaced with the home directory of the current user. If the plus character appears as an element in the list, it is replaced by the value of the `acl.write.default` property. This provides an easy way to enable write access--by simply setting `acl.write` to "+". By default, untrusted applets are not allowed to write to any files or directories.

acl.write.default

This is a list of files and directories that are writable by untrusted applets if the `acl.write` property contains a plus character.

appletviewer.security.mode

This property specifies the types of network access an untrusted applet is allowed to perform. If it is set to "none", then the applet can perform no networking at all. The value "host" is the default, and specifies that the applet can connect only to the host from which it was loaded. The value "unrestricted" specifies that an applet may connect to any host without restrictions.

package.restrict.access.package-prefix

Properties of this form may be set to `true` to prevent untrusted applets from using classes in any package that has the specified package name prefix as the first component of its name. For example, to prevent applets from using any of the Sun classes (such as the Java compiler and the appletviewer itself) that are shipped with the JDK, you could specify the following property:

```
package.restrict.access.sun=true
```

appletviewer sets this property to `true` by default for the `sun.*` and `netscape.*` packages.

package.restrict.definition.package-prefix

Properties of this form may be set to `true` to prevent untrusted applets from defining classes in a package that has the specified package name prefix as the first component of its name. For example, to prevent an applet from defining classes in any of the standard Java packages, you could specify the following property:

```
package.restrict.definition.java=true
```

appletviewer sets this property to `true` by default for the `java.*`, `sun.*`, and `netscape.*` packages.

`property.applet`

When a property of this form is set to `true` in Java 1.1, it specifies that an applet should be allowed to read the property named `property` from the system properties list. By default, applets are only allowed to read ten standard system properties (see Chapter 14, *System Properties*, for a list). For example, to allow an applet to read the `user.home` property, specify a property of the form

```
user.home.applet=true
```

Proxies

appletviewer uses the following properties to configure its use of firewall and caching proxy servers:

`firewallHost`

This is the firewall proxy host to connect to if the `firewallSet` property is `true`.

`firewallPort`

This is the port of the firewall proxy host to connect to if the `firewallSet` property is `true`.

`firewallSet`

This tells you whether the applet viewer should use a firewall proxy. Values are `true` or `false`.

`proxyHost`

This is the caching proxy host to connect to if the `proxySet` property is `true`.

`proxyPort`

This is the port of the caching proxy host to connect to if the `proxySet` property is `true`.

`proxySet`

This tells you whether the applet viewer should use a caching proxy. Values are `true` or `false`.

Environment

`CLASSPATH`

Specifies an ordered list (colon-separated on UNIX, semicolon-separated on Windows systems) of directories and ZIP files in which *appletviewer* should look for class definitions. When a path is specified with this environment variable, *appletviewer* always implicitly appends the location of the system classes to the end of the path. If this environment variable is not specified, the default path is the current directory and the system classes. Note that *appletviewer* does not support the `-classpath` command-line argument, except indirectly through the `-J` option.

See Also

java, *javac*, *jdb*

6.9 Signed Applets

In Java 1.1 it is possible to circumvent these applet security restrictions by attaching a digital signature to a JAR file. When a Web browser or applet viewer loads a JAR file that has been signed by a trusted entity (the user specifies whom she trusts), the browser may grant the applet contained in the JAR file special privileges, such as the ability to read and write local files, that are not available to untrusted applets.

Signing an applet with the *javakey* tool provided by the JDK is a somewhat cumbersome task. First, of course, you must have a security database set up. The database must contain the certificate and the public and private keys that you want to use to sign the applet. See the *javakey* documentation in Chapter 16, *JDK Tools* for details on this process.

Once you have a properly configured security database, you must create a simple "directive file" that gives *javakey* the information it needs to sign your JAR file. A directive file might look like this:

```
# The entity doing the signing
signer=david
# The certificate number to use
cert=1
# Currently unused
chain=0
# The base name of the signature files to be added to the JAR manifest
signature.file=DAVIDSIG
```

Having created a directive file named *mysig*, for example, you could then sign a JAR file like this:

```
% javakey -gs mysig soundmap.jar
```

This command creates a new JAR file named *soundmap.jar.sig* that you can use in an HTML `archive` attribute just as you would use an unsigned JAR file.

The *javakey* tool is used for all aspects of administering the Java system security database. One of the other things you can do with it is to declare which entities are trusted. You do this with the `-t` option. For example, you might declare your trust for the author as follows:

```
% javakey -t DavidFlanagan true
```

Or you could revoke your trust like this:

```
% javakey -t DavidFlanagan false
```

The *appletviewer* program makes use of any trust values declared in this way. Note that *javakey* and *appletviewer* support only untrusted entities and fully-trusted entities, without any gradations in between. We may see additional levels of trust in the future.

Bear in mind that the *javakey* techniques described here apply only to the JDK. Other vendors may provide other mechanisms for signing applets, and Web browsers may provide other ways of declaring trusted entities.

7.2 Scribbling in Java 1.0

Example 7.1 shows a simple applet that uses the Java 1.0 event model. It overrides the `mouseDown()` and `mouseDrag()` methods to allow the user to scribble with the mouse. It overrides the `keyDown()` method and clears the screen when it detects the "C" key. And it overrides the `action()` method to clear the screen when the user clicks on a **Clear** button. We've seen applets much like this elsewhere in the book; this one is not pictured here.

Example 7.1: Scribble: Using the 1.0 Event Model

```
import java.applet.*;
import java.awt.*;
/** A simple applet using the Java 1.0 event handling model */
public class Scribble1 extends Applet {
  private int lastx, lasty;    // Remember last mouse coordinates.
  Button clear_button;         // The Clear button.
  Graphics g;                  // A Graphics object for drawing.
  /** Initialize the button and the Graphics object. */
  public void init() {
    clear_button = new Button("Clear");
    this.add(clear_button);
    g = this.getGraphics();
  }
  /** Respond to mouse clicks. */
  public boolean mouseDown(Event e, int x, int y) {
    lastx = x; lasty = y;
    return true;
  }
  /** Respond to mouse drags. */
  public boolean mouseDrag(Event e, int x, int y) {
    g.setColor(Color.black);
    g.drawLine(lastx, lasty, x, y);
    lastx = x; lasty = y;
    return true;
  }
  /** Respond to key presses. */
  public boolean keyDown(Event e, int key) {
    if ((e.id == Event.KEY_PRESS) && (key == 'c')) {
      clear();
      return true;
    }
    else return false;
  }
  /** Respond to Button clicks. */
  public boolean action(Event e, Object arg) {
    if (e.target == clear_button) {
      clear();
      return true;
    }
    else return false;
  }
  /** convenience method to erase the scribble */
  public void clear() {
    g.setColor(this.getBackground());
    g.fillRect(0, 0, bounds().width, bounds().height);
  }
}
```

PREVIOUS | HOME | NEXT
The Java 1.0 Event Model | BOOK INDEX | The Java 1.1 Event Model

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

7.3 The Java 1.1 Event Model

The Java 1.1 event model is used by both the AWT and by Java Beans. In this model, different classes of events are represented by different Java classes. Every event is a subclass of `java.util.EventObject`. AWT events, which is what we are concerned with here, are subclasses of `java.awt.AWTEvent`. For convenience, the various types of AWT events, such as `MouseEvent` and `ActionEvent`, are placed in the new `java.awt.event` package.

Every event has a source object, which can be obtained with `getSource()`, and every AWT event has a type value, which can be obtained with `getID()`. This value is used to distinguish the various types of events that are represented by the same event class. For example, the `FocusEvent` has two possible types: `FocusEvent.FOCUS_GAINED` and `FocusEvent.FOCUS_LOST`. Event subclasses contain whatever data values are pertinent to the particular event type. For example, `MouseEvent` has `getX()`, `getY()`, and `getClickCount()` methods; it also inherits the `getModifiers()` and `getWhen()` methods, among others.

The 1.1 event handling model is based on the concept of an "event listener." An object interested in receiving events is an *event listener*. An object that generates events (an *event source*) maintains a list of listeners that are interested in being notified when events occur, and provides methods that allow listeners to add themselves and remove themselves from this list of interested objects. When the event source object generates an event (or when a user input event occurs on the event source object), the event source notifies all the listener objects that the event has occurred.

An event source notifies an event listener object by invoking a method on it and passing it an event object (an instance of a subclass of `EventObject`). In order for a source to invoke a method on a listener, all listeners must implement the required method. This is ensured by requiring that all event listeners for a particular type of event implement a corresponding interface. For example, event listener objects for `ActionEvent` events must implement the `ActionListener` interface. The `java.awt.event` package defines an event listener interface for each of the event types it defines. (Actually, for `MouseEvent` events, it defines two listener interfaces: `MouseListener` and `MouseMotionListener`.) All event listener interfaces themselves extend `java.util.EventListener`. This interface does not define any methods, but instead acts as a marker interface, clearly identifying all event listeners as such.

An event listener interface may define more than one method. For example, an event class like `MouseEvent` represents several different types of mouse events, such as a button press event and a button release event, and these different event types cause different methods in the corresponding event listener to be invoked. By convention, the methods of an event listener are passed a single argument, which is an event object of the type that corresponds to the listener. This event object should contain all the information a program needs to respond to the event. Table 7.6 lists the event types defined in `java.awt.event`, the corresponding listener (or listeners), and the methods defined by each listener interface.

Table 7.6: Java 1.1 Event Types, Listeners, and Listener Methods

Event Class	**Listener Interface**	**Listener Methods**
`ActionEvent`	`ActionListener`	`actionPerformed()`
`AdjustmentEvent`	`AdjustmentListener`	`adjustmentValueChanged()`
`ComponentEvent`	`ComponentListener`	`componentHidden()`
		`componentMoved()`
		`componentResized()`
		`componentShown()`
`ContainerEvent`	`ContainerListener`	`componentAdded()`
		`componentRemoved()`
`FocusEvent`	`FocusListener`	`focusGained()`
		`focusLost()`
`ItemEvent`	`ItemListener`	`itemStateChanged()`
`KeyEvent`	`KeyListener`	`keyPressed()`
		`keyReleased()`
		`keyTyped()`
`MouseEvent`	`MouseListener`	`mouseClicked()`
		`mouseEntered()`
		`mouseExited()`
		`mousePressed()`
		`mouseReleased()`
	_	_
	`MouseMotionListener`	`mouseDragged()`

		`mouseMoved()`
`TextEvent`	`TextListener`	`textValueChanged()`
`WindowEvent`	`WindowListener`	`windowActivated()`
		`windowClosed()`
		`windowClosing()`
		`windowDeactivated()`
		`windowDeiconified()`
		`windowIconified()`
		`windowOpened()`

For each of the event listener interfaces that contains more than one method, `java.awt.event` defines a simple "adapter" class that provides an empty body for each of the methods in the interface. When you are only interested in one or two of the defined methods, it is sometimes easier to subclass the adapter class than it is to implement the interface. If you subclass the adapter, you only have to override the methods of interest, but if you implement the interface directly you have to define all of the methods, which means you must provide empty bodies for all the methods that are not of interest. These pre-defined no-op adapter classes bear the same name as the interfaces they implement, with "Listener" changed to "Adapter": `MouseAdapter`, `WindowAdapter`, etc.

Once you have implemented a listener interface, or subclassed a adapter class, you must instantiate your new class to define an individual event listener object. You then register that listener with the appropriate event source. In AWT programs, an event source is always some sort of AWT component. Event listener registration methods follow a standard naming convention: if an event source generates events of type `X`, it has a method named `addXListener()` to add an event listener, and a method `removeXListener()` to remove a listener. One of the nice features of the 1.1 event model is that it is easy to determine the types of events a component can generate--just look for the event listener registration methods. For example, by inspecting the API of the `Button` object, you can determine that it generates `ActionEvent` events. Table 7.7 lists AWT components and the events they generate.

Table 7.7: AWT Components and the Java 1.1 Events They Generate

Component	**Events Generated**	**Meaning**
Button	ActionEvent	User clicked on the button
Checkbox	ItemEvent	User selected or deselected an item
CheckboxMenuItem	ItemEvent	User selected or deselected an item
Choice	ItemEvent	User selected or deselected an item
Component	ComponentEvent	Component moved, resized, hidden, or shown

	FocusEvent	Component gained or lost focus
	KeyEvent	User pressed or released a key
	MouseEvent	User pressed or released mouse button, mouse entered or exited component, or user moved or dragged mouse. Note: `MouseEvent` has two corresponding listeners, `MouseListener` and `MouseMotionListener`.
Container	ContainerEvent	Component added to or removed from container
List	ActionEvent	User double-clicked on list item
	ItemEvent	User selected or deselected an item
MenuItem	ActionEvent	User selected a menu item
Scrollbar	AdjustmentEvent	User moved the scrollbar
TextComponent	TextEvent	User changed text
TextField	ActionEvent	User finished editing text
Window	WindowEvent	Window opened, closed, iconified, deiconified, or close requested

PREVIOUS | HOME | NEXT
Scribbling in Java 1.0 | BOOK INDEX | Scribbling in Java 1.1

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

7.4 Scribbling in Java 1.1

The Java 1.1 event model is quite flexible, and, as we'll see, there are several different ways you can use it to structure your event-handling code. Example 7.2 shows the first technique. Once again, this is our basic `Scribble` applet, this time using the Java 1.1 event model. This version of the applet implements the `MouseListener` and `MouseMotionListener` interfaces itself, and registers itself with its own `addMouseListener()` and `addMouseMotionListener()` methods.

Example 7.2: Scribble: Implementing the Listener Interfaces Directly

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class Scribble2 extends Applet
                      implements MouseListener,  MouseMotionListener {
  private int last_x, last_y;
  public void init() {
    // Tell this applet what MouseListener and MouseMotionListener
    // objects to notify when mouse and mouse motion events occur.
    // Since we implement the interfaces ourself, our own methods are called.
    this.addMouseListener(this);
    this.addMouseMotionListener(this);
  }
  // A method from the MouseListener interface.  Invoked when the
  // user presses a mouse button.
  public void mousePressed(MouseEvent e) {
    last_x = e.getX();
    last_y = e.getY();
  }
  // A method from the MouseMotionListener interface.  Invoked when the
  // user drags the mouse with a button pressed.
  public void mouseDragged(MouseEvent e) {
    Graphics g = this.getGraphics();
    int x = e.getX(), y = e.getY();
    g.drawLine(last_x, last_y, x, y);
    last_x = x; last_y = y;
  }
  // The other, unused methods of the MouseListener interface.
  public void mouseReleased(MouseEvent e) {;}
  public void mouseClicked(MouseEvent e) {;}
  public void mouseEntered(MouseEvent e) {;}
  public void mouseExited(MouseEvent e) {;}
  // The other method of the MouseMotionListener interface.
  public void mouseMoved(MouseEvent e) {;}
}
```

PREVIOUS | HOME | NEXT
The Java 1.1 Event Model | BOOK INDEX | Scribbling with Inner Classes

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

7.5 Scribbling with Inner Classes

The Java 1.1 event model was designed to work well with another new Java 1.1 feature: inner classes. Example 7.3 shows what the applet looks like when the event listeners are implemented as anonymous inner classes. Note how succinct this representation is. This is perhaps the most common way to use the Java 1.1 event model, so you'll probably see a lot of code that looks like this. In this case, our simple applet is nothing but event-handling code, so this version of it consists almost entirely of anonymous class definitions.

Note that we've added a feature to the applet. It now includes a **Clear** button. An `ActionListener` object is registered with the button; it clears the scribble when the appropriate event occurs.

Example 7.3: Scribble: Using Inner Classes

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class Scribble3 extends Applet {
  int last_x, last_y;
  public void init() {
    // Define, instantiate, and register a MouseListener object.
    this.addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent e) {
        last_x = e.getX();
        last_y = e.getY();
      }
    });
    // Define, instantiate, and register a MouseMotionListener object.
    this.addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent e) {
        Graphics g = getGraphics();
        int x = e.getX(), y = e.getY();
        g.setColor(Color.black);
        g.drawLine(last_x, last_y, x, y);
        last_x = x; last_y = y;
      }
    });
    // Create a clear button.
    Button b = new Button("Clear");
    // Define, instantiate, and register a listener to handle button presses.
    b.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {  // clear the scribble
        Graphics g = getGraphics();
        g.setColor(getBackground());
        g.fillRect(0, 0, getSize().width, getSize().height);
      }
    });
    // And add the button to the applet.
    this.add(b);
  }
}
```

PREVIOUS	HOME	NEXT
Scribbling in Java 1.1	BOOK INDEX	Inside the Java 1.1 Event Model

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

7.6 Inside the Java 1.1 Event Model

The listener-based event model we've seen in the sections above is ideal for creating a GUI out of pre-defined AWT components or out of Java Beans. It becomes a little cumbersome, however, when developing custom AWT components. AWT components (but not beans) provide a lower-level interface to this event model that is sometimes more convenient to use.

When an AWTEvent is delivered to a component, there is some default processing that goes on before the event is dispatched to the appropriate event listeners. When you define a custom component (by subclassing), you have the opportunity to override methods and intercept the event before it is sent to listener objects. When an AWTEvent is delivered to a component, it is passed to the processEvent() method.

By default, processEvent() simply checks the class of the event object and dispatches the event to a class-specific method. For example, if the event object is an instance of FocusEvent, it dispatches it to a method named processFocusEvent(). Or, if the event is of type ActionEvent, it is dispatched to processActionEvent(). In other words, any event type *X*Event is dispatched to a corresponding process*X*Event() method. The exception is for MouseEvent events, which are dispatched either to processMouseEvent() or processMouseMotionEvent(), depending on the type of the mouse event that occurred. For any given component, it is the individual process*X*Event() methods that are responsible for invoking the appropriate methods of all registered event listener objects. The processMouseEvent() method, for example, invokes the appropriate method for each registered MouseListener object. There is a one-to-one mapping between these methods and the event listener interfaces defined in java.awt.event. Each process*X*Event() method corresponds to an *X*Listener interface.

As you can see, there is a clear analogy between the Java 1.0 event model and this Java 1.1 low-level event model. processEvent() is analogous to the Java 1.0 handleEvent() method, and methods like processKeyEvent() are analogous to the Java 1.0 keyDown() and keyUp() methods. As with the Java 1.0 model, there are two levels at which you can intercept events: you can override processEvent() itself or you can rely on the default version of processEvent() to dispatch the events based on their class and instead override the individual event methods, such as processFocusEvent() and processActionEvent().

There is one additional requirement to make this low-level Java 1.1 event model work. In order to receive events of a particular type for a component, you must tell the component that you are interested in receiving that type of event. If you do not do this, for efficiency, the component does not bother to deliver that type of event. When using event listeners, the act of registering a listener is enough to notify the component that you are interested in receiving events of that type. But when you use the low-level model, you must register your interest explicitly. You do this by calling the enableEvents() method of the component and passing a bit mask that specifies each of the event types you are interested in. The bit mask is formed by ORing together various EVENT_MASK constants defined by the AWTEvent class.

Scribbling with Low-Level Event Handling

Example 7.4 is another variation on the Scribble applet. This one uses the Java 1.1 low-level event-handling model. It overrides the event-specific methods processMouseEvent(), processMouseMotionEvent(), and processKeyEvent(). Note how it calls enableEvents() in its init() method to register interest in events of that type. Furthermore, it calls requestFocus() to ask that it be given the keyboard focus, so that it can receive key events. Notice also that it passes events it is not interested in to the superclass event-processing method. In this case, the superclass is not going to use those events, but this is still a good practice.

Example 7.4: Scribble: Using the Low-Level Event Model

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class Scribble4 extends Applet {
  private int lastx, lasty;
  /** Tell the system we're interested in mouse events, mouse motion events,
   *  and keyboard events.  This is required or events won't be sent.
   */
  public void init() {
    this.enableEvents(AWTEvent.MOUSE_EVENT_MASK |
                      AWTEvent.MOUSE_MOTION_EVENT_MASK |
                      AWTEvent.KEY_EVENT_MASK);
    this.requestFocus();  // Ask for keyboard focus so we get key events.
  }
  /** Invoked when a mouse event of some type occurs */
  public void processMouseEvent(MouseEvent e) {
    if (e.getID() == MouseEvent.MOUSE_PRESSED) {  // Check the event type.
      lastx = e.getX(); lasty = e.getY();
    }
    else super.processMouseEvent(e); // Pass unhandled events to superclass.
  }
  /** Invoked when a mouse motion event occurs */
  public void processMouseMotionEvent(MouseEvent e) {
    if (e.getID() == MouseEvent.MOUSE_DRAGGED) {  // check type
      int x = e.getX(), y = e.getY();
      Graphics g = this.getGraphics();
      g.drawLine(lastx, lasty, x, y);
      lastx = x; lasty = y;
    }
    else super.processMouseMotionEvent(e);
  }
  /** Called on key events:  clear the screen when 'c' is typed. */
  public void processKeyEvent(KeyEvent e) {
    if ((e.getID() == KeyEvent.KEY_TYPED) && (e.getKeyChar() == 'c')) {
      Graphics g = this.getGraphics();
      g.setColor(this.getBackground());
      g.fillRect(0, 0, this.getSize().width, this.getSize().height);
    }
    else super.processKeyEvent(e);  // Pass unhandled events to our superclass.
  }
}
```

Another way to implement this example would be to override `processEvent()` directly instead of overriding the various methods that it invokes. If we did this, we'd end up with a large `switch` statement that separated events by type. When overriding `processEvent()`, it is particularly important to remember to pass unhandled events to `super.processEvent()` so that they can be dispatched correctly.

PREVIOUS | HOME | NEXT
Scribbling with Inner Classes | BOOK INDEX | New AWT Features

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

jar

Name

jar---Java Archive Tool

Availability

JDK 1.1 and later.

Synopsis

```
jar c|t|x[f][m][v] [jar-file] [manifest-file] [files]
```

Description

jar is a tool that can be used to create and manipulate Java Archive (JAR) files. A JAR file is a compressed ZIP file with an additional "manifest" file. The syntax of the *jar* command is reminiscent of the Unix *tar* (tape archive) command.

Options to *jar* are specified as a block of concatenated letters passed as a single argument, rather than as individual command-line arguments. The first letter of this option argument specifies what action *jar* is to perform and is required. Other option letters are optional. The various file arguments depend on which option letters are specified.

Options

The first argument to *jar* is a set of letters that specifies the operation it is to perform. The first letter specifies the basic action and is required. The three possible values are the following:

`c`

Create a new JAR archive. A list of input files and/or directories must be specified as the final arguments to *jar*.

`t`

List the contents of a JAR archive. If a JAR file is specified with the `f` option, its contents are listed. Otherwise, the JAR file to be listed is read from standard input.

`x`

Extract the contents of a JAR archive. If a JAR file is specified with the `f` option, its contents are extracted. Otherwise, the JAR file to be extracted is read from standard input. If the command is followed by a list of files and/or directories, only those files or directories are extracted from the JAR archive. Otherwise, all files in the archive are extracted.

This action specifier can be followed by optional command letters:

`f`

This option indicates that the JAR file to create, list, or extract from is specified on the command line. When `f` is used with `c`, `t`, or `x`, the JAR filename must be the second command-line argument to *jar* (i.e., it must follow the block of option letters). If this option is not specified, *jar* writes the JAR file it creates to standard output, or reads a JAR file from standard input.

`m`

This option is only used with the `c` action. It indicates that *jar* should read the manifest file (partial or complete) specified on the command line and use that file as the basis for the manifest it includes in the JAR file. When this argument is specified after the `f` option, the manifest filename should follow the destination filename. If `m` precedes the `f` option, the manifest filename should precede the destination filename.

`v`

Verbose. If this letter is specified with a `c` action, *jar* lists each file it adds to the archive with compression statistics. If it is used with a `t` action, *jar* lists the size and modification date for each file in the archive, instead of simply listing the filename. If `v` is used with `x`, *jar* displays the name of each file it extracts from the archive.

Examples

To create a simple JAR file:

```
% jar cvf my.jar *.java images
```

To list the contents of a file:

```
% jar tvf your.jar
```

To extract the manifest file from a JAR file:

```
% jar xf the.jar META-INF/MANIFEST.MF
```

To create a JAR file with a partial manifest specified:

```
% jar cfmv YesNoDialog.jar manifest.stub oreilly/beans/yesno
```

See Also

javakey

PREVIOUS | HOME | NEXT
appletviewer | BOOK INDEX | java

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

java

Name

java---The Java Interpreter

Availability

JDK 1.0 and later.

Synopsis

```
java [ interpreter options ] classname [ program arguments ]
java_g [ interpreter options ] classname [ program arguments ]
```

Description

java is the Java byte-code interpreter--it runs Java programs. *java_g* is a debugging version of the interpreter. It is unoptimized, and has some additional options for tracing the execution of a program.

The program to be run is the class specified by `classname`. This must be a fully qualified name, it must include the package name of the class, but not the *.class* file extension. Note that you specify the package and class name, with components separated by '.', not the directory and filename of the class, which has its components separated by '/' or '/'. If a Java class has no `package` statement, then it is not in any package, and the class name is specified alone. Examples:

```
% java david.games.Checkers
% java test
```

See the description of the `-classpath` option and the `CLASSPATH` environment variable below for information on specifying where *java* should look for classes.

The class specified by `classname` must contain a method `main()` with exactly the following signature:

```
public static void main(String argv[])
```

Any arguments following the `classname` on the *java* command line are placed into an array and passed to the `main()` method when *java* starts up.

If `main()` creates any threads, *java* runs until the last thread exits. Otherwise, the interpreter executes the body of `main()` and exits.

Although only a single class name is specified when invoking *java*, the interpreter automatically loads any additional classes required by the program. These classes are located relative to the Java class path, described under the `-classpath` option below.

By default, *java* runs a byte-code verifier on all classes loaded over the network. This verifier performs a number of tests on the byte-code of the loaded class to ensure, for example, that it does not corrupt the internal operand stack and that it performs appropriate run-time checks on such things as array references. The `-verify`, `-noverify`, and `-verifyremote` options control the byte-code verification process.

Options

`-classpath` *path*

> The path that *java* uses to look up the specified `classname` and all other classes that it loads. Specifying this option overrides the default path and the `CLASSPATH` environment variable. The class path is an ordered list of directories and ZIP files within and below which *java* searches for named classes. On UNIX systems, a path is specified as a colon-separated list of directories and ZIP files. On Windows systems, directories and ZIP files (which may have drive specifiers that use colons) are separated from each other with semicolons. For example, a UNIX `-classpath` specification might look like this:
>
> ```
> -classpath /usr/lib/java/classes:.:~/java/classes
> ```
>
> On a Windows system, the specification might be:

```
-classpath C:\tools\java\classes.zip;.;D:\users\david\classes
```

A period by itself in the path indicates that the current working directory is searched. Directories and ZIP files are searched in the order they appear. Place the standard Java classes first in the path if you do not want them to be accidentally or maliciously overridden by classes with the same name in other directories.

java expects to find class files in a directory hierarchy (or with a directory name within a ZIP file) that maps to the fully qualified name of the class. Thus, on a UNIX system, Java would load the class `java.lang.String` by looking for the file *java/lang/String.class* beneath one of the directories specified in the class path. Similarly, on a Windows 95 or Windows NT system (which support long filenames), *java* would look for the file *java\lang\String.class* beneath a specified directory or within a specified ZIP file.

If you do not specify `-classpath` or the `CLASSPATH` environment variable, the default class path is:

```
.:$JAVA/classes:$JAVA/lib/classes.zip       UNIX systems
.;$JAVA\classes;$JAVA\lib\classes.zip       Windows systems
```

Where `$JAVA` is JDK installation directory.

`-cs`, `-checksource`

Both of these options tell *java* to check the modification times on the specified class file and its corresponding source file. If the class file cannot be found or if it is out of date, it is automatically recompiled from the source.

`-D`*propertyname=value*

Defines `propertyname` to equal `value` in the system properties list. Your Java program can then look up the specified value by its property name. You may specify any number of `-D` options. For example:

```
% java -Dawt.button.color=gray -Dmy.class.pointsize=14 my.class
```

`-debug`

Causes *java* to display a password as it starts up. This password can be used to allow the *jdb* debugger to attach itself to this interpreter session. Note that this password should not be considered cryptographically secure.

`-help`

Print a usage message and exit.

`-l`*digit*

Sets the logging level for trace output. *java_g* only.

`-ms` *initmem*`[k|m]`

Specifies how much memory is allocated for the heap when the interpreter starts up. By default, `initmem` is specified in bytes. You can specify it in kilobytes by appending the letter `k` or in megabytes by appending the letter `m`. The default is 1 MB. For large or memory intensive applications (such as the Java compiler), you can improve run-time performance by starting the interpreter with a larger amount of memory. You must specify an initial heap size of at least 1000 bytes.

`-mx` *maxmem*`[k|m]`

Specifies the maximum heap size the interpreter will use for dynamically allocated objects and arrays. `maxmem` is specified in bytes by default. You can specify `maxmem` in kilobytes by appending the letter `k` and in megabytes by appending the letter `m`. The default is 16 MB. You must not specify a heap size less than 1000 bytes.

`-noasyncgc`

Do not do garbage collection asynchronously. With this option specified, *java* only performs garbage collection when it runs out of memory or when the garbage collector is explicitly invoked. Without this option, *java* runs the garbage collector as a separate, low-priority thread.

`-noclassgc`

Do not garbage collect loaded classes that are no longer in use. This option is only available in JDK 1.1 and later.

`-noverify`

Never run the byte-code verifier.

`-oss` *stacksize*`[k|m]`

Sets the size of each thread's Java code stack. By default, `stacksize` is specified in bytes. You can specify it in kilobytes by appending the letter `k` or in megabytes by appending the letter `m`. The default value is 400 KB. You must specify at least 1000 bytes.

`-prof[:`*file*`]`

Output profiling information to the specified `file` or to the file *java.prof* in the current directory. The format of this profiling information is not well documented. Prior to JDK 1.1, no `file` can be specified; profiling information is always output to *./java.prof*.

`-ss` *stacksize*`[k|m]`

Sets the size of each thread's native code stack. By default, `stacksize` is specified in bytes. You can specify it in kilobytes by appending the letter `k` or in megabytes by appending the letter `m`. The default value is 128 KB. You must specify at least 1000 bytes.

`-t`

Output a trace of all bytecodes executed. *java_g* only.

`-tm`

Output a trace of all methods executed. *java_g* only.

`-v`, `-verbose`

Print a terminal message each time *java* loads a class.

`-verbosegc`

Print a message whenever the garbage collector frees memory.

`-verify`

Run the byte-code verifier on all classes that are loaded.

`-verifyremote`

Run the byte-code verifier on all classes that are loaded through a class loader. (This generally means classes that are dynamically loaded from an untrusted location.) This is the default behavior for *java*.

`-version`

Print the version of the Java interpreter and exit.

Environment

`CLASSPATH`

Specifies an ordered list (colon-separated on UNIX, semicolon-separated on Windows systems) of directories and ZIP files in which *java* should look for class definitions. When a path is specified with this environment variable, *java* always implicitly appends the location of the system classes to the end of the path. If this environment variable is not specified, the default path is the current directory and the system classes. This variable is overridden by the `-classpath` option. See `-classpath` above for more information on specifying paths.

See Also

javac, *jdb*

javac

Name

javac---The Java Compiler

Availability

JDK 1.0 and later.

Synopsis

```
javac [ options ] files
```

Description

javac is the Java compiler--it compiles Java source code (in *.java* files) into Java byte-codes (in *.class* files). The Java compiler is itself written in Java.

javac may be passed any number of Java source files, whose names must all end with the *.java* extension. *javac* produces a separate *.class* class file for each class defined in the source files, regardless of how many source files there are. In other words, there need not be a one-to-one mapping between Java source files and Java class files. Note also that the compiler requires that there be only a single public class defined in any one source file, and that the name of the file (minus the *.java* extension) be the same as the name of the class (minus its package name, of course).

By default, *javac* places the class files it generates in the same directory as the corresponding source file. You can override this behavior with the `-d` option.

When a source file references a class that is not defined in another source file on the command line, *javac* searches for the definition of that class using the class path. The default class path contains only the current directory and the system classes. You may specify additional classes and packages to be searched with the `-classpath` option or the `CLASSPATH` environment variable.

If *javac* compiles a source file that relies on a class that is out of date (i.e., if the source file for that class is newer than the class file), it automatically recompiles that file.

Options

`-classpath` *path*

> The path that *javac* uses to look up classes referenced in the specified source code. This option overrides the default path and any path specified by the `CLASSPATH` environment variable. The `path` specified is an ordered list of directories and ZIP files, separated by colons on UNIX systems or semicolons on Windows systems.
>
> To specify additional directories or ZIP files to be searched, without overriding the default system class path, use the `CLASSPATH` environment variable. See the *java* reference page for more information on specifying paths.

`-d` *directory*

> The directory in which (or beneath which) class files should be stored. By default, *javac* stores the *.class* files it generates in the same directory as the *.java* file that those classes were defined in. If the `-d` flag is specified, however, the specified `directory` is treated as the root of the class hierarchy and *.class* files are placed in this directory, or in the appropriate subdirectory below it, depending on the package name of the class. Thus, the following command:

```
% javac -d java/classes java/src/Checkers.java
```

> places the file *Checkers.class* in the directory *java/classes* if the *Checkers.java* file has no `package` statement. On the other hand, if the source file specifies that it is in a package:

```
package david.games;
```

> then the *.class* file is stored in *java/classes/david/games*. When the `-d` option is specified, *javac* automatically creates any directories it needs to store its class files in the appropriate place.

`-depend`

Tells *javac* to recompile any out-of-date class files it encounters, not just those that are referenced from one of the specified source files.

`-deprecation`

Tells *javac* to issue a warning for every use of a deprecated API. By default, *javac* issues only a single warning if a program uses deprecated APIs. Available in JDK 1.1 and later.

`-g`

This option tells *javac* to add line numbers and local variable information to the output class files, for use by debuggers. By default, *javac* only generates the line numbers. With the `-O` option, *javac* does not generate even that information.

`-J`*javaoption*

Pass the argument `javaoption` directly through to the Java interpreter. `javaoption` should not contain spaces; if multiple arguments must be passed to the interpreter, use multiple `-J` options. Available in JDK 1.1 and later.

`-nowarn`

Tells *javac* not to print warning messages. Errors are still reported as usual.

`-nowrite`

Tells *javac* not to create any class files. Source files are parsed as usual, but no output is written. This option is useful when you want to check that a file will compile without actually compiling it.

`-O`

Enable optimization of class files. This option may cause *javac* to compile `static`, `final`, and `private` methods inline, so that they execute faster. The trade-off is that the class files will be larger. This option also prevents *javac* from adding line number debugging information to the class files.

`-verbose`

Tells the compiler to display messages about what it is doing.

Environment

`CLASSPATH`

Specifies an ordered list (colon-separated on UNIX, semicolon-separated on Windows systems) of directories and ZIP files in which *javac* should look for class definitions. When a path is specified with this environment variable, *javac* always implicitly appends the location of the system classes to the end of the path. If this environment variable is not specified, the default path is the current directory and the system classes. This variable is overridden by the `-classpath` option.

See Also

java, *jdb*

PREVIOUS

NEXT

javadoc

Name

javadoc---The Java Documentation Generator

Availability

JDK 1.0 and later.

Synopsis

```
javadoc [ options ] packagename
javadoc [ options ] filenames
```

Description

javadoc generates API documentation, in HTML format, for the specified package, or for the individual Java source files specified on the command line.

When a package name is specified on the command line, *javadoc* looks for a corresponding package directory relative to the class path. It then parses all of the *.java* source files in that directory and generates an HTML documentation file for each class and an HTML index of the classes in the package. By default, the HTML files are placed in the current directory. The `-d` option allows you to override this default.

Note that the `packagename` argument to *javadoc* is the name of the package (components separated by periods) and not the name of the package directory. You may need to specify the `-sourcepath` option so that *javadoc* can find your package source code correctly, if it is not stored in the same location as the package class files.

javadoc may also be invoked with any number of Java source files specified on the command line. Note that these are filenames, not class names, and are specified with any necessary directory components, and with the *.java* extension. When *javadoc* is invoked in this way, it reads the specified source files and generates HTML files (in the current directory, by default) that describe each public class defined in the specified source files.

The class documentation files that *javadoc* generates describe the class (or interface) and its inheritance hierarchy, and index and describe each of the `public` and `protected` members of the class. The generated file also contains any "doc comments" that are associated with the class and with its methods, constructors, and variables. A "doc comment," or documentation comment, is a Java comment that begins with `/**` and ends with `*/`. A doc comment may include any HTML markup tags (although it should not include structuring tags like <H1> or <HR>), and may also include tag values that are treated specially by *javadoc*. These special tags and their syntax are documented fully in Chapter 13, *Java Syntax*.

Options

`-author` *path*

> Specifies that author information specified with the `@author` tag should be output. This information is not output by default.

`-classpath` *path*

> This option specifies a path that *javadoc* uses to look up both class files and source files for the specified package. If you specify this option to tell *javadoc* to look for your source files, you must also be sure to include the standard system classpath as well, or *javadoc* will not be able to find the classes it needs. This option overrides the default path and any path specified by the `CLASSPATH` environment variable. The `path` specified is an ordered list of directories and ZIP files, separated by colons on UNIX systems or semicolons on Windows systems. To specify additional directories or ZIP files to search without overriding the default system class path, use the `CLASSPATH` environment variable. See the *java* reference page for more information on specifying paths.

`-d` *directory*

> The directory in which *javadoc* should store the HTML files it generates. The default is the current directory.

`-docencoding` *encoding-name*

Specifies the character encoding to be used for the output documents generated by *javadoc*. Available in JDK 1.1 and later.

`-encoding` *encoding-name*

Specifies the character encoding to be used to read the input source files and the documentation comments they contain. Available in JDK 1.1 and later.

`-J`*javaoption*

Pass the argument `javaoption` directly through to the Java interpreter. `javaoption` should not contain spaces; if multiple arguments must be passed to the interpreter, use multiple `-J` options. Available in JDK 1.1 and later.

`-nodeprecated`

Specifies that *javadoc* should not include `@deprecated` tags in its output, as it does by default. Available in JDK 1.1 and later.

`-noindex`

Specifies that *javadoc* should not generate the *AllNames.html* index file that it creates by default.

`-notree`

Specifies that `javadoc` should not generate the *tree.html* class hierarchy file that it creates by default.

`-sourcepath` *path*

A synonym for `-classpath`. Note that any specified "sourcepath" must include the system classpath.

`-verbose`

Tells *javadoc* to print additional messages about what it is doing.

`-version` *path*

Specifies that version information specified with the `@version` tag should be output. This information is not output by default. Note that this option does *not* tell *javadoc* to print its own version number.

Environment

`CLASSPATH`

Specifies an ordered list (colon-separated on UNIX, semicolon-separated on Windows systems) of directories and ZIP files in which *javadoc* should look for class definitions. When a path is specified with this environment variable, *javadoc* always implicitly appends the location of the system classes to the end of the path. If this environment variable is not specified, then the default path is the current directory and the system classes. This variable is overridden by the `-classpath` option.

Bugs

When *javadoc* cannot find a specified package, it produces a stub HTML file and does not warn you that the package was not found.

See Also

java, *javac*

PREVIOUS HOME NEXT
javac BOOK INDEX javah

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

javah

Name

javah---Native Method C File Generator

Availability

JDK 1.0 and later.

Synopsis

```
javah [ options ] classnames
```

Description

javah generates C header and source files (*.h* and *.c* files) that describe the specified classes. Note that classes are specified by classname, not filename. These generated C files provide the information necessary for implementing native methods for the specified classes in C. By default, *javah* produces output files suitable for the native interface used in JDK 1.0. If the `-jni` option is specified, it generates output files for use with the Java 1.1 Java Native Interface (JNI).

By default, *javah* generates a header file for the specified class or classes. This header file declares a C `struct` that contains fields that correspond to the instance fields of the Java class. The header also declares a procedure that you must implement for each of the native methods that the Java class contains. (A full description of how to implement Java native methods in C is beyond the scope of this reference page.)

If *javah* is run with the `-stubs` option, it generates a *.c* file that contains additional stub procedures necessary for linking the native method into the Java environment. Note that you should not put your native method implementation in this generated stub file.

With the `-jni` option specified, *javah* generates C header files that declare function prototypes each of the native methods of the specified classes. No structure definitions are required using this new native interface. The JNI does not require stub files, either, so `-stubs` should not be specified with `-jni`.

By default, *javah* creates C files in the current directory and bases their name on the name of the class. If the name of the class includes a package name, then the C files include all the components of the fully qualified class name, with periods replaced by underscores. You can override this default behavior with the `-d` and `-o` options.

Options

`-classpath` *path*

> The path that *javah* uses to look up the classes named on the command line. This option overrides the default path, and any path specified by the `CLASSPATH` environment variable. The `path` specified is an ordered list of directories and ZIP files, separated by colons on UNIX systems or semicolons on Windows systems.
>
> To specify additional directories or ZIP files for *javah* to search without overriding the default system class path, use the `CLASSPATH` environment variable. See the *java* reference page for more information on specifying paths.

`-d` *directory*

> Specifies a directory where *javah* should store the files it generates. By default it stores them in the current directory. This option does not work with `-o`; you must specify any desired directory within the `-o` filename.

`-help`

> Causes *javah* to display a simple usage message and exit.

`-jni`

> Specifies that *javah* should output a header file for use with the new JNI (Java Native Interface), rather than using the old JDK 1.0 native interface. Available in JDK 1.1 and later.

`-o` *outputfile*

Combine all *.h* or *.c* file output into a single file, `outputfile`. This is a convenience when you want to implement native methods for a number of classes in a single package. It allows you to avoid having many short *.h* and *.c* files that must be manipulated separately.

`-stubs`

Generate *.c* stub files for the class or classes, and do not generate the *.h* header files. Without this option, *javah* generates header files.

`-td` *directory*

The directory where *javah* should store temporary files. The default is */tmp*.

`-trace`

Specifies that *javah* should include tracing output commands in the stub files it generates.

`-v`

Verbose. Causes *javah* to print messages about what it is doing.

`-version`

Causes *javah* to display its version number.

Environment

`CLASSPATH`

Specifies an ordered list (colon-separated on UNIX, semicolon-separated on Windows systems) of directories and ZIP files in which *javah* should look for class definitions. When a path is specified with this environment variable, *javah* always implicitly appends the location of the system classes to the end of the path. If this environment variable is not specified, the default path is the current directory and the system classes. This variable is overridden by the `-classpath` option.

See Also

java, *javac*

javakey

Name

javakey---Key Management and Digital Signatures

Availability

JDK 1.1 and later.

Synopsis

```
javakey options
```

Description

javakey provides a command-line interface to a number of complex key and certificate generation and management tasks, including the generation of digital signatures. There are quite a few options that perform a number of distinct operations. *javakey* manages a system database of entities. Each entity may have public and private keys and/or certificates associated with it, and in addition, each entity may be declared to be trusted or not. Any entity in the database may be an "identity" or a "signer." Identities have only a public key associated with them, while signers have both a public and private key, and thus may sign files.

The different *javakey* operations are specified with the various options described below.

Options

`-c` *identity-name* `[true|false]`

Create. Create and add a new identity to the database, using the specified name. If the identity name is followed by `true`, declare the identity to be trusted. Otherwise make it untrusted.

`-cs` *signer-name* `[true|false]`

Create signer. Create and add a new signer entity to the database, using the specified name. If the name is followed by `true`, declare the signer to be trusted. Otherwise make it untrusted.

`-t` *entity-name* `true|false`

Assign trust. Specify whether the named entity is trusted (`true`) or not (`false`).

`-l`

List. List the names of all entities in the security database.

`-ld`

List details. List the names and other details about all entities in the security database.

`-li` *entity-name*

List information. List detailed information about the named entity from the security database.

`-r` *entity-name*

Remove. Remove the named entity from the security database.

`-ik` *identity-name keyfile*

Import key. Read a public key from the specified file and associate it with the named identity. The key must be in X.509 format.

`-ikp` *signer-name pubkeyfile privkeyfile*

Import key pair. Read the specified public key and private key files and associate them with the named signer entity. The keys must be in X.509 format.

`-ic` *entity-name certificate-file*

Import certificate. Read a certificate from the named certificate file and associate it with the named entity. If the entity already has a public key, compare it to the key in the certificate and issue a warning if they do not match. If the entity has not had a public key assigned, use the public key from the certificate.

`-ii` *entity-name*

Import information. This command allows you to enter arbitrary textual information about an entity into the database.

`-gk` *signer algorithm size [pubfile [privfile]]*

Generate key. Generate a public and private key and associate them with the named signer. Use the specified algorithm. Currently, the only supported algorithm is "DSA." Generates keys of the specified number of bits, which must be between 512 and 1024. If `pubfile` is specified, write the public key to the specified file. If `privfile` is specified, write the private key to the specified file.

`-g` *signer algorithm size [pubfile [privfile]]*

A synonym for the `-gk` command.

`-gc` *directivefile*

Generate certificate. Generate a certificate according to the parameters specified in the directive file. The directive file is a `Properties` file that must provide values for the following named properties:

- `issuer.name`. The name of the entity issuing the certificate.
- `issuer.cert`. The issuer's certificate number to be used to sign the generated certificate (unless the certificate will be self-signed.)
- `subject.name`. The database name of the entity that the certificate is being issued to.
- `subject.real.name`. The real name of the entity that the certificate is being issued to.
- `subject.country`. The country that the subject entity is in.

- `subject.org`. The organization that the subject entity is affiliated with.
- `subject.org.unit`. A division within the subject's organization.
- `start.date`. The starting date (and time) of the certificate.
- `end.date`. The ending date (and time) of the certificate.
- `serial.number`. A serial number for the certificate. This number must be unique among all certificates generated by the issuer.
- `out.file`. An optional filename that specifies what file the certificate should be written to.

`-dc` *certfile*

Display certificate. Display the contents of the certificate stored in `certfile`.

`-ec` *entity certificate-number file*

Export certificate. Output the numbered certificate of the specified entity into the specified file. Use the `-li` command to inspect the certificate numbers for a given entity.

`-ek` *entity pubfile [privfile]*

Export key. Output the public key of the specified entity into the specified file. If the entity is a signer, and the `privfile` is specified, additionally export the private key of the entity to that file.

`-gs` *directivefile jarfile*

Generate signature. Apply a digital signature to the specified JAR file using the directives in the specified directive file. The directive file is a `Properties` file that must provide values for the following named properties:

- `signer`. The entity name of the signer.
- `cert`. The certificate number to use for the signature.
- `chain`. The length of a chain of certificates to include. This is not currently supported; specify 0.

    - `signature.file`. The basename of the signature file and signature block to be inserted into the JAR file. It must be 8 characters or less. This name should not conflict with any other digital signatures that may be inserted into the JAR file.
    - `out.file`. This optional property specifies the name that should be used for the signed JAR file that is generated.

See Also

jar

PREVIOUS javah | HOME BOOK INDEX | NEXT javap

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

javap

Name

javap---The Java Class Disassembler

Availability

JDK 1.0 and later.

Synopsis

```
javap [ options ] classnames
```

Description

javap disassembles the class files specified by the class names on the command line and prints a human-readable version of those classes.

By default, *javap* prints declarations of the non-`private` members of each of the classes specified on the command line. The `-l`, `-p`, and `-c` options specify additional information to be printed, including a complete disassembly of the byte-codes in each of the specified classes. *javap* can also be used to run the class verifier on Java classes.

Options

`-c`

Print the Java Virtual Machine instructions for each of the methods in each of the specified classes. This option disassembles all methods, including `private` methods.

`-classpath` *path*

The path that *javap* uses to look up the classes named on the command line. This option overrides the default path and any path specified by the `CLASSPATH` environment variable. The `path` specified is an ordered list of directories and ZIP files, separated by colons on UNIX systems or semicolons on Windows systems.

To specify additional directories or ZIP files for *javap* to search without overriding the default system class path, use the `CLASSPATH` environment variable. See the *java* reference page for more information on specifying paths.

`-h`

Outputs the class in a form suitable for inclusion in a C header file.

`-l`

Prints line numbers and local variable tables in addition to the public fields of the class. Note that line numbers and local variable information is included for use with debuggers. Local variable information is available only if a class was compiled with the `-g` option to *javac*; line number information is available only if a class was compiled *without* the `-O` option.

`-p`

Prints `private` methods and variables of the specified class in addition to the `public` ones. Note that some compilers (though not *javac*) may allow this `private` field information to be "obfuscated" in such a way that `private` fields and method arguments no longer have meaningful names. This makes Java classes harder to disassemble or reverse engineer.

`-s`

Outputs the class member declarations using the internal Virtual Machine format.

`-v`

Verbose. Outputs additional information (in the form of Java comments) about each member of each specified class.

`-verify`

Causes *javap* to run the class verifier on the specified classes and display the results of verification.

`-version`

Causes *javap* to display its version number.

Environment

`CLASSPATH`

Specifies an ordered list (colon-separated on UNIX, semicolon-separated on Windows systems) of directories and ZIP files in which *javap* should look for class definitions. When a path is specified with this environment variable, *javap* always implicitly appends the location of the system classes to the end of the path. If this environment variable is not specified, the default path is the current directory and the system classes. This variable is overridden by the `-classpath` option.

See Also

java, *javac*

PREVIOUS HOME NEXT
javakey BOOK INDEX jdb

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

jdb

Name

jdb---The Java Debugger

Availability

JDK 1.0 and later.

Synopsis

```
jdb [ java options ] class
jdb [ -host hostname ] -password password
```

Description

jdb is a debugger for Java classes. It is text-based, command-line oriented, and has a command syntax like that of the UNIX *dbx* or *gdb* debuggers.

When *jdb* is invoked with the name of a Java class, it starts another copy of the *java* interpreter, passing any specified *java* options to the interpreter. *jdb* is itself a Java program, running in its own copy of the interpreter. This new interpreter loads the specified class file and stops for debugging before executing the first Java byte-code.

jdb may also be started with the `-password` and optional `-host` arguments. Invoked in this way, *jdb* "attaches itself" to an already running copy of the interpreter. In order for this to work, the Java interpreter running the program to be debugged must have been started with the `-debug` option. When the interpreter is started with this option, it prints a password that must be used with the *jdb* `-password` option.

Once a debugging session is started, you may issue any of the commands described below.

Options

When invoking *jdb* with a specified class file, any of the *java* interpreter options may be specified. See the *java* reference page for an explanation of these options.

When attaching *jdb* to an already running Java interpreter, the following options are available:

`-host` *hostname*

> Specifies the name of the host upon which the desired interpreter session is running.

`-password` *password*

> This option is required to attach to a running interpreter. The interpreter must have been started with the `-debug` option, and this `-password` option specifies the password that the interpreter generated. Only a debugger that knows this password is allowed to attach to the interpreter. Note that the passwords generated by *java* should not be considered cryptologically secure.

Commands

jdb understands the following debugging commands:

`!!`

> This is a shorthand command that is replaced with the text of the last command entered. It may be followed with additional text that is appended to that previous command.

`catch` [*exception class*]

> Cause a breakpoint whenever the specified exception is thrown. If no exception is specified, the command lists the exceptions currently being caught. Use `ignore` to stop these breakpoints from occurring.

`classes`

List all classes that have been loaded.

`clear` [*class*:*line*]

Remove the breakpoint set at the specified line of the specified class. Typing `clear` or `stop` with no arguments displays a list of current breakpoints and the line numbers that they are set at.

`cont`

Resume execution. This command should be used when the current thread is stopped at a breakpoint.

`down` [*n*]

Move down `n` frames in the call stack of the current thread. If `n` is not specified, move down one frame.

`dump` *id(s)*

Print the value of all fields of the specified object or objects. If you specify the name of a class, `dump` displays all class (static) methods and variables of the class, and also displays the superclass and list of implemented interfaces. Objects and classes may be specified by name or by their eight-digit hexadecimal ID number. Threads may also be specified with the shorthand `t@`*thread-number*.

`exit (or quit)`

Quit *jdb*.

`gc`

Run the garbage collector to force unused objects to be reclaimed.

`help (or ?)`

Display a list of all *jdb* commands.

`ignore` *exception class*

Do not treat the specified exception as a breakpoint. This command turns off a `catch` command.

`list` [*line number*]

List the specified line of source code as well as several lines that appear before and after it. If no line number is specified, use the line number of the current stack frame of the current thread. The lines listed are from the source file of the current stack frame of the current thread. Use the `use` command to tell *jdb* where to find source files.

`load` *classname*

Load the specified class into *jdb*.

`locals`

Display a list of local variables for the current stack frame. Java code must be compiled with the `-g` option in order to contain local variable information.

`memory`

Display a summary of memory usage for the Java program being debugged.

`methods` *class*

List all methods of the specified class. Use `dump` to list the instance variables or an object or the class (static) variables of a class.

`print` *id(s)*

Print the value of the specified item or items. Each item may be a class, object, field, or local variable, and may be specified by name or by eight-digit hexadecimal ID number. You may also refer to threads with the special syntax `t@`*thread-number*. The `print` command displays an object's value by invoking its `toString()` method.

`resume` [*thread(s)*]

Resume execution of the specified thread or threads. If no threads are specified, all suspended threads are resumed. See also `suspend`.

`run` [*class*] [*args*]

Run the `main()` method of the specified class, passing the specified arguments to it. If no class or arguments are specified, use the class and arguments specified on the *jdb* command line.

`step`

Run the current line of the current thread and stop again.

`stop [ at` *class*:*line*]
`stop [ in` *class*.*method*]

Set a breakpoint at the specified line of the specified class or at the beginning of the specified method of the specified class. Program execution stops when it reaches this line or enters the method. If `stop` is executed with no arguments, then it lists the current breakpoints.

`suspend` [*thread(s)*]

Suspend the specified thread or threads. If no threads are specified, suspend all running threads. Use `resume` to restart them.

`thread` *thread*

Set the current thread to the specified thread. This thread is used implicitly by a number of other *jdb* commands. The thread may be specified by name or number.

`threadgroup` *name*

Set the current thread group to the named thread group.

`threadgroups`

List all thread groups running in the Java interpreter session being debugged.

`threads` [*threadgroups*]

List all threads in the named thread group. If no thread group is specified, list all threads in the current thread group (specified by `threadgroup`).

`up` [*n*]

Move up `n` frames in the call stack of the current thread. If `n` is not specified, move up one frame.

`use` [*source-file-path*]

Set the path used by *jdb* to look up source files for the classes being debugged. If no path is specified, display the current source path being used.

`where [` *thread* `] [ all ]`

Display a stack trace for the specified thread. If no thread is specified, display a stack trace for the current thread. If `all` is specified, display a stack trace for all threads.

Environment

`CLASSPATH`

Specifies an ordered list (colon-separated on UNIX, semicolon-separated on Windows systems) of directories and ZIP files in which *jdb* should look for class definitions. When a path is specified with this environment variable, *jdb* always implicitly appends the location of the system classes to the end of the path. If this environment variable is not specified, the default path is the current directory and the system classes. This variable is overridden by the `-classpath` option.

See Also

java

native2ascii

Name

native2ascii---Convert Java source code to ASCII

Availability

JDK 1.1 and later.

Synopsis

```
native2ascii [ options ] [ inputfile [ outputfile ]]
```

Description

javac can only process files encoded in ASCII, with any other characters encoded using the `\u`*xxxx* Unicode notation. *native2ascii* is a simple program that reads a Java source file encoded using a local encoding and converts it to the ASCII-plus-encoded-Unicode form required by *javac*.

The `inputfile` and `outputfile` are optional. If unspecified, standard input and standard output are used, making *native2ascii* suitable for use in pipes.

Options

`-encoding` *encoding-name*

Specifies the encoding used by source files. If this option is not specified, the encoding is taken from the `file.encoding` system property.

`-reverse`

Specifies that the conversion should be done in reverse--from encoded `\u`*xxxx* characters to characters in the native encoding.

See Also

`java.io.InputStreamReader`, `java.io.OutputStreamWriter`

PREVIOUS HOME NEXT
jdb BOOK INDEX serialver

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

serialver

Name

serialver---Class Version Number Generator

Availability

JDK 1.1 and later.

Synopsis

```
serialver [-show] classname...
```

Description

serialver displays the version number, or serialization-unique identifier, for a named class or classes. If the class declares a `long serialVersionUID` constant, the value of that field is displayed. Otherwise, a unique version number is computed by applying the Secure Hash Algorithm (SHA) to the API defined by the class. This program is primarily useful for computing an initial unique version number for a class, which is then declared as a constant in the class. The output of *serialver* is a line of legal Java code, suitable for pasting into a class definition.

Options

`-show`

When the `-show` option is specified, *serialver* displays a simple graphical interface that allows the user to type in a single classname at a time and obtain its serialization UID. When using `-show`, no class names may be specified on the command-line.

Environment

`CLASSPATH`

serialver is written in Java, and so it is sensitive to the `CLASSPATH` environment variable in the same way that the *java* interpreter is. The specified classes are looked up relative to this class path.

See Also

`java.io.ObjectStreamClass`

PREVIOUS | HOME | NEXT
native2ascii | BOOK INDEX | How To Use This Quick Reference

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

14. System Properties

Contents:

Java programs cannot read environment variables the way that native programs can. The reason is that environment variables are platform dependent. Similar mechanisms exist, however, that allow applications to read the value of a named resource. These resource values allow customization of an application's behavior based on site-specific parameters, such as the type of host, or based on user preferences.

These named resource values are specified for applications in the "system properties" list. Applications can read these "system properties" with the `System.getProperty()` method, or can read the entire list of properties with `System.getProperties()`. `System.getProperty()` returns the property value as a string. Applications can also read properties in parsed form using methods that are based on `System.getProperty()`, such as `Font.getFont()`, `Color.getColor()`, `Integer.getInteger()`, and `Boolean.getBoolean()`.

14.1 Standard System Properties

When the Java interpreter starts, it inserts a number of standard properties into the system properties list. These properties, and the meaning of their values, are listed in Table 14.1. The table also specifies whether untrusted applets are allowed (at least by default) to read the value of these properties. For reasons of security, untrusted code is only allowed to read the values of properties to which it has explicitly been granted access. (Untrusted applets are not allowed to set the value of system properties, nor are they allowed to call `System.getProperties()` to obtain the entire list of properties.)

Table 14.1: Standard System Properties

Name	Value	Applet Access

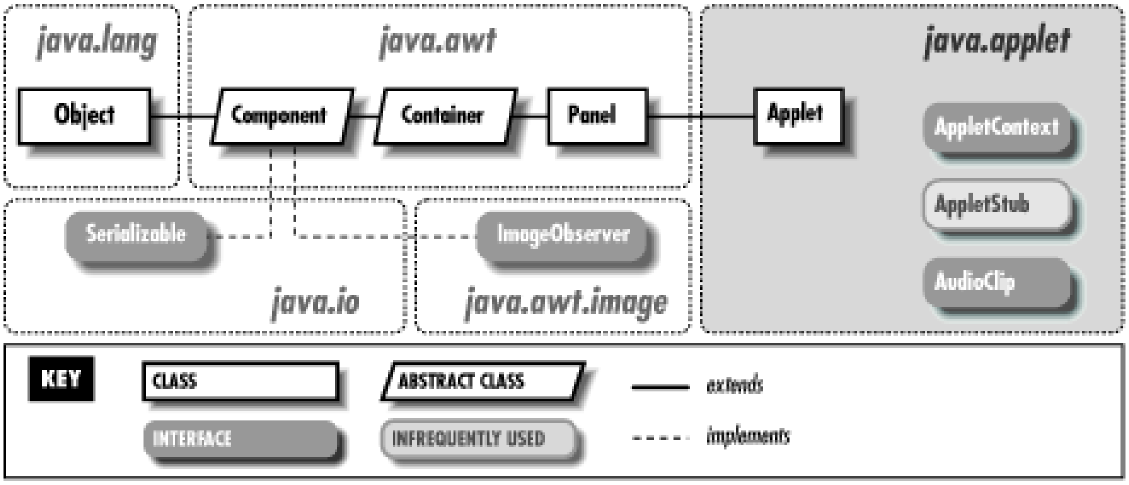
java.version	Version of the Java interpreter	yes
java.vendor	Vendor-specific identifier string	yes
java.vendor.url	Vendor's URL	yes
java.class.version	The version of the Java API	yes
java.class.path	The classpath value	no
java.home	The directory Java is installed in	no
java.compiler	The JIT compiler to use, if any (Java 1.1)	no
os.name	The name of the operating system	yes
os.arch	The host hardware architecture	yes
os.version	Version of the host operating system	yes
file.separator	Platform-dependent file separator (e.g., `/` or #)	yes
path.separator	Platform-dependent path separator (e.g., `:` or `;`)	yes
line.separator	Platform-dependent line separator (e.g., `#n` or `#r#n`)	yes
user.name	The username of the current user	no
user.home	The home directory of the current user	no
user.dir	The current working directory	no
user.language	The 2-letter language code of the default locale (Java 1.1)	no
user.region	The 2-letter country code of the default locale (Java 1.1)	no
user.timezone	The default time zone (Java 1.1)	no
file.encoding	The character encoding for the default locale (Java 1.1)	no
file.encoding.pkg	The package that contains converters between local encodings and Unicode (Java 1.1)	no

17. The java.applet Package

Contents:

An *applet* is a small, embeddable Java program. The `java.applet` package is a small one. It contains the `Applet` class, which is the superclass of all applets, and three related interfaces. Figure 17.1 shows the class hierarchy of this package. See Chapter 6, *Applets*, for more information about this package.

Figure 17.1: The java.applet package



17.1 java.applet.Applet (JDK 1.0)

This class implements an applet. To create your own applet, you should create a subclass of this class and override some or all of the following methods. Note that you never need to call these methods--they are called when appropriate by a Web browser or other applet viewer.

`init()` should perform any initialization for the applet; it is called when the applet first starts. `destroy()` should free up any resources that the applet is holding; it is called when the applet is about to be permanently stopped. `start()` is called to make the applet start doing whatever it is that it does. Often, it starts a thread to perform an animation or similar task. `stop()` should temporarily stop the applet from executing. It is called when the applet temporarily becomes hidden or non-visible.

`getAppletInfo()` should return text suitable for display in an **About** dialog posted by the Web browser or applet viewer. `getParameterInfo()` should return an arbitrary-length array of three-element arrays of strings where each element describes one of the parameters that this applet understands. The three elements of each parameter description are strings that specify, respectively, the parameter's name, type, and description.

In addition to these methods, an applet also typically overrides several of the methods of `java.awt.Component`, notably the `paint()` method to draw the applet on the screen. There are also several `Applet` methods that you do not override but may call from applet code: `showStatus()` displays text in the Web browser or applet viewer's status line. `getImage()` and `getAudioClip()` read image (GIF and JPEG formats) and audio files (AU format) over the network and return corresponding Java objects. `getParameter()` looks up the value of a parameter specified with a `<PARAM>` tag within an `<APPLET>...</APPLET>` pair. `getCodeBase()` returns the base URL from which the applet's code was loaded, and `getDocumentBase()` returns the base URL from which the HTML document containing the applet was loaded. `getAppletContext()` returns an `AppletContext` object, which also has useful methods.

```
public class Applet extends Panel {
    // Default Constructor: public Applet()
    // Public Instance Methods
            public void destroy();
            public AppletContext getAppletContext();
            public String getAppletInfo();
            public AudioClip getAudioClip(URL url);
            public AudioClip getAudioClip(URL url, String name);
            public URL getCodeBase();
            public URL getDocumentBase();
            public Image getImage(URL url);
            public Image getImage(URL url, String name);
            public Locale getLocale();  // Overrides Component
            public String getParameter(String name);
            public String[][] getParameterInfo();
            public void init();
            public boolean isActive();
            public void play(URL url);
            public void play(URL url, String name);
            public void resize(int width, int height);  // Overrides Component
            public void resize(Dimension d);  // Overrides Component
            public final void setStub(AppletStub stub);
            public void showStatus(String msg);
            public void start();
            public void stop();
}
```

Hierarchy:

```
Object->Component(ImageObserver, MenuContainer, Serializable)->Container->Panel->Applet
```

Returned By:

```
AppletContext.getApplet()
```

PREVIOUS | HOME | NEXT
Reading a Quick Reference Entry | BOOK INDEX | java.applet.AppletContext (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

5.2 Reading a Quick Reference Entry

Each class and interface has its own entry in this quick reference. These quick-reference entries document the class or interface as described below. Because the information in each entry is quite dense, the descriptions of it that follow are somewhat complicated. I recommend that you flip through the following chapters as you read to find examples of each of the features described.

Name and Availability

Each quick reference entry has a title that is the name of the class or interface it documents. To the right of that title, you'll find availability information that indicates when the class or interface was added to the Java API. The string "JDK 1.0" indicates that the class or interface has been around since the original release of Java. The string "JDK 1.1" indicates that it has been added in the Java 1.1 release, and is therefore not backwards compatible with Java 1.0 environments. If the availability string is followed by the word "Deprecated," it means that the class or interface has been deprecated and its use is discouraged. There are two such deprecated classes in Java 1.1.

Description

The class name is followed by a short description of the most important features of the class. This description may be anywhere from a couple of sentences to several paragraphs long.

Synopsis

The description is always followed by a synopsis of the class or interface. This is a listing that looks like a Java class definition, except that method bodies and field initializers are omitted. This synopsis contains the following information:

Class Modifiers

The synopsis begins with a list of class modifiers. All classes and interfaces in this quick reference are `public`; some are also declared `abstract` or `final`

Class or Interface

If the modifiers are followed by the `class` keyword, it is a class that is being documented. If they are followed by the `interface` keyword, it is an interface that is being documented.

Class Name

The name of the class or interface follows the `class` or `interface` keyword. It is highlighted in bold.

Superclass

The superclass of the class follows the `extends` keywords.

Interfaces

The list of interfaces that the class implements, if any, follows the `implements` keyword.

Members

The constructors, fields, and methods defined by the class or interface form the bulk of the synopsis. All `public` and `protected` members are listed. They are divided into the following categories, and listed alphabetically by name within each category. Each category begins with a comment to break the synopsis listing into logical sections. The categories, in the order listed, are:

1. Public constructors
2. Protected constructors
3. Constants
4. Class variables
5. Public instance variables
6. Protected instance variables

7. Class methods

8. Public instance methods

9. Protected instance methods

Availability

If a member synopsis begins with the string "1.1", it indicates that the constructor, field, or method has been added to the class or interface in Java 1.1. Note that this indication only appears in classes and interfaces that are not themselves new in Java 1.1. If a member synopsis begins with "#", it means that the constructor, field, or method has been deprecated in Java 1.1, and that its use is discouraged.

Member Modifiers

The modifiers for each member are listed. These provide important information about how the members are used. The modifiers you may find listed are: `public`, `protected`, `static`, `abstract`, `final`, `synchronized`, `native`, and `transient`.

Member Type

The listing for a member may include a type. The types of fields and constants are shown, as are the return types of methods. Constructors do not have return types in Java.

Member Name

The name of each class member is in bold, for easy scanning.

Parameters

The synopsis for a method or constructor includes the type and name of each parameter that it takes. The parameter names are shown in italic to indicate that they are not to be used literally.

Exceptions

The exceptions that may be thrown by a method or constructor follow the `throws` keyword in the synopsis.

Inheritance

The synopsis for a method may be followed by a comment that includes a class or interface name. If a method is followed by a `//Overrides` comment, the method overrides a method by the same name in the specified superclass. If a method synopsis is followed by a `//Defines` comment, the method provides the definition of an `abstract` method of the specified superclass. Finally, if a method synopsis is followed by a `//From` comment, the method implements a method from the named interface (which is implemented by the class or a superclass).

Cross References

The synopsis section is followed by a number of optional "cross reference" sections that indicate other, related classes that may be of interest. In the first edition of this book, this information was available in separate index chapters. We think it should be even more useful when associated directly with each class and interface entry. The cross reference sections are the following:

Hierarchy

This section lists all of the superclasses of the class, as well as any interfaces implemented by those superclasses. It may also list any interfaces extended by an interface. This section only appears when it provides information that is not available from the `extends` and `implements` clauses of the class synopsis. In the hierarchy listing, arrows indicate superclass to subclass relationships, while the interfaces implemented by a class follow the class name in parentheses. This information can be useful, for example, to determine whether a class implements `Serializable` or `Cloneable` somewhere up its superclass hierarchy.

Extended By

This section lists all direct subclasses of this class, or any interfaces that extend this interface, which tells you that there are more specific classes or interfaces to look at.

Implemented By

This section lists all of the classes that directly implement this interface, which is useful when you know that you want to use the interface but you don't know what implementations of it are available.

Passed To

This section lists all of the methods and constructors that are passed an object of this type as an argument, which is useful when you have an object of a given type and want to figure out what you can do with it.

Returned By

This section lists all of the methods (but not constructors) that return an object of this type, which is useful when you know that you want to work with an object of this type, but don't know how to obtain one.

Type Of

This section lists all of the fields and constants that are of this type, which can help you figure out how to obtain an object of this type.

Thrown By

For exception and error classes, this section lists all of the methods and constructors that throw exceptions of this type. This material helps you figure out when a given exception or error may be thrown. Note, however, that this section is based on the exception types listed in the `throws` clauses of methods and constructors. Subclasses of `RuntimeException` do not have to be listed in `throws` clauses, so it is not possible to generate a complete cross reference of methods that throw these types of "unchecked" exceptions.

PREVIOUS | HOME | NEXT
Finding a Quick Reference Entry | BOOK INDEX | The java.applet Package

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

5. How To Use This Quick Reference

Contents:

The quick-reference section that follows packs a lot of information into a small space. This introduction explains how to get the most out of that information. It explains how the quick reference is organized and how to read the individual quick-ref entries.

5.1 Finding a Quick Reference Entry

The following chapters each document one package of the Java API. The packages are listed alphabetically, beginning with `java.applet` and ending with `java.util.zip`. Each chapter begins with an overview of the package, including a hierarchy diagram for the classes and interfaces in the package. Within each chapter, the classes and interfaces of a package are themselves listed alphabetically.

If you know the name of a class, but not of the package that it is a part of, or if you know the name of a method or field, but do not know what class defines it, use the index in Chapter 32, *Class, Method, and Field Index* to find the information you need.

17.2 java.applet.AppletContext (JDK 1.0)

This interface defines the methods that allow an applet to interact with the context in which it runs (which is usually a Web browser or an applet viewer). The object that implements the `AppletContext` interface is returned by `Applet.getAppletContext()`. You can use it to take advantage of a Web browser's cache, or to display a message to the user in the Web browser's or applet viewer's message area.

The `getAudioClip()` and `getImage()` methods may make use of a Web browser's caching mechanism. `showDocument()` and `showStatus()` give an applet a small measure of control over the appearance of the browser or applet viewer. The `getApplet()` and `getApplets()` methods allow an applet to find out what other applets are running at the same time.

```
public abstract interface AppletContext {
    // Public Instance Methods
            public abstract Applet getApplet(String name);
            public abstract Enumeration getApplets();
            public abstract AudioClip getAudioClip(URL url);
            public abstract Image getImage(URL url);
            public abstract void showDocument(URL url);
            public abstract void showDocument(URL url, String target);
            public abstract void showStatus(String status);
}
```

Returned By:

`Applet.getAppletContext(), AppletStub.getAppletContext()`

17.3 java.applet.AppletStub (JDK 1.0)

This is an internal interface used when implementing an applet viewer.

```
public abstract interface AppletStub {
    // Public Instance Methods
            public abstract void appletResize(int width, int height);
            public abstract AppletContext getAppletContext();
            public abstract URL getCodeBase();
            public abstract URL getDocumentBase();
            public abstract String getParameter(String name);
            public abstract boolean isActive();
}
```

Passed To:

```
Applet.setStub()
```

17.4 java.applet.AudioClip (JDK 1.0)

This interface describes the essential methods that an audio clip must have. `AppletContext.getAudioClip()` and `Applet.getAudioClip()` both return an object that implements this interface. The current JDK implementations of this interface only work with sounds encoded in AU format. The `AudioClip` interface is in the `java.applet` package only because there is not a better place for it.

```
public abstract interface AudioClip {
    // Public Instance Methods
            public abstract void loop();
            public abstract void play();
            public abstract void stop();
}
```

Returned By:

`Applet.getAudioClip(), AppletContext.getAudioClip()`

18. The java.awt Package

Contents:

The `java.awt` package is the Abstract Windowing Toolkit. The classes of this package may be roughly divided into three categories (see Figure 18.1 and Figure 18.2).

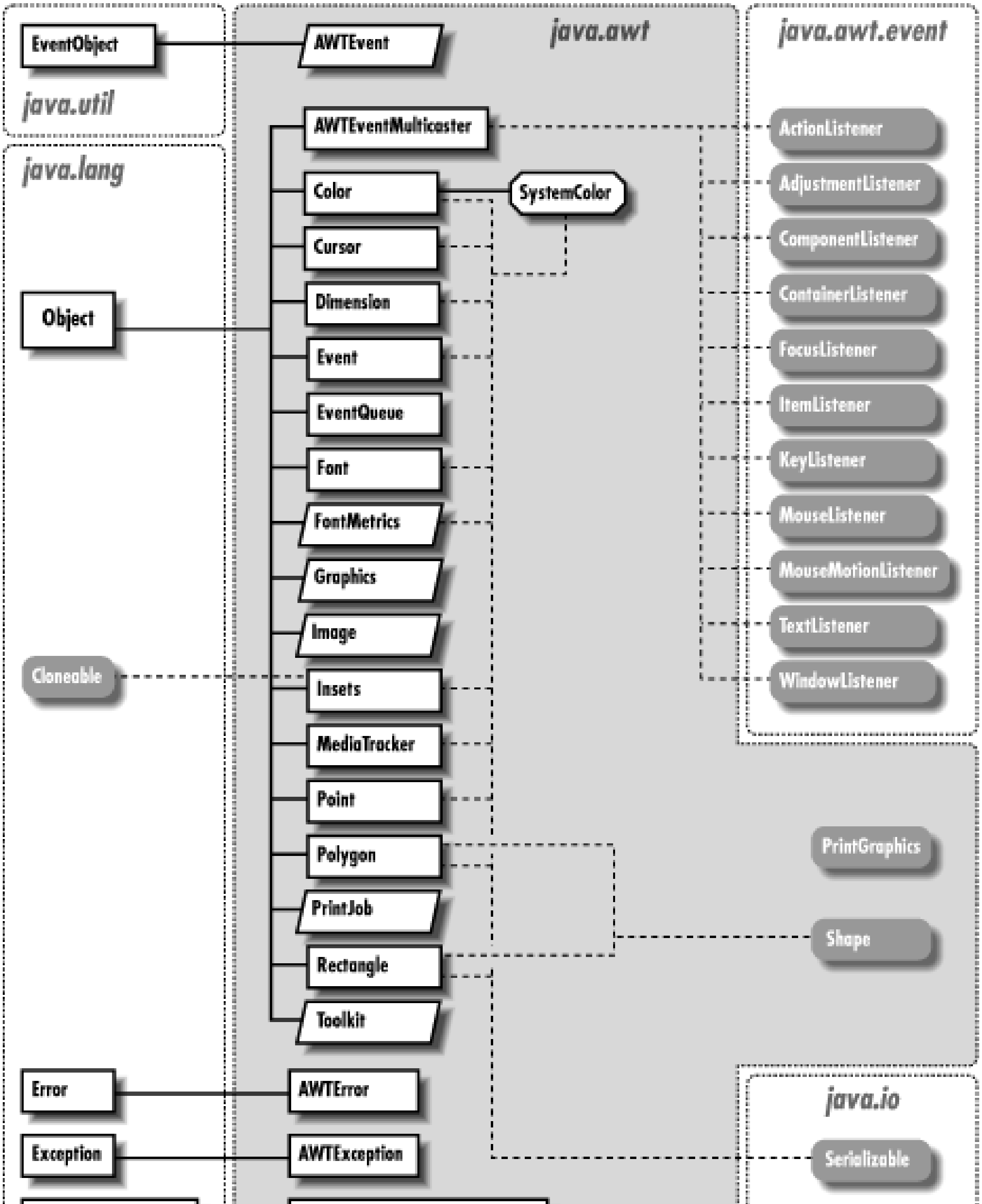
- Graphics: These classes define colors, fonts, images, polygons, and so forth.

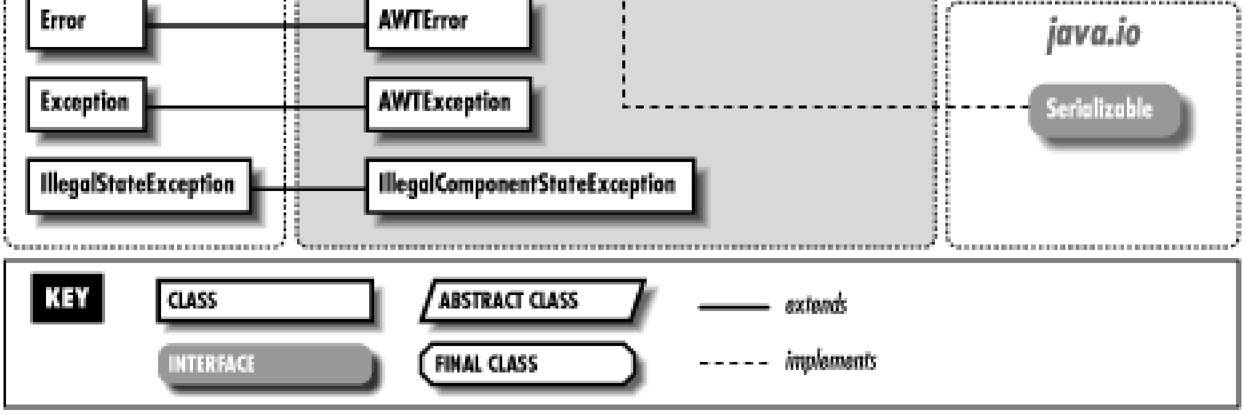
- Components: These classes are GUI (graphical user interface) components such as buttons, menus, lists, and dialog boxes.

- Layout Managers: These classes control the layout of components within their container objects.

Note that separate packages, `java.awt.datatransfer`, `java.awt.event`, and `java.awt.image`, contain classes for cut-and-paste, event handling, and image manipulation.

Figure 18.1: Graphics, event, and exception classes of the java.awt package





In the first category of classes, `Graphics` is probably the most important. This class defines methods for doing line and text drawing and image painting. It relies on other classes such as `Color`, `Font`, `Image`, and `Polygon`. `Image` is itself an important class, used in many places in `java.awt` and throughout the related package `java.awt.image`. `Event` is another important class that describes a user or window system event that has occurred. In Java 1.1, `Event` is superseded by the `AWTEvent` class.

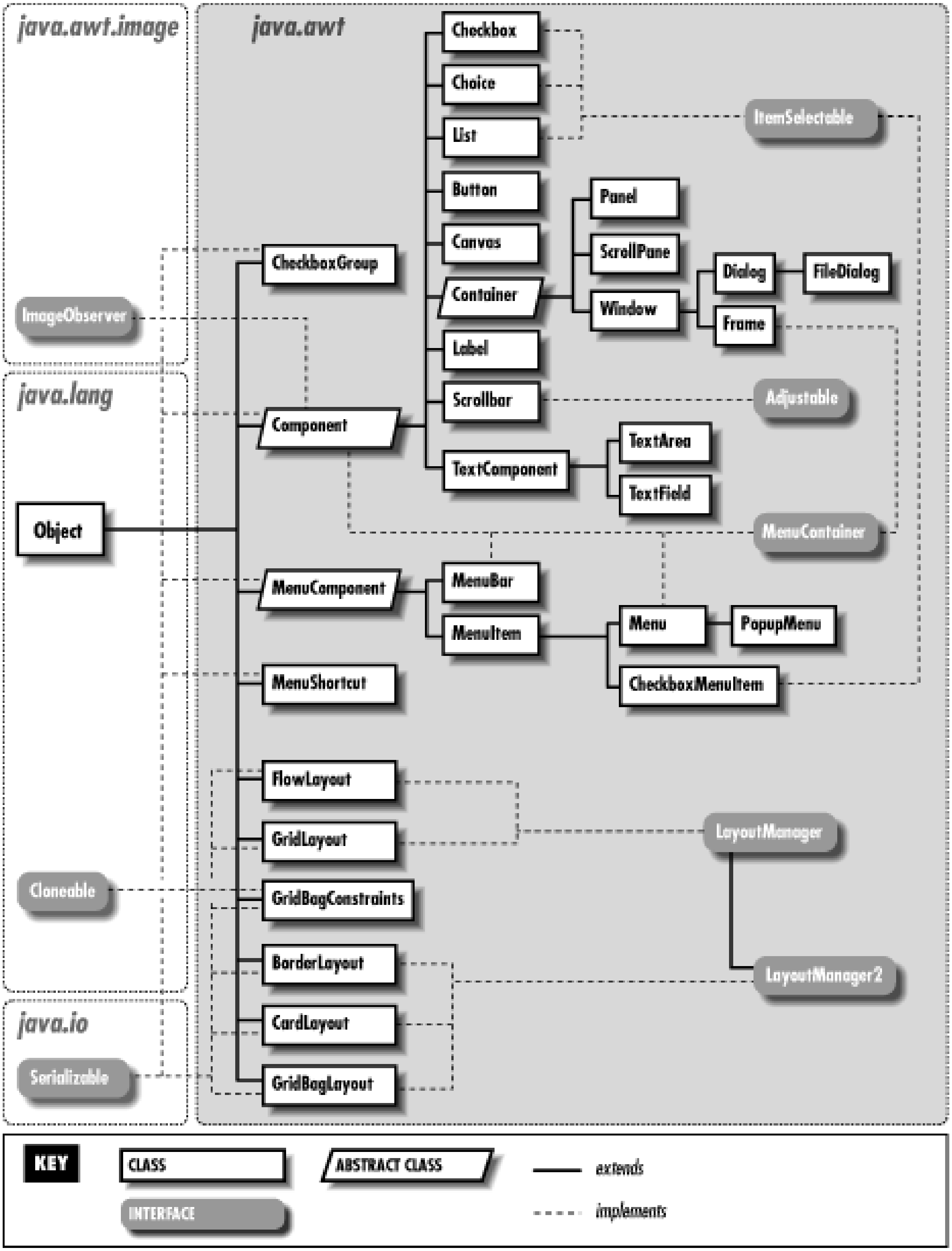
`Component` and `MenuComponent` are root classes in the second category of `java.awt` classes. Their subclasses are GUI components that can appear in interfaces and menus. The `Container` class is one that contains components and arranges them visually. You add components to a container with the `add()` method and specify a layout manager for the container with the `setLayout()` method.

There are three commonly used `Container` subclasses. `Frame` is a toplevel window that can contain a menu bar and have a custom cursor and an icon. `Dialog` is a dialog window. `Panel` is a container that does not have its own window--it is contained within some other container.

The third category of `java.awt` classes is the layout managers. The subclasses of `LayoutManager` are responsible for arranging the `Component` objects contained within a specified `Container`. `GridBagLayout`, `BorderLayout`, and `GridLayout` are probably the most useful of these layout managers.

See Chapter 8, *New AWT Features*, for examples of using some of the new Java 1.1 features of this package.

Figure 18.2: Component and layout classes of the java.awt package



18.1 java.awt.AWTError (JDK 1.0)

Signals that an error has occurred in the `java.awt` package.

```
public class AWTError extends Error {
    // Public Constructor
            public AWTError(String msg);
}
```

Hierarchy:

```
Object->Throwable(Serializable)->Error->AWTError
```

PREVIOUS
java.applet.AudioClip (JDK 1.0)

HOME
BOOK INDEX

NEXT
java.awt.AWTEvent (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

8. New AWT Features

Contents:

In addition to the new AWT event model that we saw in Chapter 7, *Events*, there are a number of important new AWT features in Java 1.1. These new features were outlined in Chapter 4, *What's New in Java 1.1*. This chapter details many of those new features and demonstrates them in a single extended example application at the end of the chapter. The major features of the example are:

- The new `ScrollPane` component
- Popup menus and menu shortcuts
- Printing
- Data transfer through cut-and-paste

In addition, the example also demonstrates the use of object serialization to save and load application state. This functionality is described in Chapter 9, *Object Serialization*.

8.1 The ScrollPane Container

The new `ScrollPane` container holds a single child component that is usually larger than the `ScrollPane` itself. The `ScrollPane` displays a fixed-size area of the child and provides horizontal and vertical scrollbars so the user can scroll the child component within the "viewport" of the `ScrollPane`. Figure 8.1 shows a top-level window created by the application listed in Example 8.1 at the end of this chapter. As you can see, the application creates a `ScrollPane` container to hold the larger `Scribble` component that supports free-hand drawing.

Figure 8.1: New AWT features demo application

[Graphic: Figure 8-1]

The ScrollPane is quite easy to use. Simply create it and add a child component as you would with any other container. Note, however, that ScrollPane only supports a single child and it cannot have a LayoutManager specified. The ScrollPane is created in the ScribbleFrame() constructor of the example. The important thing to note is that the ScrollPane does not have any preferred or natural size of its own, so you should use setSize() to specify the size you want it to be. The ScrollPane class defines three constants that are legal values of its "scrollbar display policy." Because the example does not specify one of these constants, the policy defaults to SCROLLBARS_AS_NEEDED, which indicates that scrollbars are displayed for any dimension in which the contained child is larger than the available "viewport" space of the ScrollPane container.

Here is an excerpt of the ScribbleFrame() constructor that shows the creation of the ScrollPane:

```
ScrollPane pane = new ScrollPane();      // Create a ScrollPane.
pane.setSize(300, 300);                  // Specify its size.
this.add(pane, "Center");                // Add it to the frame.
Scribble scribble;
scribble = new Scribble(this, 500, 500); // Create a bigger scribble area.
pane.add(scribble);                      // Add it to the ScrollPane.
```

PREVIOUS HOME NEXT

Inside the Java 1.1 Event Model

BOOK INDEX

Popup Menus and Menu Shortcuts

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

28.23 java.net.URLConnection (JDK 1.0)

This abstract class defines a network connection to an object specified by a URL. `URL.openConnection()` returns a `URLConnection` instance. You would use a `URLConnection` object when you want more control over the downloading of data than is available through the simpler `URL` methods.

`connect()` actually performs the network connection. Other methods that depend on being connected will call this method. `getContent()` returns the data referred to by the `URL`, parsed into an appropriate type of `Object`. If the URL protocol supports read and write operations, then `getInputStream()` and `getOutputStream()` respectively return input and output streams to the object referred to by the URL.

`getContentLength()`, `getContentType()`, `getContentEncoding()`, `getExpiration()`, `getDate()`, and `getLastModified()` return the appropriate information about the object referred to by the URL, if that information can be determined (e.g., from HTTP header fields). `getHeaderField()` returns an HTTP header field specified by name or by number. `getHeaderFieldInt()` and `getHeaderFieldDate()` return the value of a named header field parsed as an integer or a date.

There are a number of options that you may specify to control how the `URLConnection` behaves. These options are set with the various `set()` methods, and may be queried with corresponding `get()` methods. The options must be set before the `connect()` method is called. `setDoInput()` and `setDoOutput()` allow you to specify whether you use the `URLConnection` for input and/or output. The default is input-only. `setAllowUserInteraction()` specifies whether user interaction (such as typing a password) is allowed during the data transfer. The initial default is `false`. `setDefaultAllowUserInteraction()` is a class method that allows you to change the default value for user interaction. `setUseCaches()` allows you to specify whether a cached version of the URL may be used. You can set this to `false` to force a URL to be reloaded. `setDefaultUseCaches()` sets the default value for `setUseCaches()`. `setIfModifiedSince()` allows you to specify that a URL should not be fetched (if it is possible to determine its modification date) unless it has been modified since a specified time.

```
public abstract class URLConnection extends Object {
    // Protected Constructor
            protected URLConnection(URL url);
    // Class Variables
        1.1public static FileNameMap fileNameMap;
    // Protected Instance Variables
            protected boolean allowUserInteraction;
            protected boolean connected;
            protected boolean doInput;
            protected boolean doOutput;
            protected long ifModifiedSince;
            protected URL url;
            protected boolean useCaches;
    // Class Methods
            public static boolean getDefaultAllowUserInteraction();
            public static String getDefaultRequestProperty(String key);
            protected static String guessContentTypeFromName(String fname);
            public static String guessContentTypeFromStream(InputStream is) throws
IOException;
            public static synchronized void
setContentHandlerFactory(ContentHandlerFactory fac);
            public static void setDefaultAllowUserInteraction(boolean
defaultallowuserinteraction);
            public static void setDefaultRequestProperty(String key, String value);
    // Public Instance Methods
            public abstract void connect() throws IOException;
            public boolean getAllowUserInteraction();
            public Object getContent() throws IOException;
            public String getContentEncoding();
            public int getContentLength();
            public String getContentType();
            public long getDate();
            public boolean getDefaultUseCaches();
            public boolean getDoInput();
            public boolean getDoOutput();
            public long getExpiration();
            public String getHeaderField(String name);
            public String getHeaderField(int n);
            public long getHeaderFieldDate(String name, long Default);
            public int getHeaderFieldInt(String name, int Default);
            public String getHeaderFieldKey(int n);
            public long getIfModifiedSince();
            public InputStream getInputStream() throws IOException;
            public long getLastModified();
            public OutputStream getOutputStream() throws IOException;
            public String getRequestProperty(String key);
            public URL getURL();
            public boolean getUseCaches();
            public void setAllowUserInteraction(boolean allowuserinteraction);
            public void setDefaultUseCaches(boolean defaultusecaches);
            public void setDoInput(boolean doinput);
            public void setDoOutput(boolean dooutput);
            public void setIfModifiedSince(long ifmodifiedsince);
            public void setRequestProperty(String key, String value);
            public void setUseCaches(boolean usecaches);
            public String toString();  // Overrides Object
}
```

Extended By:

HttpURLConnection

Passed To:

ContentHandler.getContent()

Returned By:

URL.openConnection(), URLStreamHandler.openConnection()

PREVIOUS | HOME | NEXT

java.net.URL (JDK 1.0) | BOOK INDEX | java.net.URLEncoder (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

28.25 java.net.URLStreamHandler (JDK 1.0)

This abstract class defines the `openConnection()` method that creates a `URLConnection` for a given `URL`. A separate subclass of this class may be defined for various URL protocol types. A `URLStreamHandler` is created by a `URLStreamHandlerFactory`.

Normal applications never need to use or subclass this class.

```
public abstract class URLStreamHandler extends Object {
    // Default Constructor: public URLStreamHandler()
    // Protected Instance Methods
            protected abstract URLConnection openConnection(URL u) throws 
IOException;
            protected void parseURL(URL u, String spec, int start, int limit);
            protected void setURL(URL u, String protocol, String host, int port, 
String file, String ref);
            protected String toExternalForm(URL u);
}
```

Returned By:

URLStreamHandlerFactory.createURLStreamHandler()

13.2 Character Escape Sequences

Java uses the escape sequences listed in Table 13.2 to represent certain special character values. These escape sequences may appear in any Java `char` or `String` literal.

Table 13.2: Java Escape Characters

Escape Sequence	**Character Value**
\b	Backspace
\t	Horizontal tab
\n	Newline
\f	Form feed
\r	Carriage return
\"	Double quote
\'	Single quote
\\	Backslash
\xxx	The character corresponding to the octal value `xxx`, where `xxx` is between 000 and 0377.
\uxxxx	The Unicode character with encoding `xxxx`, where `xxxx` is one to four hexidecimal digits. Unicode escapes are distinct from the other escape types listed here; they are described below in more detail.

Java characters, strings, and identifiers (e.g., field, method, and class names) are composed of 16-bit Unicode characters. The Unicode `\u` escape sequence may be used anywhere in a Java program (not only in `char` and `String` literals) to represent a Unicode character.

Unicode `\u` escape sequences are processed before the other escape sequences described in Character Escape Sequences, and thus the two types of escape sequences can have very different semantics. A

Unicode escape is simply an alternative way to represent a character which may not be displayable on non-Unicode systems. The character escapes, however, can represent special characters in a way that prevents the usual interpretation of those characters by the compiler.

PREVIOUS | HOME | NEXT
Primitive Data Types | BOOK INDEX | Operators

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

13.4 Modifiers

There are quite a few Java keywords that serve as modifiers for Java classes, interfaces, methods, and fields. They are described in Table 13.4.

Table 13.4: Java Modifiers

Modifier	Used On	Meaning
abstract	class	The class contains unimplemented methods and cannot be instantiated.
	interface	All interfaces are `abstract`. The modifier is optional in interface declarations.
	method	No body is provided for the method (it is provided by a subclass). The signature is followed by a semicolon. The enclosing class must also be `abstract`.
final	class	The class may not be subclassed.
	field	The field may not have its value changed (compiler may precompute expressions).
	method	The method may not be overridden (compiler may optimize).
	variable	Java 1.1 and later: the local variable or method or exception parameter may not have its value changed.
native	method	The method is implemented in C, or in some other platform-dependent way. No body is provided; the signature is followed by a semicolon.
none (package)	class	A non-`public` class is accessible only in its package
	interface	A non-`public` interface is accessible only in its package
	member	A member that is not `private`, `protected`, or `public` has package visiblity and is accessible only within its package.
private	member	The member is only accessible within the class that defines it.

protected	member	The member is only accessible within the package in which it is defined, and within subclasses.
public	class	The class is accessible anywhere its package is.
	interface	The interface is accessible anywhere its package is.
	member	The member is accessible anywhere its class is.
static	class	In Java 1.1, a class delared `static` is a toplevel class, not an inner class.
	field	A `static` field is a "class field." There is only one instance of the field, regardless of the number of class instances created. It may be accessed through the class name.
	initializer	The intializer is run when the class is loaded, rather than when an instance is created.
	method	A `static` method is a "class method." It is not passed as an implicit `this` object reference. It may be invoked through the class name.
synchronized	method	The method makes non-atomic modifications to the class or instance, and care must be taken to ensure that two threads cannot modify the class or instance at the same time. For a `static` method, a lock for the class is acquired before executing the method. For a non-`static` method, a lock for the specific object instance is acquired.
transient	field	The field is not part of the persistent state of the object, and should not be serialized with the object.
volatile	field	The field may be accessed by unsynchronized threads, so certain code optimizations must not be performed on it.

Table 13.5 summarizes the visibility modifiers; it shows the circumstances under which class members of the various visibility types are accessible.

Table 13.5: Class Member Accessibility

Accessible to:	**Member Visibility**			
	public	**protected**	**package**	**private**
Same class	yes	yes	yes	yes
Class in same package	yes	yes	yes	no
Subclass in different package	yes	yes	no	no
Non-subclass, different package	yes	no	no	no

PREVIOUS HOME NEXT

13.5 Reserved Words

Table 13.6 lists reserved words in Java. These are keywords or `boolean` literal values. Note that `byvalue`, `cast`, `const`, `future`, `generic`, `goto`, `inner`, `operator`, `outer`, `rest`, and `var` are reserved by Java but currently unused.

Table 13.6: Java Reserved Words

abstract	default	goto	operator	synchronized
boolean	do	if	outer	this
break	double	implements	package	throw
byte	else	import	private	throws
byvalue	extends	inner	protected	transient
case	false	instanceof	public	true
cast	final	int	rest	try
catch	finally	interface	return	var
char	float	long	short	void
class	for	native	static	volatile
const	future	new	super	while
continue	generic	null	switch	

Reserved Method Names

Table 13.7 lists the method names of the `Object` class. Strictly speaking, these method names are not reserved, but since the methods are inherited by every class, they should not be used as the name of a method, except when intentionally overriding an `Object` method.

Table 13.7: Reserved Method Names

clone	getClass	notifyAll
equals	hashCode	toString
finalize	notify	wait

PREVIOUS
Modifiers

HOME
BOOK INDEX

NEXT
Java Documentation Comment Syntax

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Acknowledgments

Many people contributed to putting this book together under a schedule that became increasingly rushed as time passed. Thanks to their efforts, we gave birth to something we can all be proud of.

Foremost we would like to thank Tim O'Reilly for giving us the opportunity to write this book. Special thanks to Mike Loukides, the series editor, whose endless patience and experience got us through the difficult parts and to Paula Ferguson, whose organizational and editing abilities got the material into its final form. It's due largely to Mike and Paula's tireless efforts that this book has gotten to you as quickly as it has. We could not have asked for a more skillful or responsive team of people with whom to work.

Particular thanks are due to our technical reviewers: Andrew Cohen, Eric Raymond, and Lisa Farley. All of them gave thorough reviews that were invaluable in assembling the final draft. Eric contributed many bits of text that eventually found their way into the book.

Speaking of borrowings, the original version of the glossary came from David Flanagan's book, *Java in a Nutshell*. We also borrowed the class hierarchy diagrams from David's book. These diagrams were based on similar diagrams by Charles L. Perkins. His original diagrams are available at http://rendezvous.com/java/.

Thanks also to Marc Wallace and Steven Burkett for reading the book in progress. As for the crowd in St. Louis: a special thanks to LeeAnn Langdon of the Library Ltd. and Kerri Bonasch. Deepest thanks to Victoria Doerr for her patience and love. Finally, thanks for the support of the "lunch" crowd: Karl "Gooch" Stefvater, Bryan "Butter" O'Connor, Brian "Brian" Gottlieb, and the rest of the clan at Washington University.

Many people in O'Reilly's production and design groups contributed their blood, sweat, and tears to the project. Mary Anne Weeks Mayo was production editor and copy editor, and had the stress-filled job of working under a very tight deadline with chapters arriving asynchronously (which means at random and later than expected). Seth Maislin wrote the index, and Stephen Spainhour adapted David Flanagan's glossary for this book. Chris Reilley converted rough diagrams into professional technical illustrations. Erik Ray, Ellen Siever, and Lenny Muellner converted HTML files into SGML and made sure we could convert electrons into paper without mishap. Lenny also implemented a new design for this book, which was created by Nancy Priest. Hanna Dyer created the back cover; Edie Freedman designed the front cover.

PREVIOUS | HOME | NEXT
Conventions Used in This Book | BOOK INDEX | Yet Another Language?

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

1.2 A Virtual Machine

Java is both a compiled and an interpreted language. Java source code is turned into simple binary instructions, much like ordinary microprocessor machine code. However, whereas C or C++ source is refined to native instructions for a particular model of processor, Java source is compiled into a universal format--instructions for a virtual machine.

Compiled Java byte-code, also called J-code, is executed by a Java run-time interpreter. The run-time system performs all the normal activities of a real processor, but it does so in a safe, virtual environment. It executes the stack-based instruction set and manages a storage heap. It creates and manipulates primitive data types, and loads and invokes newly referenced blocks of code. Most importantly, it does all this in accordance with a strictly defined open specification that can be implemented by anyone who wants to produce a Java-compliant virtual machine. Together, the virtual machine and language definition provide a complete specification. There are no features of Java left undefined or implementation dependent. For example, Java specifies the sizes of all its primitive data types, rather than leave it up to each implementation.

The Java interpreter is relatively lightweight and small; it can be implemented in whatever form is desirable for a particular platform. On most systems, the interpreter is written in a fast, natively compiled language like C or C++. The interpreter can be run as a separate application, or it can be embedded in another piece of software, such as a Web browser.

All of this means that Java code is implicitly portable. The same Java application can run on any platform that provides a Java run-time environment, as shown in Figure 1.1. You don't have to produce alternate versions of your application for different platforms, and you never have to distribute source code to end users.

Figure 1.1: Java virtual machine environment

[Graphic: Figure 1-1]

The fundamental unit of Java code is the *class*. As in other object-oriented languages, classes are application components that hold executable code and data. Compiled Java classes are distributed in a universal binary format that contains Java byte-code and other class information. Classes can be maintained discretely and stored in files or archives on a local system or on a network server. Classes are located and loaded dynamically at run-time, as they are needed by an application.

In addition to the platform-specific run-time system, Java has a number of fundamental classes that contain architecture-dependent methods. These *native methods* serve as Java's gateway to the real world. These methods are implemented in a native language on the host platform. They provide access to resources such as the network, the windowing system, and the host filesystem. The rest of Java is written entirely in Java, and is therefore portable. This includes fundamental Java utilities like the Java compiler, which is also a Java application and is therefore immediately available on all Java platforms.

In general, interpreters are slow, but because the Java interpreter runs compiled byte-code, Java is a fast interpreted language. Java has also been designed so that software implementations of the run-time system can optimize their performance by compiling byte-code to native machine code on the fly. This is called "just in time" compilation. Sun claims that with just in time compilation, Java code can execute nearly as fast as native compiled code and maintain its transportability and security. The one performance hit that natively compiled Java code will always suffer is array bounds checking. But on the other hand, some of the basic design features of Java place more information in the hands of the compiler, which allows for certain kinds of optimizations not possible in C or C++.

PREVIOUS Enter Java

HOME

BOOK INDEX

NEXT Java Compared

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

1.3 Java Compared

Java is a new language, but it draws on many years of programming experience with other languages in its choice of features. So a lot can be said in comparing and contrasting Java with other languages. There are at least three pillars necessary to support a universal language for network programming today: portability, speed, and security. Figure 1.2 shows how Java compares to other languages.

Figure 1.2: Programming languages compared

[Graphic: Figure 1-2]

You may have heard that Java is a lot like C or C++, but that's really not true, except at a superficial level. When you first look at Java code, you'll see that the basic syntax looks a lot like C or C++. But that's where the similarities end. Java is by no means a direct descendant of C or a next generation C++. If you compare language features, you'll see that Java actually has more in common with languages like Smalltalk and Lisp. In fact, Java's implementation is about as far from native C as you can imagine.

The surface-level similarities to C and C++ are worth noting, however. Java borrows heavily from C and C++ syntax, so you'll see lots of familiar language constructs, including an abundance of curly braces and semicolons. Java also subscribes to the C philosophy that a good language should be compact; in other words, it should be sufficiently small and regular so that a programmer can hold all the language's capabilities in his or her head at once. As C is extensible with libraries, packages of Java classes can be added to the core language components.

C has been successful because it provides a reasonably featureful programming environment, with high performance and an acceptable degree of portability. Java also tries to balance functionality, speed, and portability, but it does so in a very different way. While C trades functionality to get portability, Java trades speed for portability. Java also addresses security issues, while C doesn't.

Java is an interpreted language, so it won't be as fast as a compiled language like C. But Java is fast enough, especially for interactive, network-based applications, where the application is often idle, waiting for the user to do something or waiting for data from the network. For situations where speed is critical, a Java implementation can optimize performance by compiling byte-code to native machine code on the fly.

Scripting languages, like Tcl, Perl, and Wksh, are becoming quite popular, and for good reason. There's no reason a scripting language could not be suitable for safe, networked applications (e.g., Safe Tcl), but most scripting languages are not designed for serious, large-scale programming. The attraction to scripting languages is that they are dynamic; they are powerful tools for rapid prototyping. Some scripting languages, like awk and Perl, also provide powerful tools for text-processing tasks more general-purpose languages find unwieldy. Scripting languages are also highly portable.

One problem with scripting languages, however, is that they are rather casual about program structure and data typing. Most scripting languages (with a hesitant exception for Perl 5.0) are not object oriented. They also have vastly simplified type systems and generally don't provide for sophisticated scoping of variables and functions. These characteristics make them unsuitable for building large, modular applications. Speed is another problem with scripting languages; the high-level, fully interpreted nature of these languages often makes them quite slow.

Java offers some of the essential advantages of a scripting language, along with the added benefits of a lower-level language.[1] Incremental development with object-oriented components, combined with Java's simplicity, make it possible to develop applications rapidly and change them easily, with a short concept to implementation time. Java also comes with a large base of core classes for common tasks such as building GUIs and doing network communications. But along with these features, Java has the scalability and software-engineering advantages of more static languages. It provides a safe structure on which to build higher-level networked tools and languages.

> [1] Don't confuse Java with JavaScript. JavaScript is an object-based scripting language being developed by Netscape and is designed to create dynamic, interactive Web applications. JavaScript is a very different language from Java in most respects. For more information on JavaScript, check out Netscape's Web site (http://home.netscape.com).

As I've already said, Java is similar in design to languages like Smalltalk and Lisp. However, these languages are used mostly as research vehicles, rather than for developing large-scale systems. One reason is that they never developed a standard portable binding to operating-system services analogous to the C standard library or the Java core classes. Smalltalk is compiled to an interpreted byte-code format, and it can be dynamically compiled to native code on the fly, just like Java. But Java improves on the design by using a byte-code verifier to ensure the correctness of Java code. This verifier gives Java a performance advantage over Smalltalk because Java code requires fewer run-time checks. Java's byte-code verifier also helps with security issues, something that Smalltalk doesn't address. Smalltalk is a mature language though, and Java's designers took lessons from many of its features.

Throughout the rest of this chapter, we'll take a bird's eye view of the Java language. I'll explain what's new and what's not so new about Java; how it differs from other languages, and why.

PREVIOUS | HOME | NEXT
A Virtual Machine | BOOK INDEX | Safety of Design

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

1.4 Safety of Design

You have no doubt heard a lot about the fact that Java is designed to be a safe language. But what do we mean by safe? Safe from what or whom? The security features that attract the most attention for Java are those features that make possible new types of dynamically portable software. Java provides several layers of protection from dangerously flawed code, as well as more mischievous things like viruses and trojan horses. In the next section, we'll take a look at how the Java virtual machine architecture assesses the safety of code before it's run, and how the Java class loader builds a wall around untrusted classes. These features provide the foundation for high-level security policies that allow or disallow various kinds of activities on an application-by-application basis.

In this section though, we'll look at some general features of the Java programming language. What is perhaps more important, although often overlooked in the security din, is the safety that Java provides by addressing common design and programming problems. Java is intended to be as safe as possible from the simple mistakes we make ourselves, as well as those we inherit from contractors and third-party software vendors. The goal with Java has been to keep the language simple, provide tools that have demonstrated their usefulness, and let users build more complicated facilities on top of the language when needed.

Syntactic Sweet 'n Low

Java is parsimonious in its features; simplicity rules. Compared to C, Java uses few automatic type coercions, and the ones that remain are simple and well-defined. Unlike C++, Java doesn't allow programmer-defined operator overloading. The string concatenation operator `+` is the only system-defined, overloaded operator in Java. All methods in Java are like C++ virtual methods, so overridden methods are dynamically selected at run-time.

Java doesn't have a preprocessor, so it doesn't have macros, `#define` statements, or conditional source compilation. These constructs exist in other languages primarily to support platform dependencies, so in that sense they should not be needed in Java. Another common use for conditional compilation is for debugging purposes. Debugging code can be included directly in your Java source code by making it conditional on a constant (i.e., `static` and `final`) variable. The Java compiler is smart enough to remove this code when it determines that it won't be called.

Java provides a well-defined package structure for organizing class files. The package system allows the compiler to handle most of the functionality of *make*. The compiler also works with compiled Java classes because all type information is preserved; there is no need for header files. All of this means that Java code requires little context to read. Indeed, you may sometimes find it faster to look at the Java source code than to refer to class documentation.

Java replaces some features that have been troublesome in other languages. For example, Java supports only a single inheritance class hierarchy, but allows multiple inheritance of interfaces. An interface, like an abstract class in C++, specifies the behavior of an object without defining its implementation, a powerful mechanism borrowed from Objective C. It allows a class to implement the behavior of the interface, without needing to be a subclass of the interface. Interfaces in Java eliminate the need for multiple inheritance of classes, without causing the problems associated with multiple inheritance.

As you'll see in Chapter 4, *The Java Language*, Java is a simple, yet elegant, programming language.

Type Safety and Method Binding

One attribute of a language is the kind of type checking it uses. When we categorize a language as static or dynamic we are referring to the amount of information about variable types that is known at compile time versus what is determined while the application is running.

In a strictly statically typed language like C or C++, data types are etched in stone when the source code is compiled. The compiler benefits from having enough information to enforce usage rules, so that it can catch many kinds of errors before the code is executed. The code doesn't require run-time type checking, which is advantageous because it can be compiled to be small and fast. But statically typed languages are inflexible. They don't support high-level constructs like lists and collections as naturally as languages with dynamic type checking, and they make it impossible for an application to safely import new data types while it's running.

In contrast, a dynamic language like Smalltalk or Lisp has a run-time system that manages the types of objects and performs necessary type checking while an application is executing. These kinds of languages allow for more complex behavior, and are in many respects more powerful. However, they are also generally slower, less safe, and harder to debug.

The differences in languages have been likened to the differences between kinds of automobiles.[2] Statically typed languages like C++ are analogous to a sports car--reasonably safe and fast--but useful only if you're driving on a nicely paved road. Highly dynamic languages like Smalltalk are more like an offroad vehicle: they afford you more freedom, but can be somewhat unwieldy. It can be fun (and sometimes faster) to go roaring through the back woods, but you might also get stuck in a ditch or mauled by bears.

[2] The credit for the car analogy goes to Marshall P. Cline, author of the C++ FAQ.

Another attribute of a language deals with when it binds method calls to their definitions. In an early-binding language like C or C++, the definitions of methods are normally bound at compile-time, unless the programmer specifies otherwise. Smalltalk, on the other hand, is a late-binding language because it locates the definitions of methods dynamically at run-time. Early-binding is important for performance reasons; an application can run without the overhead incurred by searching method tables at run-time. But late-binding is more flexible. It's also necessary in an object-oriented language, where a subclass can override methods in its superclass, and only the run-time system can determine which method to run.

Java provides some of the benefits of both C++ and Smalltalk; it's a statically typed, late-binding language. Every object in Java has a well-defined type that is known at compile time. This means the Java compiler can do the same kind of static type checking and usage analysis as C++. As a result, you can't assign an object to the wrong type of reference or call nonexistent methods within it. The Java compiler goes even further and prevents you from messing up and trying to use uninitialized variables.

However, Java is fully run-time typed as well. The Java run-time system keeps track of all objects and makes it possible to determine their types and relationships during execution. This means you can inspect an object at run-time to determine what it is. Unlike C or C++, casts from one type of object to another are checked by the run-time system, and it's even possible to use completely new kinds of dynamically loaded objects with a level of type safety.

Since Java is a late-binding language, all methods are like virtual methods in C++. This makes it possible for a subclass to override methods in its superclass. But Java also allows you to gain the performance benefits of early-binding by declaring (with the `final` modifier) that certain methods can't be overridden by subclassing, removing the need for run-time lookup.

Incremental Development

Java carries all data-type and method-signature information with it from its source code to its compiled byte-code form. This means that Java classes can be developed incrementally. Your own Java classes can also be used safely with classes from other sources your compiler has never seen. In other words, you can write new code that references binary class files, without losing the type safety you gain from having the source code. The Java run-time system can load new classes while an application is running, thus providing the capabilities of a dynamic linker.

A common irritation with C++ is the "fragile base class" problem. In C++, the implementation of a base class can be effectively frozen by the fact that it has many derived classes; changing the base class may require recompilation of the derived classes. This is an especially difficult problem for developers of class libraries. Java avoids this problem by dynamically locating fields within classes. As long as a class maintains a valid form of its original structure, it can evolve without breaking other classes that are derived from it or that make use of it.

Dynamic Memory Management

Some of the most important differences between Java and C or C++ involve how Java manages memory. Java eliminates ad hoc pointers and adds garbage collection and true arrays to the language. These features eliminate many otherwise insurmountable problems with safety, portability, and optimization.

Garbage collection alone should save countless programmers from the single largest source of programming errors in C or C++: explicit memory allocation and deallocation. In addition to maintaining objects in memory, the Java run-time system keeps track of all references to those objects. When an object is no longer in use, Java automatically removes it from memory. You can simply throw away references to objects you no longer use and have confidence that the system will clean them up at an appropriate time. Sun's current implementation of Java uses a conservative, mark and sweep garbage collector that runs intermittently in the background, which means that most garbage collecting takes place between mouse clicks and keyboard hits. Once you get used to garbage collection, you won't go back. Being able to write air-tight C code that juggles memory without dropping any on the floor is an important skill, but once you become addicted to Java you can "realloc" some of those brain cells to new tasks.

You may hear people say that Java doesn't have pointers. Strictly speaking, this statement is true, but it's also misleading. What Java provides are *references*--a safe kind of pointer--and Java is rife with them. A reference is a strongly typed handle for an object. All objects in Java, with the exception of primitive numeric types, are accessed through references. If necessary, you can use references to build all the normal kinds of data structures you're accustomed to building with pointers, like linked lists, trees, and so forth. The only difference is that with references you have to do so in a type-safe way.

Another important difference between a reference and a pointer is that you can't do pointer arithmetic with references. A reference is an atomic thing; you can't manipulate the value of a reference except by assigning it to an object. References are passed by value, and you can't reference an object through more than a single level of indirection. The protection of references is one of the most fundamental aspects of Java security. It means that Java code has to play by the rules; it can't peek into places it shouldn't.

Unlike C or C++ pointers, Java references can point only to class types. There are no pointers to methods. People often complain about this missing feature, but you will find that most tasks that call for pointers to methods, such as callbacks, can be accomplished using interfaces and anonymous adapter classes instead. [3]

> [3] Java 1.1 does have a `Method` class, which lets you have a reference to a method. You can use a `Method` object to construct a callback, but it's tricky.

Finally, arrays in Java are true, first-class objects. They can be dynamically allocated and assigned like other objects. Arrays know their own size and type, and although you can't directly define array classes or subclass them, they do have a well-defined inheritance relationship based on the relationship of their base types. Having true arrays in the language alleviates much of the need for pointer arithmetic like that in C or C++.

Error Handling

Java's roots are in networked devices and embedded systems. For these applications, it's important to have robust and intelligent error management. Java has a powerful exception-handling mechanism, somewhat like that in newer implementations of C++. Exceptions provide a more natural and elegant way to handle errors. Exceptions allow you to separate error-handling code from normal code, which makes for cleaner, more readable applications.

When an exception occurs, it causes the flow of program execution to be transferred to a predesignated catcher block of code. The exception carries with it an object that contains information about the situation that caused the exception. The Java compiler requires that a method either declare the exceptions it can generate or catch and deal with them itself. This promotes error information to the same level of importance as argument and return typing. As a Java programmer, you know precisely what exceptional conditions you must deal with, and you have help from the compiler in writing correct software that doesn't leave them unhandled.

Multithreading

Applications today require a high degree of parallelism. Even a very single-minded application can have a complex user interface. As machines get faster, users become more sensitive to waiting for unrelated tasks that seize control of their time. Threads provide efficient multiprocessing and distribution of tasks. Java makes threads easy to use because support for them is built into the language.

Concurrency is nice, but there's more to programming with threads than just performing multiple tasks simultaneously. In many cases, threads need to be synchronized, which can be tricky without explicit language support. Java supports synchronization based on the monitor and condition model developed by C.A.R. Hoare--a sort of lock and key system for accessing resources. A keyword, `synchronized`, designates methods for safe, serialized access within an object. Only one synchronized method within the object may run at a given time. There are also simple, primitive methods for explicit waiting and signaling between threads interested in the same object.

Learning to program with threads is an important part of learning to program in Java. See Chapter 6, *Threads* for a discussion of this topic. For complete coverage of threads, see *Java Threads*, by Scott Oaks and Henry Wong (O'Reilly & Associates).

Scalability

At the lowest level, Java programs consist of classes. Classes are intended to be small, modular components. They can be separated physically on different systems, retrieved dynamically, and even cached in various distribution schemes. Over classes, Java provides packages, a layer of structure that groups classes into functional units. Packages provide a naming convention for organizing classes and a second level of organizational control over the visibility of variables and methods in Java applications.

Within a package, a class is either publicly visible or protected from outside access. Packages form another type of scope that is closer to the application level. This lends itself to building reusable components that work together in a system. Packages also help in designing a scalable application that can grow without becoming a bird's nest of tightly coupled code dependency.

PREVIOUS | HOME | NEXT
Java Compared | BOOK INDEX | Safety of Implementation

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

1.5 Safety of Implementation

It's one thing to create a language that prevents you from shooting yourself in the foot; it's quite another to create one that prevents others from shooting you in the foot.

Encapsulation is a technique for hiding data and behavior within a class; it's an important part of object-oriented design. It helps you write clean, modular software. In most languages however, the visibility of data items is simply part of the relationship between the programmer and the compiler. It's a matter of semantics, not an assertion about the actual security of the data in the context of the running program's environment.

When Bjarne Stroustrup chose the keyword `private` to designate hidden members of classes in C++, he was probably thinking about shielding you from the messy details of a class developer's code, not the issues of shielding that developer's classes and objects from the onslaught of someone else's viruses and trojan horses. Arbitrary casting and pointer arithmetic in C or C++ make it trivial to violate access permissions on classes without breaking the rules of the language. Consider the following code:

```
// C++

class Finances {
    private:
        char creditCardNumber[16];
        ...
    };

main() {
    Finances finances;

    // Forge a pointer to peek inside the class
    char *cardno = (char *)finances;
    printf("Card Number = %s\n", cardno);
}
```

In this little C++ drama, we have written some code that violates the encapsulation of the `Finances` class and pulls out some secret information. If this example seems unrealistic, consider how important it is to protect the foundation (system) classes of the run-time environment from similar kinds of attacks. If untrusted code can corrupt the components that provide access to real resources, such as the filesystem, the network, or the windowing system, it certainly has a chance at stealing your credit card numbers.

In Visual BASIC, it's also possible to compromise the system by peeking, poking, and, under DOS, installing interrupt handlers. Even some recent languages that have some commonalities with Java lack important security features. For example, the Apple Newton uses an object-oriented language called NewtonScript that is compiled into an interpreted byte-code format. However, NewtonScript has no concept of public and private members, so a Newton application has free reign to access any information it finds. General Magic's Telescript language is another example of a device-independent language that does not fully address security concerns.

If a Java application is to dynamically download code from an untrusted source on the Internet and run it alongside applications that might contain confidential information, protection has to extend very deep. The Java security model wraps three layers of protection around imported classes, as shown in Figure 1.3.

Figure 1.3: The Java security model

[Graphic: Figure 1-3]

At the outside, application-level security decisions are made by a security manager. A security manager controls access to system resources like the filesystem, network ports, and the windowing environment. A security manager relies on the ability of a class loader to protect basic system classes. A class loader handles loading classes from the network. At the inner level, all system security ultimately rests on the Java verifier, which guarantees the integrity of incoming classes.

The Java byte-code verifier is an integral part of the Java run-time system. Class loaders and security managers, however, are implemented by applications that load applets, such as applet viewers and Web browsers. All these pieces need to be functioning properly to ensure security in the Java environment.[4]

[4] You may have seen reports about various security flaws in Java. While these weaknesses are real, it's important to realize that they have been found in the implementations of various components, namely Sun's byte-code verifier and Netscape's class loader and security manager, not in the basic security model itself. One of the reasons Sun has released the source code for Java is to encourage people to search for weaknesses, so they can be removed.

The Verifier

Java's first line of defense is the byte-code *verifier*. The verifier reads byte-code before they are run and makes sure they are well-behaved and obey the basic rules of the Java language. A trusted Java compiler won't produce code that does otherwise. However, it's possible for a mischievous person to deliberately assemble bad code. It's the verifier's job to detect this.

Once code has been verified, it's considered safe from certain inadvertent or malicious errors. For example, verified code can't forge pointers or violate access permissions on objects. It can't perform illegal casts or use objects in ways other than they are intended. It can't even cause certain types of internal errors, such as overflowing or underflowing the operand stack. These fundamental guarantees underlie all of Java's security.

You might be wondering, isn't this kind of safety implicit in lots of interpreted languages? Well, while it's true that you shouldn't be able to corrupt the interpreter with bogus BASIC code, remember that the protection in most interpreted languages happens at a higher level. Those languages are likely to have heavyweight interpreters that do a great deal of run-time work, so they are necessarily slower and more cumbersome.

By comparison, Java byte-code is a relatively light, low-level instruction set. The ability to statically verify the Java byte-code before execution lets the Java interpreter run at full speed with full safety, without expensive run-time checks. Of course, you are always going to pay the price of running an interpreter, but that's not a serious problem with the speed of modern CPUs. Java byte-code can also be compiled on the fly to native machine code, which has even less run-time overhead.

The verifier is a type of theorem prover. It steps through the Java byte-code and applies simple, inductive rules to determine certain aspects of how the byte-code will behave. This kind of analysis is possible because compiled Java byte-code has a lot more type information stored within it than other languages of this kind. The byte-code also has to obey a few extra rules that simplify its behavior. First, most byte-code instructions operate only on individual data types. For example, with stack operations, there are separate instructions for object references and for each of the numeric types in Java. Similarly, there is a different instruction for moving each type of value into and out of a local variable.

Second, the type of object resulting from any operation is always known in advance. There are no byte-code operations that consume values and produce more than one possible type of value as output. As a result, it's always possible to look at the next instruction and its operands, and know the type of value that will result.

Because an operation always produces a known type, by looking at the starting state, it's possible to determine the types of all items on the stack and in local variables at any point in the future. The collection of all this type information at any given time is called the *type state* of the stack; this is what Java tries to analyze before it runs an application. Java doesn't know anything about the actual values of stack and variable items at this time, just what kind of items they are. However, this is enough information to enforce the security rules and to insure that objects are not manipulated illegally.

To make it feasible to analyze the type state of the stack, Java places an additional restriction on how Java byte-code instructions are executed: all paths to the same point in the code have to arrive with exactly the same type state.[5] This restriction makes it possible for the verifier to trace each branch of the code just once and still know the type state at all points. Thus, the verifier can insure that instruction types and stack value types always correspond, without actually following the execution of the code.

> [5] The implications of this rule are mainly of interest to compiler writers. The rule means that Java byte-code can't perform certain types of iterative actions within a single frame of execution. A common example would be looping and pushing values onto the stack. This is not allowed because the path of execution would return to the top of the loop with a potentially different type state on each pass, and there is no way that a static analysis of the code can determine whether it obeys the security rules.

Class Loader

Java adds a second layer of security with a *class loader*. A class loader is responsible for bringing Java binary classes that contain byte-code into the interpreter. Every application that loads classes from the network must use a class loader to handle this task.

After a class has been loaded and passed through the verifier, it remains associated with its class loader. As a result, classes are effectively partitioned into separate namespaces based on their origin. When a class references another class, the request is served by its original class loader. This means that classes retrieved from a specific source can be restricted to interact only with other classes retrieved from that same location. For example, a Java-enabled Web browser can use a class loader to build a separate space for all the classes loaded from a given uniform resource locator (URL).

The search for classes always begins with the built-in Java system classes. These classes are loaded from the locations specified by the Java interpreter's class path (see Chapter 3, *Tools of the Trade*). Classes in the class path are loaded by the system only once and can't be replaced. This means that it's impossible for an applet to replace fundamental system classes with its own versions that change their functionality.

Security Manager

Finally, a security manager is responsible for making application-level security decisions. A security manager is an object that can be installed by an application to restrict access to system resources. The security manager is consulted every time the application tries to access items like the filesystem, network ports, external processes, and the windowing environment, so the security manager can allow or deny the request.

A security manager is most useful for applications that run untrusted code as part of their normal operation. Since a Java-enabled Web browser can run applets that may be retrieved from untrusted sources on the Net, such a browser needs to install a security manager as one of its first actions. This security manager then restricts the kinds of access allowed after that point. This lets the application impose an effective level of trust before running an arbitrary piece of code. And once a security manager is installed, it can't be replaced.

A security manager can be as simple or complex as a particular application warrants. Sometimes it's sufficient simply to deny access to all resources or to general categories of services such as the filesystem or network. But it's also possible to make sophisticated decisions based on high-level information. For example, a Java-enabled Web browser could implement a security manager that lets users specify how much an applet is to be trusted or that allows or denies access to specific resources on a case-by-case basis. Of course, this assumes that the browser can determine which applets it ought to trust. We'll see how this problem is solved shortly.

The integrity of a security manager is based on the protection afforded by the lower levels of the Java security model. Without the guarantees provided by the verifier and the class loader, high-level assertions about the safety of system resources are meaningless. The safety provided by the Java byte-code verifier means that the interpreter can't be corrupted or subverted, and that Java code has to use components as they are intended. This, in turn, means that a class loader can guarantee that an application is using the core Java system classes and that these classes are the only means of accessing basic system resources. With these restrictions in place, it's possible to centralize control over those resources with a security manager.

PREVIOUS | HOME | NEXT
Safety of Design | BOOK INDEX | Application and User Level Security

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

1.6 Application and User Level Security

There's a fine line between having enough power to do something useful and having all the power to do anything you want. Java provides the foundation for a secure environment in which untrusted code can be quarantined, managed, and safely executed. However, unless you are content with keeping that code in a little black box and running it just for its own benefit, you will have to grant it access to at least some system resources so that it can be useful. Every kind of access carries with it certain risks and benefits. The advantages of granting an untrusted applet access to your windowing system, for example, are that it can display information and let you interact in a useful way. The associated risks are that the applet may instead display something worthless, annoying, or offensive. Since most people can accept that level of risk, graphical applets and the World Wide Web in general are possible.

At one extreme, the simple act of running an application gives it a resource, computation time, that it may put to good use or burn frivolously. It's difficult to prevent an untrusted application from wasting your time, or even attempting a "denial of service" attack. At the other extreme, a powerful, trusted application may justifiably deserve access to all sorts of system resources (e.g., the filesystem, process creation, network interfaces); a malicious application could wreak havoc with these resources. The message here is that important and sometimes complex security issues have to be addressed.

In some situations, it may be acceptable to simply ask the user to "OK" requests. Sun's HotJava Web browser can pop up a dialog box and ask the user's permission for an applet to access an otherwise restricted file. However, we can put only so much burden on our users. An experienced person will quickly grow tired of answering questions; an inexperienced user may not even be able to answer the questions. Is it okay for me to grant an applet access to something if I don't understand what that is?

Making decisions about what is dangerous and what is not can be difficult. Even ostensibly harmless access, like displaying a window can become a threat when paired with the ability for an untrusted application to communicate off of your host. The Java `SecurityManager` provides an option to flag windows created by an untrusted application with a special, recognizable border to prevent it from impersonating another application and perhaps tricking you into revealing your password or your secret recipe collection. There is also a grey area, in which an application can do devious things that aren't quite destructive. An applet that can mail a bug report can also mail-bomb your boss. The Java language provides the tools to implement whatever security policies you want. However, what these policies will be ultimately depends on who you are, what you are doing, and where you are doing it.

To fully exploit the power of Java, we need to have some basis on which to make reasonable decisions about the level of trust an application should have. Web browsers such as HotJava start by defining a few rules and some coarse levels of security that restrict where applets may come from and what system resources they may access. These rules are sufficient to keep the waving Duke applet from clutching your password file, but they aren't sufficient for applications you'd like to trust with sensitive information. What if you want to implement a secure applet to carry a credit card number to the mall, or more likely the credit-card company? How are people to trust that the applet they are using is really secure? If it's named the "Bank of Boofa" applet, how do they know it's legit?

You might think of trusting only certain hosts for these kinds of applications. However, as Java class files begin to fill the Net, the situation will become more complicated. Hosts can be impersonated. If your communications pass through an untrusted network, you can't be sure you're talking to the intended entity. Furthermore, class files may need to be cached or retrieved through complicated distribution mechanisms. For these kinds of applications, what we really need is a mechanism for verifying the authorship and authenticity of an item and making sure that it has not been tampered with by the time that you received it. Fortunately, this is a problem solved a while ago by your friendly neighborhood cryptographers.

Signing Classes

Digital signatures provide a means of authenticating documents. Like their inky analogs, they associate a name with an item in a way that is supposed to be difficult to forge. Unlike pen on paper, though, electronic digital signatures are actually difficult to forge when used properly. By their nature, digital signatures also provide the benefit that, if authenticated, a document is known not to have been altered in transit. In other words, you can't clip out a digital signature and attach it to a new document.

The details of cryptography are a bit beyond the scope of this book but the basics are important and interesting.[6] Digital signatures are one side of the coin of public-key cryptography. Public-key algorithms rely on the fundamental mathematical difficulty of factoring arbitrarily large numbers. In a public-key system, there are two pieces of information: a public key (as you might have guessed) and a private one. These keys have a special asymmetric relationship such that a message encrypted with one key can be decrypted only by knowing the other. This means that by giving you my public key, you can send me messages that only I can read. No one else, including you, has enough information to decrypt the encoded message, so it's safe to send it over untrusted networks. Now, by reversing this process, I can encrypt something with my private key so that anyone can use my public key to read the message. The important thing in this case is that the task of creating such a message without the private key is just as difficult as decoding the message in the first scenario. Since no one else knows my private key, only the real me could have sent the message. This is the basis for digital signatures. For Java, this means that we can tell a browser "I trust applets signed by John Doe"; if the browser succeeds in decoding an applet using John Doe's public key, it knows that the applet really came from John Doe, and therefore can be allowed greater privileges.

This process can be used to authenticate Java class files and other types of objects sent over the network. The author of a class signs the code with a digital signature, and we authenticate it when we retrieve it. Now we know that we have the authentic class, or do we? There is one problem that a digital signature alone doesn't solve: at some point we still have to assume we have the author's authentic public key. This is where a key-certification agency comes into play.

A key-certification agency validates a key by issuing a certificate that lists a name and an official public key. The certificate is signed with the agency's own digital signature. The agency presumably has a well-known public key to verify the certificate. Of course, this doesn't solve the problem entirely, but it reduces the number of people you have to trust and the amount of information you have to transport reliably. Presumably the agency is a reputable organization, its private keys are well guarded, and it certifies keys only after some kind of real-world validation such as person-to-person contact.

The most recent Java release (1.1) contains the tools you need to work with signed classes. You can sign Java classes; you can tell the HotJava browser whose classes you trust (and how much you trust them). Other browsers, like Netscape Navigator, should support signed classes in the future. You can also use the security API in your own Java programs to handle sensitive data safely. The important thing is, as always, to know who you are dealing with and what kind of software and security you have in place before sending any kind of confidential information over the Net. Don't become paranoid, just keep yourself informed so that you can weigh the risks and the benefits.

[6] See Bruce Schneier's encyclopedic *Applied Cryptography* (John Wiley & Sons).

PREVIOUS | HOME | NEXT
Safety of Implementation | BOOK INDEX | Java and the World Wide Web

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

1.7 Java and the World Wide Web

> Alice was beginning to get very tired of sitting by her sister on the bank, and of having nothing to do: once or twice she had peeped into the book her sister was reading, but it had no pictures or conversations in it, "and what is the use of a book," thought Alice, "without pictures or conversations?"
>
> --*Alice in Wonderland*

The application-level safety features of Java make it possible to develop new kinds of applications not necessarily feasible before now. A Web browser that implements the Java run-time system can incorporate Java applets as executable content inside of documents. This means that Web pages can contain not only static hypertext information but also full-fledged interactive applications. The added potential for use of the WWW is enormous. A user can retrieve and use software simply by navigating with a Web browser. Formerly static information can be paired with portable software for interpreting and using the information. Instead of just providing some data for a spreadsheet, for example, a Web document might contain a fully functional spreadsheet application embedded within it that allows users to view and manipulate the information.

Applets

The term *applet* is used to mean a small, subordinate, or embeddable application. By embeddable, I mean it's designed to be run and used within the context of a larger system. In that sense, most programs are embedded within a computer's operating system. An operating system manages its native applications in a variety of ways: it starts, stops, suspends, and synchronizes applications; it provides them with certain standard resources; and it protects them from one another by partitioning their environments.

In this book, I'll be describing Java applets, which are Java applications meant to be embedded in and controlled by a larger application, such as a Java-enabled Web browser or an applet viewer. To include an applet as executable content in a Web document, you use a special HTML tag. The `<applet>` tag points to an applet and provides configuration information about the applet.

As far as the Web browser model is concerned, an applet is just another type of object to display. Browsers make a distinction between items presented inline and items anchored via hypertext links and made available by external means, such as a viewer or helper application. If you download an MPEG video clip, for instance, and your browser doesn't natively understand MPEG, it will look for a helper application (an MPEG player) to pass the information to. Java-enabled Web browsers generally execute applets inline, in the context of a particular document, as shown in Figure 1.4. However, less capable browsers could initially provide some support for Java applets through an external viewer.

Figure 1.4: Applets in a Web document

[Graphic: Figure 1-4]

A Java applet is a compiled Java program, composed of classes just like any Java program. While a simple applet may consist of only a single class, most large applets should be broken into many classes. Each class is stored in a separate class file. The class files for an applet are retrieved from the network as they are needed. A large applet doesn't need to retrieve all its parts or all its data before beginning to interact with the user. Well-designed applets can take advantage of multithreading to wait for certain resources in the background, while performing other activities.

An applet has a four-part life cycle. When an applet is initially loaded by a Web browser, it's asked to initialize itself. The applet is then informed each time it's displayed and each time it's no longer visible to the user. Finally, the applet is told when it's no longer needed, so that it can clean up after itself. During its lifetime, an applet may start and suspend itself, do work, communicate with other applications, and interact with the Web browser.

Applets are autonomous programs, but they are confined within the walls of a Web browser or applet viewer, and have to play by its rules. I'll be discussing the details of what applets can and can't do as we explore features of the Java language. However, under the most conservative security policies, an applet can interact only with the user and can only communicate over the network with the host from which it originated. Other types of activities, like accessing files or interacting directly with outside applications, are typically prevented by the security manager that is part of the Web browser or applet viewer. But aside from these restrictions, there is no fundamental difference between a Java applet and a standalone Java application.

New Kinds of Applications

Sun's HotJava Web browser is written entirely in Java. Because it's a Java application, HotJava is immediately available on any platform with the Java run-time system. This goes a long way towards the goal of a Web browser serving as a universal access point for resources on the Net. And where one Web browser leads the way, more will surely follow.

In addition to displaying Java applets as executable content in Web pages, the HotJava application dynamically extends itself by loading Java code from the Net. HotJava uses *protocol handlers* and *content handlers* to provide this functionality.[7] Protocol handlers and content handlers are classes in the Java API that let an application implement new types of URLs and interpret the objects retrieved from them. A Web browser that supports this functionality can load handlers from a remote location and effectively upgrade itself on the fly to use new protocols and access new kinds of information.

[7] Downloadable content and protocol handlers are not supported in HotJava 1.0, but will be supported in a future release.

Like applets, content handlers and protocol handlers can be served by a Web site, along with the information they interpret. As an example, consider the new Portable Network Graphics (PNG) format, a freely distributable alternative to GIF. By supplying a PNG content handler along with PNG images on our server, we give users the ability to use the new image format, just as they would a built-in format. We don't have to create a new standard and force every Web browser to support the new format. Instead, the first time a user loads a document referencing a PNG image from our site, the Web browser will realize it doesn't understand the object and will ask the server if it has a content handler for it. Since we've provided a content handler, the browser can load it and then use it to interpret and display the image dynamically.

In a similar manner, protocol handlers allow a Web browser to start speaking a new protocol with the server. This is especially useful for things like security protocols. If we invent a revolutionary new cryptographic protocol late one night, all we have to do is implement it in the form of a protocol handler and place it on our server. We can then start using URLs that point through our new protocol at objects on our server, and people can immediately begin using it.

These scenarios describe just a few things that safe, transportable code will allow. We will undoubtedly see many other new breeds of application we can't even begin to anticipate.

New Kinds of Media

When it was first released, Java quickly achieved a reputation for multimedia capabilities. Frankly, this wasn't really deserved. At that point, Java provided facilities for doing simple animations and playing audio. You could even animate and play audio simultaneously, though you couldn't synchronize the two. Still, this was a significant advance for the Web, and people thought it was pretty impressive.

JavaSoft is now working on real media features, which go far beyond anything that has yet been seen on the Web. This includes CD quality sound, 3D animation, media players that synchronize audio and video, speech synthesis and recognition, and more. These new features aren't yet in Java 1.1, but will make their way into Java over the next eighteen months. In short, if you were impressed by Java 1.0, you haven't seen anything yet. Java's multimedia capabilities will be taking shape over the next two years.

New Software Development Models

For some time now, people have been using visual development environments to develop user interfaces. These environments let you generate applications by moving components around on the screen, connecting components to each other, and so on. In short, designing a user interface is a lot more like drawing a picture than like writing code.

For visual development environments to work well, you need to be able to create reusable software components. That's what Java Beans are all about. The JavaBeans architecture defines a way to package software as reusable building blocks. A graphical development tool can figure out a component's capabilities, customize the component, and connect it to other components to build applications. JavaBeans takes the idea of graphical development a step further. Beans aren't limited to visible, user interface components: you can have Beans that are entirely invisible, and whose job is purely computational. For example, you could have a Bean that did database access; you could connect this to a Bean that let the user request information from the database; and you could use another Bean to display the result. Or you could have a set of Beans that implemented the functions in a mathematical library; you could then do numerical analysis by connecting different functions to each other. In either case, you could "write" programs without writing a single line of code. Granted, someone would have to write the Beans in the first place; but that's a much smaller task, and we expect markets to develop for "off the shelf" Bean collections.

Before it can use a Bean, an application builder must find out the Bean's capabilities. There are a few ways it can do this; the simplest is called "reflection". To write a Bean that uses reflection, all you need to do is follow some simple design patterns. Let's say that you're writing a Bean that is capable of changing its color. Then you would write two methods that report the current color and change its value:

```
...
private Color c;
public Color getMyColor() { return c; }
public void setMyColor( Color c) { this.c = c; }
```

These methods follow some well defined conventions (design patterns) that let the graphical interface builder (or any other tool that wants to do the work) analyze the Bean, and discover that it has a property called `MyColor`, that the value of this property is a `Color` object, and that the methods `getMyColor()` and `setMyColor()` should be used to change its value.

If they need to, Beans can provide additional information using a process called "introspection". But even without introspection, you can see that a graphical development tool can easily analyze a Bean, figure out what it can do, and let a user change the Bean's properties without writing any code.

Of course, once a development tool has customized a Bean and connected it to other Beans, it needs a way to save the result. A process called "serialization" lets a tool save the Bean's current state, along with any extra code it has written to stitch Beans together in an application.

Visual development tools that support Java Beans include Borland's JBuilder (http://www.borland.com), Symantec's Visual Cafe (http://www.symantec.com), and SunSoft's Java Workshop. By using a "bridge", Java Beans will be able to interact with other component environments, including ActiveX, OpenDoc, and LiveConnect. A beta version of the ActiveX bridge is currently available.

1.8 Java as a General Application Language

The Java applet API is a framework that allows Java-enabled Web browsers to manage and display embedded Java applications within WWW documents. However, Java is more than just a tool for building transportable multimedia applications. Java is a powerful, general-purpose programming language that just happens to be safe and architecture independent. Standalone Java applications are not subject to the restrictions placed on applets; they can do all activities that software written in a language like C does.

Any software that implements the Java run-time system can run Java applications. Applications written in Java can be large or small, standalone or component-like, as in other languages. Java applets are different from other Java applications only in that they expect to be managed by a larger application. In this book, we will build examples of both applets and standalone Java applications. With the exception of the few things applets can't do, such as access files, all of the tools we examine in this book apply to both applets and standalone Java applications.

1.9 A Java Road Map

With everything that's going on, it's hard to keep track of what's available now, what's promised, and what has been around for some time. Here's a road map that tries to impose some order on Java's past, present, and future.

The Past: Java 1.0

Java 1.0 provided the basic framework for Java development: the language itself, plus packages that let you write applets and simple applications. Java 1.0 is officially obsolete, though it will be some time before vendors catch up with the new release.

The Present: Java 1.1

Java 1.1 is the current version of Java. It incorporates major improvements in the AWT package (Java's windowing facility). It also adds many completely new features, including:

JDBC

> A general facility for interacting with databases.

RMI

> Remote Method Invocation: a facility that lets you call methods that are provided by a server running somewhere else on the network.

JavaBeans

> Java's component architecture, which we discussed earlier.

Security

A facility for cryptography; this is the basis for signed classes, which we discussed earlier.

Internationalization

The ability to write programs that adapt themselves to the language the user wants to use; the program automatically displays text in the appropriate language.

Java 1.1 incorporates many other improvements and new features, but these are the most important. As of May, 1997, most Web browsers haven't yet incorporated Java 1.1, but they will as soon as possible. In this book, we'll try to give you a taste of as many features as possible; unfortunately for us, the Java environment has become so rich that it's impossible to cover everything in a single book.

The Future

We've mentioned a few of the things that are in the pipeline, including high quality audio, advanced 3D rendering, and speech synthesis. Other things to look forward to are class libraries for advanced 2D graphics (Java 2D), electronic commerce (JECF), managing network devices (Java Management), naming and directory services (JNDI), telephony (JTAPI), and writing network servers (Java Server). Beta versions of some of these facilities are available now.

We're also starting to see new kinds of computing devices that incorporate Java. Network computers that are based on Java and use the HotJava browser as their user interface are already available, as are "smart cards": credit card-like devices with a Java processor built in. You can expect to see Java incorporated into PDAs, telephones, and many other devices.

PREVIOUS | HOME | NEXT
Java as a General Application Language | BOOK INDEX | Availability

1.10 Availability

By the time you read this book, you should have several choices for Java development environments and run-time systems. As this book goes to press, Sun's Java Development Kit (JDK) 1.1 is available for Solaris, Windows NT, and Windows 95. The JDK provides an interpreter and a compiler for building general-purpose Java applications. A beta version of JDK 1.1 for the Macintosh will be available later in 1997. Visit Sun's Java Web site, http://www.javasoft.com/ for more information about the JDK. There are also a number of JDK ports for various platforms. Some of the most significant platforms are Novell, HP-UX, OSF/1 (including Digital UNIX), Silicon Graphics' IRIX, and Linux. For more information, see the Web pages maintained by the vendor you're interested in. JavaSoft maintains a Web page summarizing porting efforts at http://www.javasoft.com/products/jdk/jdk-ports.html. Another good source for current information is the Java FAQ from the comp.lang.java newsgroup.

There are efforts under way to produce a free clone of Java, redistributable in source form. The Java Open Language Toolkit (JOLT) Project is working to assemble a high-quality Java implementation that will pass Sun's validation tests and earn a Java stamp. The JOLT Project Web page is accessible from http://www.redhat.com/.

The Netscape Navigator Web browser comes with its own implementation of the Java run-time system that runs Java applets. Netscape also provides a `-java` switch that lets you execute Java applications (including the Java compiler) and applets and run nongraphical applications. Netscape's Web site is located at http://home.netscape.com/. Check there for information on the latest version of Netscape Navigator.

2.2 Hello Web! II: The Sequel

Let's make our applet a little more interactive, shall we? The following improvement, `HelloWeb2`, allows us to drag the message around with the mouse. `HelloWeb2` is also customizable. It takes the text of its message from a parameter of the `<applet>` tag of the HTML document.

`HelloWeb2` is a new applet--another subclass of the `Applet` class. In that sense, it's a sibling of `HelloWeb`. Having just seen inheritance at work, you might wonder why we aren't creating a subclass of `HelloWeb` and exploiting inheritance to build upon our previous example and extend its functionality. Well, in this case, that would not necessarily be an advantage, and for clarity we simply start over.[2] Here is `HelloWeb2`:

[2] You are left to consider whether such a subclassing would even make sense. Should `HelloWeb2` really be a kind of `HelloWeb`? Are we looking for refinement or just code reuse?

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class HelloWeb2 extends Applet implements MouseMotionListener {
    int messageX = 125, messageY = 95;
    String theMessage;

    public void init() {
        theMessage = getParameter("message");
        addMouseMotionListener(this);
    }

    public void paint( Graphics graphics ) {
        graphics.drawString( theMessage, messageX, messageY );
    }
    public void mouseDragged( MouseEvent e ) {
        messageX = e.getX();
        messageY = e.getY();
        repaint();
    }
    public void mouseMoved( MouseEvent e ) { }
}
```

Place the text of this example in a file called *HelloWeb2.java* and compile it as before. You should get a new class file, *HelloWeb2.class*, as a result. We need to create a new `<applet>` tag for `HelloWeb2`. You can either create another HTML document (copy *HelloWeb.html* and modify it) or simply add a second `<applet>` tag to the existing *HelloWeb.html* file. The `<applet>` tag for `HelloWeb2` has to use a parameter; it should look like:

```
<applet code=HelloWeb2 width=300 height=200>
<param name="message" value="Hello Web!" >
</applet>
```

Feel free to substitute your own salacious comment for the value of `message`.

Run this applet in your Java-enabled Web browser, and enjoy many hours of fun, dragging the text around with your mouse.

Import

So, what have we added? First you may notice that a few lines are now hovering above our class:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class HelloWeb2 extends Applet implements MouseMotionListener {
...
```

The `import` statement lists external classes to use in this file and tells the compiler where to look for them. In our first `HellowWeb` example, we designated the `Applet` class as the superclass of `HelloWeb`. `Applet` was not defined by us and the compiler therefore had to look elsewhere for it. In that case, we referred to `Applet` by its fully qualified name, `java.applet.Applet`, which told the compiler that `Applet` belongs to the `java.applet` package so it knew where to find it.

The `import` statement is really just a convenience; by importing `java.applet.Applet` in our newer example, we tell the compiler up front we are using this class and, thereafter in this file, we can simply refer to it as `Applet`. The second import statement makes use of the wildcard "*" to tell the compiler to import all of the classes in the `java.awt` package. But don't worry, the compiled code doesn't contain any excess baggage. Java doesn't do things like that. In fact, compiled Java classes don't contain other classes at all; they simply note their relationships. Our current example uses only the `java.awt.Graphics` class. However, we are anticipating using several more classes from this package in the upcoming examples. We also import all the classes from the package `java.awt.event`; these classes provide the `Event` objects that we use to communicate with the user. By listening for events, we find out when the user moved the mouse, clicked a button, and so on. Notice that importing `java.awt.*` doesn't automatically import the event package. The star only imports the classes in a particular package, not other packages.

The `import` statement may seem a bit like the C or C++ preprocessor `#include` statement, which injects header files into programs at the appropriate places. This is not true; there are no header files in Java. Unlike compiled C or C++ libraries, Java binary class files contain all necessary type information about the classes, methods, and variables they contain, obviating the need for prototyping.

Instance Variables

We have added some variables to our example:

```
public class HelloWeb2 extends Applet {
    int messageX = 125, messageY = 95;
    String theMessage;
...
```

`messageX` and `messageY` are integers that hold the current coordinates of our movable message. They are initialized to default values, which should place a message of our length somewhere near the center of the applet. Java integers are always 32-bit signed numbers. There is no fretting about what architecture your code is running on; numeric types in Java are precisely defined. The variable `theMessage` is of type `String` and can hold instances of the `String` class.

You should note that these three variables are declared inside the braces of the class definition, but not inside any particular method in that class. These variables are called *instance variables* because they belong to the entire class, and copies of them appear in each separate instance of the class. Instance variables are always visible (usable) in any of the methods inside their class. Depending on their modifiers, they may also be accessible from outside the class.

Unless otherwise initialized, instance variables are set to a default value of `0` (zero), `false`, or `null`. Numeric types are set to zero, `boolean` variables are set to `false`, and class type variables always have their value set to `null`, which means "no value." Attempting to use an object with a `null` value results in a run-time error.

Instance variables differ from method arguments and other variables that are declared inside of a single method. The latter are called *local variables*. They are effectively private variables that can be seen only by code inside the method. Java doesn't initialize local variables, so you must assign values yourself. If you try to use a local variable that has not yet been assigned a value, your code will generate a compile-time error. Local variables live only as long as the method is executing and then disappear (which is fine since nothing outside of the method can see them anyway). Each time the method is invoked, its local variables are recreated and must be assigned values.

Methods

We have made some changes to our previously stodgy `paint()` method. All of the arguments in the call to `drawString()` are now variables.

Several new methods have appeared in our class. Like `paint()`, these are methods of the base `Applet` class we override to add our own functionality. `init()` is an important method of the `Applet` class. It's called once, when our applet is created, to give us an opportunity to do any work needed to set up shop. `init()` is a good place to allocate resources and perform other activities that need happen only once in the lifetime of the applet. A Java-enabled Web browser calls `init()` when it prepares to place the `Applet` on a page.

Our overridden `init()` method does two things; it sets the text of the `theMessage` instance variable, and it tells the system "Hey, I'm interested in anything that happens involving the mouse":

```
public void init() {
    theMessage = getParameter("message");
    addMouseMotionListener(this);
}
```

When an applet is instantiated, the parameters given in the `<applet>` tag of the HTML document are stored in a table and made available through the `getParameter()` method. Given the name of a parameter, this method returns the value as a `String` object. If the name is not found, it returns a `null` value.

So what, you may ask, is the type of the argument to the `getParameter()` method? It, too, is a `String`. With a little magic from the Java compiler, quoted strings in Java source code are turned into `String` objects. A bit of funny-business is going on here, but it's simply for convenience. (See Chapter 7, *Basic Utility Classes* for a complete discussion of the `String` class.)

`getParameter()` is a public method we inherited from the `Applet` class. We can use it from any of our methods. Note that the `getParameter()` method is invoked directly by name; there is no object name prepended to it with a dot. If a method exists in our class, or is inherited from a superclass, we can call it directly by name.

In addition, we can use a special read-only variable, called `this`, to explicitly refer to our object. A method can use `this` to refer to the instance of the object that holds it. The following two statements are therefore equivalent:

```
theMessage = getParameter("message");
```

or

```
theMessage = this.getParameter("message");
```

I'll always use the shorter form. We will need the `this` variable later when we have to pass a reference to our object to a method in another class. We often do this so that methods in another class can give us a call back later or can watch our public variables.

The other method that we call in `init()` is `addMouseMotionListener()`. This method is part of the event mechanism, which we discuss next.

Events

The last two methods of `HelloWeb2` let us get information from the mouse. Each time the user performs an action, such as hitting a key on the keyboard, moving the mouse, or perhaps banging his or her head against a touch-sensitive screen, Java generates an *event*. An event represents an action that has occurred; it contains information about the action, such as its time and location. Most events are associated with a particular graphical user interface (GUI) component in an application. A keystroke, for instance, could correspond to a character being typed into a particular text entry field. Pressing a mouse button could cause a certain graphical button on the screen to activate. Even just moving the mouse within a certain area of the screen could be intended to trigger effects such as highlighting or changing the cursor to a special mouse cursor.

The way events work is one of the biggest changes between Java 1.0 and Java 1.1. We're going to talk about the Java 1.1 events only; they're a big improvement, and there's no sense in learning yesterday's news. In Java 1.1, there are many different event classes: `MouseEvent`, `KeyEvent`, `ActionEvent`, and many others. For the most part, the meaning of these events is fairly intuitive. A `MouseEvent` occurs when the user does something with the mouse, a `KeyEvent` occurs when the user types a key, and so on. `ActionEvent` is a little special; we'll see it at work in our third applet. For now, we'll focus on dealing with a `MouseEvent`.

The various GUI components in Java generate events. For example, if you click the mouse inside an applet, the applet generates a mouse event. (In the future, we will probably see events as a general purpose way to communicate between Java objects; for the moment, let's limit ourselves to the simplest case.) In Java 1.1, any object can ask to receive the events generated by some other component. We call the object that wants to receive events a "listener." To declare that it wants to receive some component's mouse motion events, you call that component's `addMouseMotionListener()` method. That's what our applet is doing in `init()`. In this case, the applet is calling its own `addMouseMotionListener()` method, with the argument `this`, meaning "I want to receive my own mouse motion events." (For the time being, we're ignoring a minor distinction between "mouse events" and "mouse motion events." Suffice it to say that in this example, we only care about mouse motions.)

That's how we register to receive events. But how do we actually get them? That's what the two remaining methods in our applet are for. The `mouseDragged()` method is called automatically whenever the user drags the mouse--that is, moves the mouse with any button pressed. The `mouseMoved()` method is called whenever the user moves the mouse without pressing a button. Our `mouseMoved()` method is simple: it doesn't do anything. We're ignoring simple mouse motions.

`mouseDragged()` has a bit more meat to it. It is called repeatedly to give us updates on the position of the mouse. Here it is:

```
    public void mouseDragged( MouseEvent e ) {
        messageX = e.getX();
        messageY = e.getY();
        repaint();
    }
```

The first argument to `mouseDragged()` is a `MouseEvent` object, `e`, that contains all the information we need to know about this event. We ask the `MouseEvent` to tell us the `x` and `y` coordinates of the mouse's current position by calling its `getX()` and `getY()` methods. These are saved in the `messageX` and `messageY` instance variables. Now, having changed the coordinates for the message, we would like `HelloWeb2` to redraw itself. We do this by calling `repaint()`, which asks the system to redraw the screen at a later time. We can't call `paint()` directly because we don't have a graphics context to pass to it.

The real beauty of this event model is that you only have to handle the kinds of events you want. If you don't care about keyboard events, you just don't register a listener for them; the user can type all he or she wants, and you won't be bothered. Java doesn't go around asking potential recipients whether they might be interested in some event, as it did in older versions. If there are no listeners for a particular kind of event, Java won't even generate it. The result is that event handling in Java 1.1 is quite efficient.

I've danced around one question that should be bothering you by now: how does the system know to call `mouseDragged()` and `mouseMoved()`? And why do we have to supply a `mouseMoved()` method that doesn't do anything? The answer to these questions has to do with "interfaces." We'll discuss interfaces after clearing up some unfinished business with `repaint()`.

The repaint() Method

We can use the `repaint()` method of the `Applet` class to request our applet be redrawn. `repaint()` causes the Java windowing system to schedule a call to our `paint()` method at the next possible time; Java supplies the necessary `Graphics` object, as shown in Figure 2.5.

Figure 2.5: Invoking the repaint() method

[Graphic: Figure 2-5]

This mode of operation isn't just an inconvenience brought about by not having the right graphics context handy at the moment. The foremost advantage to this mode of operation is that the repainting is handled by someone else, while we are free to go about our business. The Java system has a separate, dedicated thread of execution that handles all `repaint()` requests. It can schedule and consolidate `repaint()` requests as necessary, which helps to prevent the windowing system from being overwhelmed during painting-intensive situations like scrolling. Another advantage is that all of the painting functionality can be kept in our `paint()` method; we aren't tempted to spread it throughout the application.

Interfaces

Now it's time to face up to the question I avoided earlier: how does the system know to call `mouseDragged()` when a mouse event occurs? Is it simply a matter of knowing that `mouseDragged()` is some magic name that our event handling method must have? No; the answer to the question lies in the discussion of interfaces, which are one of the most important features of the Java language.

The first sign of an interface comes on the line of code that introduces the `HelloWeb2` applet: we say that the applet implements `MouseMotionListener`. `MouseMotionListener` is an interface that the applet implements. Essentially, it's a list of methods that the applet must have; this particular interface requires our applet to have methods called `mouseDragged()` and `mouseMoved()`. The interface doesn't say what these methods have to do--and indeed, `mouseMoved()` doesn't do anything. It does say that the methods must take a `MouseEvent` as an argument and return `void` (i.e., no return value).

Another way of looking at an interface is as a contract between you, the code developer, and the compiler. By saying that your applet implements the `MouseMotionListener` interface, you're saying that these methods will be available for other parts of the system to call. If you don't provide them, the compiler will notice and give you an error message.

But that's not the only impact interfaces have on this program. An interface also acts like a class. For example, a method could return a `MouseMotionListener`, or take a `MouseMotionListener` as an argument. This means that you don't care about the object's class; the only requirement is that the object implement the given interface. In fact, that's exactly what the method `addMouseMotionListener()` does. If you look up the documentation for this method, you'll find that it takes a `MouseMotionListener` as an argument. The argument we pass is `this`, the applet itself. The fact that it's an applet is irrelevant, it could be a `Cookie`, an `Aardvark`, or any other class we dream up. What is important is that it implements `MouseMotionListener`, and thus declares that it will have the two named methods. That's why we need a `mouseMoved()` method, even though the one we supplied doesn't do anything: the `MouseMotionListener` interface says we have to have one.

In other languages, you'd handle this problem by passing a function pointer; for example, in C, the argument to `addMouseMotionListener()` might be a pointer to the function you want to have called when an event occurs. This technique is called a "callback." For security reasons, the Java language has eliminated function pointers. Instead, we use interfaces to make contracts between classes and the compiler. (Some new features of the language make it easier to do something similar to a callback, but that's beyond the present discussion.)

The Java distribution comes with many interfaces that define what classes have to do in various situations. Furthermore, in Chapter 5, *Objects in Java*, you'll see how to define your own interfaces. It turns out that this idea of a contract between the compiler and a class is very important. There are many situations like the one we just saw, where you don't care what class something is, you just care that it has some capability, like listening for mouse events. Interfaces give you a way of acting on objects based on their capabilities, without knowing or caring about their actual type.

Furthermore, interfaces provide an important escape clause to the rule that any new class can only extend a single class (formally called "single inheritance"). They provide most of the advantages of multiple inheritance (a feature of languages like C++) without the confusion. A class in Java can only extend one class, but can implement as many interfaces as it wants; our next applet will implement two interfaces, and the final example in this chapter will implement three. In many ways, interfaces are almost like classes, but not quite. They can be used as data types, they can even extend other interfaces (but not classes), and can be inherited by classes (if class A implements interface B, subclasses of A also implement B). The crucial difference is that applets don't actually inherit methods from interfaces; the interfaces only specify the methods the applet must have.

2.3 Hello Web! III: The Button Strikes!

Well, now that we have those concepts under control, we can move on to some fun stuff. `HelloWeb3` brings us a new graphical interface component: the `Button`. We add a `Button` component to our applet that changes the color of our text each time the button is pressed. Our new example is shown below.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class HelloWeb3 extends Applet
    implements MouseMotionListener, ActionListener {
    int messageX = 125, messageY = 95;
    String theMessage;
    Button theButton;
    int colorIndex = 0;
    static Color[] someColors = {
        Color.black, Color.red, Color.green, Color.blue, Color.magenta };

    public void init() {
        theMessage = getParameter("message");
        theButton = new Button("Change Color");
        add(theButton);
        addMouseMotionListener(this);
        theButton.addActionListener(this);
    }

    public void paint( Graphics gc ) {
        gc.drawString( theMessage, messageX, messageY );
    }
    public void mouseDragged( MouseEvent e ) {
        messageX = e.getX();
        messageY = e.getY();
        repaint();
    }
    public void mouseMoved( MouseEvent e ) { }
    public void actionPerformed( ActionEvent e ) {
        if ( e.getSource() == theButton ) {
            changeColor();
        }
    }
    synchronized private void changeColor() {
        if ( ++colorIndex == someColors.length )
            colorIndex = 0;
        setForeground( currentColor() );
        repaint();
    }
    synchronized private Color currentColor() {
        return someColors[ colorIndex ];
    }
}
```

Create `HelloWeb3` just as the other applets and put an `<applet>` tag referencing it in an HTML document. An `<applet>` tag just like the one for `HelloWeb2` will do nicely. Run the example, and you should see the display shown in Figure 2.6. Drag the text. Each time you press the button the color should change. Call your friends! They should be duly impressed.

Figure 2.6: Hello Web! III

[Graphic: Figure 2-6]

The New Operator

So what have we added this time? Well, for starters we have a new variable:

```
Button theButton;
```

The `theButton` variable is of type `Button` and is going to hold an instance of the `java.awt.Button` class. The `Button` class, as you might expect, represents a graphical button, which should look like other buttons in your windowing system.

Two additional lines in `init()` actually create the button and cause it to be displayed in our applet:

```
theButton = new Button("Change Color");
add(theButton);
```

The first line brings us to something new. The `new` keyword is used to create an instance of a class. Recall that the variable we have declared is just an empty reference and doesn't yet point to a real object--in this case, an instance of the `Button` class. This is a fundamental and important concept. We have dealt with objects previously in our examples. We have assigned them to variables, and we have called methods in them. So far, however, these objects have always been handed to us ready to go, and we have not had to explicitly create them ourselves. In the case of our `paint()` method, we were given a `Graphics` object with which to draw. The system created this instance of the `Graphics` class for our area of the screen and passed it to us in the parameter variable `gc`. Our `theMessage` variable is of type `String`, and we used it to hold a `String` that was returned by the `getParameter()` method. In each case, some other method instantiated (created a new instance of) the class for us.

The closest we came to actually instantiating an object was when we passed the name of the HTML `<applet>` parameter as an argument to the `getParameter()` method. In that case, we created a `String` object and passed it as the argument, but we did it in a rather sneaky fashion. As I mentioned previously, `String` objects have special status in the Java language. Because strings are used so frequently, the Java compiler creates an instance of the `String` class for us whenever it comes across quoted text in our source code. `String` objects are, in all other respects, normal objects. (See Chapter 7, *Basic Utility Classes*.)

The `new` operator provides the general mechanism for instantiating objects. It's the feature of the Java language that creates a new instance of a specified class. It arranges for Java to allocate storage for the object and then calls the constructor method of the objects' class to initialize it.

Constructors

A *constructor* is a special method that is called to set up a new instance of a class. When a new object is created, Java allocates storage for it, sets variables to their default values, and then calls the constructor method for the class to do whatever application-level setup is required.

A constructor method looks like a method with the same name as its class. For example, the constructor for the `Button` class is called `Button()`. Constructors don't have a return type; by definition they return an object of that class. But like other methods, constructors can take arguments. Their sole mission in life is to configure and initialize newly born class instances, possibly using whatever information is passed to them in parameters.

It's important to understand the difference between a constructor and a method like our `init()` method. Constructors are special methods of a class that help set up and initialize an instance of a class when it's first created. The `init()` method of the `Applet` class serves a very similar purpose; however, it's quite different. Constructors are a feature of the Java language. Every class, including `Applet`, has constructors. `init()`, however, is just a method of the `Applet` class like any other. It's an application-level phenomenon that happens to perform initialization.

An object is created by using the `new` operator with the constructor for the class and any necessary arguments. The resulting object instance is returned as a value. In our example, we create a new instance of `Button` and assign it to our `theButton` variable:

```
theButton = new Button("Change Color");
```

This `Button` constructor takes a `String` as an argument and, as it turns out, uses it to set the label of the button on the screen. A class could also, of course, provide methods that allow us to configure an object manually after it's created or to change its configuration at a later time. Many classes do both; the constructor simply takes its arguments and passes them to the appropriate methods. The `Button` class, for example, has a public method, `setLabel()`, that allows us to set a `Button`'s label at any time. Constructors with parameters are therefore a convenience that allows a sort of short hand to set up a new object.

Method Overloading

I said this `Button` constructor because there could be more than one. A class can have multiple constructors, each taking different parameters and possibly using them to do different kinds of setup. When there are multiple constructors for a class, Java chooses the correct one based on the types of arguments that are passed to it. We call the `Button` constructor and pass it a `String` argument, so Java locates the constructor method of the `Button` class that takes a single `String` argument and uses it to set up the object. This is called *method overloading*. All methods in Java, not just constructors, can be overloaded; this is one aspect of the object-oriented programming principle of polymorphism.

A constructor method that takes no arguments is called the *default constructor*. As you'll see in Chapter 7, *Basic Utility Classes*, default constructors play a special role in the initialization of inherited class members.

Garbage Collection

I've told you how to create a new object with the `new` operator, but I haven't said anything about how to get rid of an object when you are done with it. If you are a C programmer, you're probably wondering why not. The reason is that you don't have to do anything to get rid of objects when you are done with them.

The Java run-time system uses a *garbage collection* mechanism to deal with objects no longer in use. The garbage collector sweeps up objects not referenced by any variables and removes them from memory. Garbage collection is one of the most important features of Java. It frees you from the error-prone task of having to worry about details of memory allocation and deallocation.

Components

I have used the terms *component* and *container* somewhat loosely to describe graphical elements of Java applications. However, you may recall from Figure 2.3 that these terms are the names of actual classes in the `java.awt` package.

`Component` is a base class from which all of Java's GUI components are derived. It contains variables that represent the location, shape, general appearance, and status of the object, as well as methods for basic painting and event handling. The familiar `paint()` method we have been using in our example is actually inherited from the `Component` class. `Applet` is, of course, a kind of `Component` and inherits all of its public members, just as other (perhaps simpler) types of GUI components do.

The `Button` class is also derived from `Component` and therefore shares this functionality. This means that the developer of the `Button` class had methods like `paint()` available with which to implement the behavior of the `Button` object, just as we did when creating our applet. What's exciting is that we are perfectly free to further subclass components like `Button` and override their behavior to create our own special types of user-interface components.

Both `Button` and `Applet` are, in this respect, equivalent types of things. However, the `Applet` class is further derived from a class called `Container`, which gives it the added ability to hold other components and manage them.

Containers

A `Button` object is a simple GUI component. It makes sense only in the context of some larger application. The `Container` class is an extended type of `Component` that maintains a list of child components and helps to group them. The `Container` causes its children to be displayed and arranges them on the screen according to a particular scheme. A `Container` may also receive events related to its child components. As I mentioned earlier, if a component doesn't respond to a particular type of event by overriding the appropriate event-handling method and handling the event, the event is passed to the parent `Container` of the component. This is the default behavior for the standard Java AWT components, which gives us a great deal of flexibility in managing interface components. We could, for example, create a smart button by subclassing the `Button` class and overriding certain methods to deal with the action of being pressed. Alternatively, we could simply have the `Button`'s container note which `Button` is pressed and handle the event appropriately. In the interest of keeping our examples contained in a single class, I am using the gestalt view and letting our `Button`'s container, `HelloWeb3`, deal with its events.

Remember that a `Container` is a `Component` too and, as such, can be placed alongside other `Component` objects in other `Containers`, in a hierarchical fashion, as shown in Figure 2.7. Our `HelloWeb3` applet is a kind of `Container` and can therefore hold and manage other Java AWT components and containers like buttons, sliders, text fields, and panels.

Figure 2.7: A hypothetical layout of Java containers and components

[Graphic: Figure 2-7]

In Figure 2.7, the italicized items are components, and the bold items are containers. The keypad is implemented as a container object that manages a number of keys. The keypad itself is contained in the `GizmoTool` container object.

Layout

After creating the `Button` object, we'd like to stick it in our applet. To do so, we invoke the `add()` method of `Applet`, passing the `Button` object as a parameter:

```
add(theButton);
```

`add()` is a method inherited by our `Applet` from the `Container` class. It appends our `Button` to the list of components `HelloWeb3` manages. Thereafter, `HelloWeb3` is responsible for the `Button`: the applet causes the button to be displayed, it determines where in our part of the screen the button should be placed, and it receives events when the button is pushed.

Java uses an object called a `LayoutManager` to determine the precise location in `HelloWeb3`'s screen area the `Button` is displayed. A `LayoutManager` object embodies a particular scheme for arranging components on the screen and adjusting their sizes. You'll learn more about layout managers in Chapter 12, *Layout Managers*. There are several standard layout managers to choose from, and we can, of course, create new ones. In our case, we have not specified a layout manager, so we get the default one, which means our button appears centered at the top of the applet.

Subclassing and Subtypes

If you look up the add() method of the Container class, you'll see that it takes a Component object as an argument. But in our example we've given it a Button object. What's going on?

Well, if you check the inheritance diagram in Figure 2.3 again, you'll see that Button is a subclass of the Component class. Because a subclass is a kind of its superclass and has, at minimum, the same public methods and variables, we can use an instance of a subclass anywhere we use an instance of its superclass. This is a very important concept, and it's a second aspect of the object-oriented principle of polymorphism. Button is a kind of Component, and any method that expects a Component as an argument will accept a Button.

More Events and Interfaces

Now that we have a Button, we need some way to communicate with it: that is, to get the events it generates. We could just listen for mouse clicks, figure out whether they were close enough to the button, and act accordingly. But that would take a lot of work, and would give up the advantages of using a pre-built component. Buttons generate a special kind of event called an ActionEvent when someone clicks on them. To receive these events, we have added another method to our class:

```
public void actionPerformed( ActionEvent e ) {
    if ( e.getSource() == theButton ) {
        changeColor();
    }
}
```

If you understood the previous applet, you shouldn't be surprised to see that HelloWeb3 now declares that it implements the ActionListener interface, in addition to MouseMotionListener. ActionListener requires us to implement an actionPerformed() method, which is called whenever an ActionEvent occurs. You also shouldn't be surprised to see that we added a line to init(), registering the applet as a listener for the button's action events: this is the call to theButton.addActionListener(this).

The actionPerformed() method takes care of any action events that arise. First, it checks to make sure that the event's source (the component generating the event) is what we think it should be: theButton, the only button we've put in the applet. This may seem superfluous; after all, what else could possibly generate an action event? In this applet, nothing. But it's a good idea to check, because another applet may have several buttons, and you may need to figure out which one is meant. Or you may add a second button to this applet later, and you don't want the applet to break something. To make this check, we call the getSource() method of the ActionEvent object, e. Then we use the "==" operator to make sure that the event source matches theButton. Remember that in Java, == is a test for identity, not equality; it is true if the event source and theButton are the same object. The distinction between equality and identity is important. We would consider two String objects to be equal if they have the same characters in the same sequence. However, they might not be the same object. In Chapter 5, *Objects in Java*, we'll look at the equals() method, which tests for equivalence. Once we establish that the event e comes from the right button, we call our changeColor() method, and we're done.

You may be wondering why we don't have to change mouseDragged() now that we have a Button in our applet. The rationale is that the coordinates of the event are all that matter for this method. We are not particularly concerned if the event happens to fall within an area of the screen occupied by another component. This means that you can drag the text right through the Button and even lose it behind the Button if you aren't careful: try it and see!

Color Commentary

To support HelloWeb3's colorful side we have added a couple of new variables and two helpful methods. We create and initialize an array of Color objects representing the colors through which we cycle when the button is pressed. We also declare an integer variable that serves as an index to this array, specifying the current color:

```
Color[] someColors = {
    Color.black, Color.red, Color.green, Color.blue, Color.magenta };
int colorIndex;
```

A number of things are going on here. First let's look at the Color objects we are putting into the array. Instances of the java.awt.Color class represent colors and are used by all classes in the java.awt package that deal with color graphics. Notice that we are referencing variables such as Color.black and Color.red. These look like normal references to an object's instance variables; however Color is not an object, it's a class. What is the meaning of this?

Static Members

If you recall our discussion of classes and instances, I hinted then that a class can contain methods and variables that are shared among all instances of the class. These shared members are called *static variables* and *static methods*. The most common use of static variables in a class is to hold predefined constants or unchanging objects all of the instances can use.

There are two advantages to this approach. The more obvious advantage is that static members take up space only in the class; the members are not replicated in each instance. The second advantage is that static members can be accessed even if no instances of the class exist. A hypothetical Component class might have a static variable called maximumWidth. Some other class that has to deal with this component, such as a layout manager, might want to know the maximum width of such a component, even if there aren't any around at the moment. Since maximumWidth is a static variable, the layout manager can get this information.

An instance of the `Color` class represents a color. For convenience, the `Color` class contains some static, predefined color objects with friendly names like `green`, `red`, and (my favorite) `magenta`. `Color.green` is thus a predefined `Color` object that is set to a green color. In this case, these static members of `Color` are not changeable, so they are effectively constants and can be optimized as such by the compiler. Constant static members make up for the lack of a `#define` construct in Java. However, static variables don't, in general, have to be constant. I'll say more about static class members in Chapter 5, *Objects in Java*. The alternative to using these predefined colors is to create a color manually by specifying its red, green, blue (RGB) components using a `Color` class constructor.

Arrays

Next, we turn our attention to the array. We have declared a variable called `someColors`, which is an array of `Color` objects. Arrays are syntactically supported by the Java language; however, they are true, first-class objects. This means that an array is, itself, a type of object that knows how to hold an indexed list of some other type of object. An array is indexed by integers; when you index an array, the resulting value is the object in the corresponding slot in the array. Our code uses the `colorIndex` variable to index `someColors`. It's also possible to have an array of simple primitive types, rather than objects.

When we declare an array, we can initialize it by using the familiar C-like curly brace construct. Specifying a comma-separated list of elements inside of curly braces is a convenience that instructs the compiler to create an instance of the array with those elements and assign it to our variable. Alternatively, we could have just declared our `someColors` variable and, later, allocated an array object for it and assigned individual elements to that array's slots. See Chapter 4, *The Java Language* for a complete discussion of arrays.

Our Color Methods

So, we now have an array of `Color` objects and a variable with which to index them. What do we do with them? Well, we have declared two private methods that do the actual work for us. The `private` modifier on these methods specifies that they can be called only by other methods in the same instance of the class. They are not visible outside of the object. We declare members to be `private` to hide the detailed inner workings of a class from the outside world. This is called *encapsulation* and is another tenet of object-oriented design, as well as good programming practice. Private methods are also often created as helper functions for use solely within the class implementation.

The first method, `currentColor()`, is simply a convenience routine that returns the `Color` object representing the current text color. It returns the `Color` object in the `someColors` array at the index specified by our `colorIndex` variable:

```
synchronized private Color currentColor() {
    return someColors[ colorIndex ];
}
```

We could just as readily have used the expression `someColors[colorIndex]` everywhere we use currentColor(); however, creating methods to wrap common tasks is another way of shielding ourselves from the details of our class. In an alternative implementation, we might have shuffled off details of all color-related code into a separate class. We could have created a class that takes an array of colors in its constructor and then provided two methods: one to ask for the current color and one to cycle to the next color (just some food for thought).

The second method, changeColor(), is responsible for incrementing the colorIndex variable to point to the next Color in the array. changeColor() is called from our action() method whenever the button is pressed.

```
synchronized private void changeColor() {
    if ( ++colorIndex == someColors.length )
        colorIndex = 0;
    setForeground( currentColor() );
    repaint();
}
```

We increment colorIndex and compare it to the length of the someColors array. All array objects have a variable called length that specifies the number of elements in the array. If we have reached the end of the array, we reset our index to zero and start over. After changing the currently selected color, we do two things. First, we call the applet's setForeground() method, which changes the color used to draw text in the applet and the color of the button's label. Then we call repaint() to cause the applet to be redrawn with the new color for the draggable message.

So, what is the synchronized keyword that appears in front of our currentColor() and changeColor() methods? Synchronization has to do with threads, which we'll examine in the next section. For now, all you need know is that the synchronized keyword indicates these two methods can never be running at the same time. They must always run one after the other.

The reason is that in changeColor() we increment colorIndex before testing its value. That means that for some brief period of time while Java is running through our code, colorIndex can have a value that is past the end of our array. If our currentColor() method happened to run at that same moment, we would see a run-time "array out of bounds" error. There are, of course, ways in which we could fudge around the problem in this case, but this simple example is representative of more general synchronization issues we need to address. In the next section, you'll see that Java makes dealing with these problems easy through language-level synchronization support.

2.4 Hello Web! IV: Netscape's Revenge

We have explored quite a few features of Java with the first three versions of the HelloWeb applet. But until now, our applet has been rather passive; it has waited patiently for events to come its way and responded to the whims of the user. Now our applet is going to take some initiative--HelloWeb4 will blink! The code for our latest version is shown below.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class HelloWeb4 extends Applet
    implements MouseMotionListener, ActionListener, Runnable {
    int messageX = 125, messageY = 95;
    String theMessage;
    Button theButton;
    int colorIndex = 0;
    static Color[] someColors = {
        Color.black, Color.red, Color.green, Color.blue, Color.magenta };
    Thread blinkThread;
    boolean blinkState;
    public void init() {
        theMessage = getParameter("message");
        theButton = new Button("Change Color");
        add(theButton);
        addMouseMotionListener(this);
        theButton.addActionListener(this);
    }

    public void paint( Graphics graphics ) {
        graphics.setColor( blinkState ? Color.white : currentColor() );
        graphics.drawString( theMessage, messageX, messageY );
    }
    public void mouseDragged( MouseEvent e ) {
        messageX = e.getX();
        messageY = e.getY();
        repaint();
    }
    public void mouseMoved( MouseEvent e ) { }
    public void actionPerformed( ActionEvent e ) {
        if ( e.getSource() == theButton ) {
            changeColor();
        }
    }
    synchronized private void changeColor() {
        if ( ++colorIndex == someColors.length )
            colorIndex = 0;
        theButton.setForeground( currentColor() );
        repaint();
    }
    synchronized private Color currentColor() {
        return someColors[ colorIndex ];
    }
    public void run() {
        while ( true ) {
            blinkState = !blinkState;
            repaint();
            try {
                Thread.sleep(500);
            } catch (Exception e ) {
                // Handle error condition here...
            }
        }
    }
    public void start() {
        if ( blinkThread == null ) {
            blinkThread = new Thread(this);
            blinkThread.start();
        }
    }
    public void stop() {
        if ( blinkThread != null ) {
            blinkThread.stop();
            blinkThread = null;
        }
    }
}
```

If you create `HelloWeb4` as you have the other applets and then run it in a Java-enabled Web browser, you'll see that the text does in fact blink. My apologies if you don't like blinking text--I'm not overly fond of it either--but it does make for a simple, instructive example.

Threads

All the changes we've made in `HelloWeb4` have to do with setting up a separate thread of execution to make the text of our applet blink. Java is a *multithreaded* language, which means there can be many threads running at the same time. A *thread* is a separate flow of control within a program. Conceptually, threads are similar to processes, except that unlike processes, multiple threads share the same address space, which means that they can share variables and methods (but they have their own local variables). Threads are also quite lightweight in comparison to processes, so it's conceivable for a single application to be running hundreds of threads concurrently.

Multithreading provides a way for an application to handle many different tasks at the same time. It's easy to imagine multiple things going on at the same time in an application like a Web browser. The user could be listening to an audio clip while scrolling an image, and in the background the browser is downloading an image. Multithreading is especially useful in GUI-based applications, as it can improve the interactive performance of these applications.

Unfortunately for us, programming with multiple threads can be quite a headache. The difficulty lies in making sure routines are implemented so they can be run by multiple concurrent threads. If a routine changes the value of a state variable, for example, then only one thread can be executing the routine at a time. Later in this section, we'll examine briefly the issue of coordinating multiple thread's access to shared data. In other languages, synchronization of threads can be an extremely complex and error-prone endeavor. You'll see that Java gives you a few simple tools that help you deal with many of these problems. Java threads can be started, stopped, suspended, and prioritized. Threads are preemptive, so a higher priority thread can interrupt a lower priority thread when vying for processor time. See Chapter 6, *Threads* for a complete discussion of threads.

The Java run-time system creates and manages a number of threads. I've already mentioned the AWT thread, which manages `repaint()` requests and event processing for GUI components that belong to the `java.awt` package. A Java-enabled Web browser typically has at least one separate thread of execution it uses to manage the applets it displays. Until now, our example has done most of its work from methods of the `Applet` class, which means that is has been borrowing time from these threads. Methods like `mouseDragged()` and `actionPerformed()` are invoked by the AWT thread and run on its time. Similarly, our `init()` method is called by a thread in the Web browser. This means we are somewhat limited in the amount of processing we do within these methods. If we were, for instance, to go into an endless loop in our `init()` method, our applet would never appear, as it would never finish initializing. If we want an applet to perform any extensive processing, such as animation, a lengthy calculation, or communication, we should create separate threads for these tasks.

The Thread Class

As you might have guessed, threads are created and controlled as `Thread` objects. We have added a new instance variable, blinkThread, to our example to hold the Thread that handles our blinking activity:

```
Thread blinkThread;
```

An instance of the Thread class corresponds to a single thread. It contains methods to start, control, and stop the thread's execution. Our basic plan is to create a Thread object to handle our blinking code. We call the Thread's start() method to begin execution. Once the thread starts, it continues to run until we call the Thread's stop() method to terminate it.

But Java doesn't allow pointers to methods, so how do we tell the thread which method to run? Well, the Thread object is rather picky; it always expects to execute a method called run() to perform the action of the thread. The run() method can, however, with a little persuasion, be located in any class we desire.

We specify the location of the run() method in one of two ways. First, the Thread class itself has a method called run(). One way to execute some Java code in a separate thread is to subclass Thread and override its run() method to do our bidding. In this case, we simply create an instance of this subclass and call its start() method.

But it's not always desirable or possible to create a subclass of Thread to contain our run() method. In this case, we need to tell the Thread which object contains the run() method it should execute. The Thread class has a constructor that takes an object reference as its argument. If we create a Thread object using this constructor and call its start() method, the Thread executes the run() method of the target object, rather than its own. In order to accomplish this, Java needs to have a guarantee that the object we are passing it does indeed contain a compatible run() method. We already know how to make such a guarantee: we use an interface. Java provides an interface named Runnable that must be implemented by any class that wants to become a Thread.

The Runnable Interface

The second technique I described for creating a Thread object involved passing an object that implements the Runnable interface to the Thread constructor. The Runnable interface specifies that the object contains a run() method that takes no arguments and returns no value. This method is called automatically when the system needs to start the thread.

Sticking with our technique for implementing our applet in a single class, we have opted to add the run() method for blinkThread to our HelloWeb4 class. This means that HelloWeb4 needs to implement the Runnable interface. We indicate that the class implements the interface in our class declaration:

```
public class HelloWeb4 extends Applet
    implements MouseMotionListener, ActionListener, Runnable {...}
```

At compile time, the Java compiler checks to make sure we abide by this statement. We have carried through by adding an appropriate run() method to our applet. Our run() method has the task of changing the color of our text a couple of times a second. It's a very short routine, but I'm going to delay looking at it until we tie up some loose ends in dealing with the Thread itself.

start( ) and stop( )

Now that we know how to create a Thread to execute our applet's run( ) method, we need to figure out where to actually do it. The start( ) and stop( ) methods of the Applet class are similar to init( ). The start( ) method is called when an applet is first displayed. If the user then leaves the Web document or scrolls the applet off the screen, the stop( ) method is invoked. If the user subsequently returns, the start( ) method is called again, and so on. Unlike init( ), start( ) and stop( ) can be called repeatedly during the lifetime of an applet.

The start( ) and stop( ) methods of the Applet class have absolutely nothing to do with the Thread object, except that they are a good place for an applet to start and stop a thread. An applet is responsible for managing the threads that it creates. It would be considered rude for an applet to continue such tasks as animation, making noise, or performing extensive calculations long after it's no longer visible on the screen. It's common practice, therefore, to start a thread when an applet becomes visible and stop it when the applet is no longer visible.

Here's the start( ) method from HelloWeb4:

```
public void start() {
    if ( blinkThread == null ) {
        blinkThread = new Thread(this);
        blinkThread.start();
    }
}
```

The method first checks to see if there is an object assigned to blinkThread; recall that an uninitialized instance variable has a default value of null. If not, the method creates a new instance of Thread, passing the target object that contains the run( ) method to the constructor. Since HelloWeb4 contains our run( ) method, we pass the special variable this to the constructor to let the thread know where to find the run( ) method it should run. this always refers to our object. Finally, after creating the new Thread, we call its start( ) method to begin execution.

Our stop( ) method takes the complimentary position:

```
public void stop() {
    if ( blinkThread != null ) {
        blinkThread.stop();
        blinkThread = null;
    }
}
```

This method checks to see if `blinkThread` is empty. If not, it calls the thread's `stop()` method to terminate its execution. By setting the value of `blinkThread` back to `null`, we have eliminated all references to the thread object we created in the `start()` method, so the garbage collector can dispose of the object.

run( )

Our `run()` method does its job by setting the value of the variable `blinkState`. We have added `blinkState`, a `boolean` value, to represent whether we are currently blinking on or off:

```
boolean blinkState;
```

The `setColor()` line of our `paint()` method has been modified slightly to handle blinking. The call to `setColor()` now draws the text in white when `blinkState` is `true`:

```
gc.setColor( blinkState ? Color.white : currentColor() );
```

Here we are being somewhat terse and using the C-like ternary operator to return one of two alternate color values based on the value of `blinkState`.

Finally, we come to the `run()` method itself:

```
public void run() {
    while ( true ) {
        blinkState = !blinkState;
        repaint();
        try {
            Thread.sleep(500);
        }
        catch (InterruptedException e ) {
        }
    }
}
```

At its outermost level, `run()` uses an infinite `while` loop. This means the method will run continuously until the thread is terminated by a call to the controlling `Thread` object's `stop()` method.

The body of the loop does three things on each pass:

- Flips the value of `blinkState` to its opposite value using the `not` operator, `!`.
- Calls `repaint()` so that our `paint()` method can have an opportunity to redraw the text in

accordance with blinkState.

- Uses a try/catch statement to trap for an error in our call to the sleep() method of the Thread class. sleep() is a static method of the Thread class. The method can be invoked from anywhere and has the effect of putting the current thread to sleep for the specified number of milliseconds. The effect here is to give us approximately two blinks per second.

Exceptions

The try/catch statement in Java is used to handle special conditions called *exceptions*. An exception is a message that is sent, normally in response to an error, during the execution of a statement or a method. When an exceptional condition arises, an object is created that contains information about the particular problem or condition. Exceptions act somewhat like events. Java stops execution at the place where the exception occurred, and the exception object is said to be *thrown* by that section of code. Like events, an exception must be delivered somewhere and handled. The section of code that receives the exception object is said to *catch* the exception. An exception causes the execution of the instigating section of code to abruptly stop and transfers control to the code that receives the exception object.

The try/catch construct allows you to catch exceptions for a section of code. If an exception is caused by a statement inside of a try clause, Java attempts to deliver the exception to the appropriate catch clause. A catch clause looks like a method declaration with one argument and no return type. If Java finds a catch clause with an argument type that matches the type of the exception, that catch clause is invoked. A try clause can have multiple catch clauses with different argument types; Java chooses the appropriate one in a way that is analogous to the selection of overloaded methods.

If there is no try/catch clause surrounding the code, or a matching catch clause is not found, the exception is thrown up the call stack to the calling method. If the exception is not caught there, it's thrown up another level, and so on until the exception is handled. This provides a very flexible error-handling mechanism, so that exceptions in deeply nested calls can bubble up to the surface of the call stack for handling. As a programmer, you need to know what exceptions a particular statement can generate, so methods in Java are required to declare the exceptions they can throw. If a method doesn't handle an exception itself, it must specify that it can throw that exception, so that the calling method knows that it may have to handle it. See Chapter 4, *The Java Language* for a complete discussion of exceptions and the try/catch clause.

So, why do we need a try/catch clause around our sleep() call? What kind of exception can Thread's sleep() method throw and why do we care about it, when we don't seem to check for exceptions anywhere else? Under some circumstances, Thread's sleep() method can throw an InterruptedException, indicating that it was interrupted by another thread. Since the run() method specified in the Runnable interface doesn't declare it can throw an InterruptedException, we must catch it ourselves, or the compiler will complain. The try/catch statement in our example has an empty catch clause, which means that it handles the exception by ignoring it. In this case, our thread's functionality is so simple it doesn't matter if it's interrupted. All of the other methods we have used either handle their own exceptions or throw only general-purpose exceptions that are assumed to be possible everywhere and don't need to be explicitly declared.

A Word About Synchronization

At any given time, there can be a number of threads running in the Java interpreter. Unless we explicitly coordinate them, these threads will be executing methods without any regard for what the other threads are doing. Problems can arise when these methods share the same data. If one method is changing the value of some variables at the same time that another method is reading these variables, it's possible that the reading thread might catch things in the middle and get some variables with old values and some with new. Depending on the application, this situation could cause a critical error.

In our `HelloWeb` examples, both our `paint()` and `mouseDrag()` methods access the `messageX` and `messageY` variables. Without knowing the implementation of our particular Java environment, we have to assume that these methods could conceivably be called by different threads and run concurrently. `paint()` could be called while `mouseDrag()` is in the midst of updating `messageX` and `messageY`. At that point, the data is in an inconsistent state and if `paint()` gets lucky, it could get the new x value with the old y value. Fortunately, in this case, we probably would not even notice if this were to happen in our application. We did, however, see another case, in our `changeColor()` and `currentColor()` methods, where there is the potential for a more serious "out of bounds" error to occur.

The `synchronized` modifier tells Java to acquire a *lock* for the class that contains the method before executing that method. Only one method can have the lock on a class at any given time, which means that only one synchronized method in that class can be running at a time. This allows a method to alter data and leave it in a consistent state before a concurrently running method is allowed to access it. When the method is done, it releases the lock on the class.

Unlike synchronization in other languages, the `synchronized` keyword in Java provides locking at the language level. This means there is no way that you can forget to unlock a class. Even if the method throws an exception or the thread is terminated, Java will release the lock. This feature makes programming with threads in Java much easier than in other languages. See Chapter 6, *Threads* for more details on coordinating threads and shared data.

Whew! Now it's time to say goodbye to `HelloWeb`. I hope that you have developed a feel for the major features of the Java language, and that this will help you as you go on to explore the details of programming with Java.

3.2 The Class Path

The concept of a path should be familiar to anyone who has worked on a DOS or UNIX platform. It's a piece of environment information that provides an application with a list of places to look for some resource. The most common example is a path for executable programs. In a UNIX shell, the `PATH` environment variable is a colon-separated list of directories that are searched, in order, when the user types the name of a command. The Java `CLASSPATH` environment variable, similarly, is a list of locations that can be searched for packages containing Java class files. Both the Java interpreter and the Java compiler use `CLASSPATH` when searching for files on the local host platform.

Classes loaded from the local host via the class path have special features. For example, the Java interpreter loads classes in the class path just once; after a core class has been loaded, it can't be modified or replaced. The interpreter can also be told to trust classes in the class path and to load them without passing them through the byte-code verification process. This is important because certain kinds of optimizations on Java virtual machine instructions produce valid byte-code that, nonetheless, can't pass the verification process. Byte-code that is precompiled on the native host is an extreme example.

The class path is a list of locations where Java class packages are found. A location can be a path such as a directory name or the name of a class archive file. Java supports archives of class files in the uncompressed ZIP format.[1] It automatically looks inside ZIP archives and retrieves classes, which then allows large groups of classes to be distributed in a single archive file. The precise means and format for setting the class path varies from system to system. On a UNIX system, you set the `CLASSPATH` environment variable with a colon-separated list of directories and class archive files:

> [1] The ZIP format is an open standard for file archiving and compression. There are ZIP utilities available for most platforms; you'll need to get one if you want to store Java classes in ZIP archives. Use ftp://ftp.uu.net/pub/archiving/zip/ to access an archive of freely available ZIP utilities.

```
CLASSPATH=/usr/lib/java/classes.zip:/home/vicky/Java/classes:\
    /home/vicky/.netscape/moz2_0.zip:.
```

On a Windows system, the `CLASSPATH` environment variable is set with a semicolon-separated list of directories and class archive files:

```
CLASSPATH=C:\tools\java\classes.zip;D:\users\vicky\Java\classes;.
```

The class path can also be set with the `-classpath` option to the Java interpreter *java* and the Java compiler *javac*.

The above UNIX example specifies a class path with four locations: a ZIP archive in */usr/lib/java*, a directory in the user's home, another ZIP file in the user's Netscape collection, and the current directory, which is always specified with a dot (`.`). The last component of the class path, the current directory, is useful when tinkering with classes, but as a general rule, it's bad practice to put the current directory in any kind of path.

The Java interpreter searches each of these four locations in order to find classes. *java* expects to find class files in a directory hierarchy or in a directory within a ZIP archive that maps to the fully qualified name of the class. The components of a class-package name become the components of a pathname. Given the above class path, the first time we reference a class with the fully qualified name of `animals.birds.BigBird`, for example, *java* begins the search with the *classes.zip* archive in */usr/lib/java*. It looks for a class archived under the path *animals/birds/BigBird*. If *java* does not find the class there, it looks for the class in */home/vicky/Java/classes/animals/birds/BigBird*. If it's not found there, *java* moves on to the archive file specified next in the class path, and so on.

If you don't specify the `CLASSPATH` environment variable or the `-classpath` option, *java* uses the following default class path:

```
.:$JAVA/classes:$JAVA/lib/classes.zip     UNIX systems
.;$JAVA\classes;$JAVA\lib\classes.zip     Windows systems
```

In this path, `$JAVA` is the main Java directory on your system. Notice that the current directory (`.`) is the first location in the default class path; this means the files in your current directory are always available. If you change the class path and don't include the current directory, these files will no longer be accessible.

PREVIOUS | HOME | NEXT
The Java Interpreter | BOOK INDEX | The Java Compiler

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

3.3 The Java Compiler

In this section, I'm going to say a few words about *javac*, the Java compiler that is supplied as part of Sun's JDK. (If you are happily working in another development environment, you may want to skip ahead to the next section.) The *javac* compiler is written entirely in Java, so it's available for any platform that supports the Java run-time system. The ability to support its own development environments is an important stage in a language's development. Java makes this bootstrapping automatic by supplying a ready-to-run compiler at the same cost as porting the interpreter.

javac turns Java source code into a compiled class that contains Java virtual machine byte-code. By convention, source files are named with a *.java* extension; the resulting class files have a *.class* extension. *javac* considers a file to be a single compilation unit. As you'll see in Chapter 5, *Objects in Java*, classes in a given compilation unit share certain features like `package` and `import` statements.

javac allows you one `public` class per file and insists the file have the same name as the class. If the filename and class name don't match, *javac* issues a compilation error. A single file can contain multiple classes, as long as only one of the classes is `public`. You should avoid packing lots of classes into a single file. The compiler lets you include extra non-`public` classes in a *.java* file, so that you can implement a class that is tightly coupled to another class without a lot of hassle. But you should have more than one class in a file if the `public` class in the file is the only one that ever uses additional classes.

Now for an example. The source code for the following class should be placed in a file called *BigBird.java*:

```
package animals.birds

public class BigBird extends Bird {
    ...
}
```

We can then compile it with:

```
% javac BigBird.java
```

Unlike the Java interpreter, which takes a class name as its argument, *javac* requires an actual filename to process. The above command produces the class file *BigBird.class* and stores it in the same directory as the source file. While it's useful to have the class file in the same directory as the source when you are working on a simple example, for most real applications you'll need to store the class file in an appropriate place in the class path.

You can use the `-d` option to *javac* to specify an alternate directory for storing the class files it generates. The specified directory is used as the root of the class hierarchy, so *.class* files are placed in this directory or in a subdirectory below it, depending on the package name of the class. For example, we can use the following command to put our *BigBird.class* file in an appropriate location:

```
% javac -d /home/vicky/Java/classes BigBird.java
```

When you use the `-d` option, *javac* automatically creates any directories needed to store the class file in the appropriate place. In the above command, the *BigBird.class* file is stored in */home/vicky/Java/classes/animals/birds*.

You can specify multiple *.java* files in a single *javac* command; the compiler simply creates a class file for each source file. But you don't need to list source files for other classes your class references, as long as the other classes have already been compiled. During compilation, Java resolves other class references using the class path. If our class references other classes in `animals.birds` or other packages, the appropriate paths should be listed in the class path at compile time, so that *javac* can find the appropriate class files. You either make sure that the `CLASSPATH` environment variable is set or use the `-classpath` option to *javac*.

The Java compiler is more intelligent than your average compiler and replaces some of the functionality of a *make* utility. For example, *javac* compares the modification times of the source and class files for all referenced classes and recompiles them as necessary. A compiled Java class remembers the source file from which it was compiled, so as long as the source file is in the same directory as the class file, *javac* can recompile the source if necessary. If, in the above example, class `BigBird` references another class, `animals.furry.Grover`, *javac* looks for the source *Grover.java* in an `animals.furry` package and recompiles it if necessary to bring the *Grover.class* class file up to date.

It's important to note that *javac* can do its job even if only the compiled versions of referenced classes are available. Java class files contain all the data type and method signature information source files do, so compiling against binary class files is as type safe (and exception safe) as compiling with Java source code.

PREVIOUS | HOME | NEXT
The Class Path | BOOK INDEX | The Netscape Alternative

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

3.5 The Applet Tag

```
Add stuff about 'archive' tag.
```

Applets are embedded in HTML documents with the `<applet>` tag. The `<applet>` tag resembles the HTML `<img>` image tag.[2] It contains attributes that identify the applet to be displayed and, optionally, give the Web browser hints about how it should be displayed. The standard image tag sizing and alignment attributes, such as `height` and `width`, can be used inside the applet tag. Unlike images, however, applets have both an opening `<applet>` and a closing `</applet>` tag. Sandwiched between these can be any number of `<param>` tags that contain application-specific parameters to be passed to the applet itself:

> [2] If you are not familiar with HTML or other markup languages, you may want to refer to *HTML: The Definitive Guide*, from O'Reilly & Associates, for a complete reference on HTML and structured Web documents.

```
<applet attribute [ attribute ] ...  >
[<param parameter >]
[<param parameter >]
...
<\applet>
```

Attributes

Attributes are name/value pairs that are interpreted by a Web browser or applet viewer. (Many HTML tags besides `<applet>` have attributes.) Attributes of the `<applet>` tag specify general features that apply to all applets, such as size and alignment. The definition of the `<applet>` tag lists a fixed set of recognized attributes; specifying an incorrect or nonexistent attribute should be considered an HTML error.

Three attributes, `code`, `width`, and `height`, are always required in the `<applet>` tag. `code` specifies the name of the applet to be loaded; `width` and `height` determine its initial size. Other attributes are optional.

The following is an HTML fragment for a hypothetical, simple clock applet that takes no parameters and requires no special HTML layout:

```
<applet code=AnalogClock width=100 height=100></applet>
```

The HTML file that contains this `<applet>` tag needs to be stored in the same directory as the `AnalogClock.class` class file. The applet tag is not sensitive to spacing, so the above is therefore equivalent to:

```
<applet
    code=AnalogClock
    width=100
    height=100>
<\applet>
```

You can use whatever form seems appropriate.

Parameters

Parameters are analogous to command-line arguments; they provide a way to pass information to an applet. Each `<param>` tag contains a name and a value that are passed as strings to the applet:

```
<param name = parameter_name value = parameter_value>
```

Parameters provide a means of embedding application-specific data and configuration information within an HTML document.[3] Our `AnalogClock` applet, for example, might accept a parameter that selects between local and universal time:

> [3] If you are wondering why the applet's parameters are specified in yet another type of tag, here's the reason. In the original alpha release of Java, applet parameters were included inside of a single `<app>` tag along with formatting attributes. However, this format was not SGML-compliant, so the `<param>` tag was added.

```
<applet code=AnalogClock width=100 height=100>
    <param name=zone value=GMT>
<\applet>
```

Presumably, this `AnalogClock` applet is designed to look for a parameter named `zone` with a possible value of `GMT`.

Parameter names and values can be quoted to contain spaces and other special characters. We could therefore be more verbose and use a parameter value like the following:

```
<param name=zone value="Greenwich Mean Time">
```

The parameters a given applet expects are determined by the developer of that applet. There is no fixed set of parameter names or values; it's up to the applet to interpret the parameter name/value pairs that are passed to it. Any number of parameters can be specified, and the applet may choose to use or ignore them as it sees fit. The applet might also consider parameters to be either optional or required and act accordingly.

Hablo Applet?

Web browsers ignore tags they don't understand; if the Web browser doesn't interpret the `<applet>` or `<param>` tags, they should disappear and any HTML between the `<applet>`and `</applet>` tags should appear normally.

By convention, Java-enabled Web browsers do the opposite and ignore any extra HTML between the `<applet>` and `</applet>` tags. This means we can place some alternate HTML inside the `<applet>` tag, which is then displayed only by Web browsers that can't run the applet.

For our `AnalogClock` example, we could display a small text explanation and an image of the clock applet as a teaser:

```
<applet code=AnalogClock width=100 height=100>
    <param name=zone value=GMT>
    <strong>If you see this you don't have a Java-enabled Web
    browser. Here's a picture of what you are missing.</strong>
    <img src="clockface.gif">
<\applet>
```

The Complete Applet Tag

We can now spell out the full-blown `<applet>` tag:[4]

[4] The HTML working group of the IETF is investigating standardization of embedded objects in HTML. A draft document can be found at ftp://ds.internic.net/internet-drafts/draft-ietf-html-cda-00.txt. They would prefer a slightly less application-centric name such as `<embed>`. However, their discussion, for the most part, parallels the `<applet>` tag.

```
<applet
    code = class name
    width = pixels wide
    height = pixels high
    [ codebase = location URL ]
    [ name = instance name ]
    [ alt = alternate text ]
    [ align = alignment style ]
    [ vspace = vertical pad pixels ]
    [ hspace = horizontal pad pixels ]
>
[ <param name = parameter name value = parameter value> ]
[ <param name = parameter name value = parameter value> ]
...
[ HTML for non Java aware browsers ]
<\applet>
```

The `width`, `height`, `align`, `vspace`, and `hspace` attributes have the same meanings as those of the `<img>` tag and determine the preferred size, alignment, and padding respectively.

The `alt` attribute specifies alternate text that is displayed by browsers that understand the `<applet>` tag and its attributes, but can't actually run applets. This attribute can also describe the applet, since in this case any alternate HTML between `<applet>` and `<\applet>` is ignored.

The `name` attribute specifies an instance name for the executing applet. This is a name specified as a unique label for each copy of an applet on a particular HTML page. For example, if we include our clock twice on the same page (using two applet tags), we could give each instance a unique name to differentiate them:

```
<applet code=AnalogClock name="bigClock" width=300 height=300><\applet>
<applet code=AnalogClock name="smallClock" width=50 height=50><\applet>
```

Applets use instance names to recognize and communicate with other applets on the same page. We could, for instance, create a "clock setter" applet that knows how to set the time on a `AnalogClock` applet and pass it the instance name of a particular target clock on this page as a parameter. This might look something like:

```
<applet code=ClockSetter>
    <param name=clockToSet value="bigClock">
<\applet>
```

Loading Class Files

The `code` attribute of the `<applet>` tag should specify the name of an applet. This is either a simple class name, or a package path and class name. For now, let's look at simple class names; I'll discuss packages in a moment. By default, the Java run-time system looks for the class file in the same location as the HTML document that contains it. This location is known as the base URL for the document.

Consider an HTML document, *clock.html*, that contains our clock applet example:

```
<applet code=AnalogClock width=100 height=100><\applet>
```

Let's say we retrieve the document at the following URL:

```
http://www.time.ch/documents/clock.html
```

Java tries to retrieve the applet class file from the same base location:

```
http://www.time.ch/documents/AnalogClock.class
```

The `codebase` attribute of the `<applet>` tag can be used to specify an alternative base URL for the class file search. Let's say our HTML document now specifies `codebase`, as in the following example:

```
<applet
    codebase=http://www.joes.ch/stuff/
    code=AnalogClock
    width=100
    height=100>
<\applet>
```

Java now looks for the applet class file at:

```
http://www.joes.ch/stuff/AnalogClock.class
```

Packages

Packages are groups of Java classes; see Chapter 5, *Objects in Java* for more information. A package name is a little like an Internet hostname, in that they both use a hierarchical, dot-separated naming convention. A Java class file can be identified by its full name by prefixing the class name with the package name. We might therefore have a package called `time` for time-related Java classes, and perhaps a subordinate package called `time.clock` to hold classes related to one or more clock applications.

In addition to providing a naming scheme, packages can be used to locate classes stored at a particular location. Before a class file is retrieved from a server, its package-name component is translated by the client into a relative path name under the base URL of the document. The components of a class package name are simply turned into the components of a path name, just like with classes on your local system.

Let's suppose that our `AnalogClock` has been placed into the `time.clock` package and now has the fully qualified name of `time.clock.AnalogClock`. Our simple `<applet>` tag would now look like:

```
<applet code=time.clock.AnalogClock width=100 height=100><\applet>
```

Let's say the *clock.html* document is once again retrieved from:

```
http://www.time.ch/documents/clock.html
```

Java now looks for the class file in the following location:

```
http://www.time.ch/documents/time/clock/AnalogClock.class
```

The same is true when specifying an alternative `codebase`:

```
<applet
    codebase=http://www.joes.ch/stuff/
    code=time.clock.AnalogClock
    width=100
    height=100>
<\applet>
```

Java now tries to find the class in the corresponding path under the new base URL:

```
http://www.joes.ch/stuff/time/clock/AnalogClock.class
```

One possible package-naming convention proposes that Internet host and domain names be incorporated as part of package names to form a unique identifier for classes produced by a given organization. If a company with the domain name `foobar.com` produced our `AnalogClock` class, they might distribute it in a package called `com.foobar.time.clock`. The fully qualified name of the `AnalogClock` class would then be `com.foo.time.clock.AnalogClock`. This would presumably be a unique name stored on an arbitrary server. A future version of the Java class loader might use this to automatically search for classes on remote hosts.

Perhaps soon we'll run Sun's latest and greatest Web browser directly from its source with:

```
% java com.sun.java.hotjava.HotJava
```

Viewing Applets

Sun's JDK comes with an applet-viewer program aptly called *appletviewer*. To use *appletviewer*, specify the URL of the document on the command line. For example, to view our `AnalogClock` at the URL shown above, use the following command:

```
% appletviewer http://www.time.ch/documents/clock.html
```

The *appletviewer* retrieves all applets in the specified document and displays each one in a separate window. *appletviewer* is not a Web browser; it doesn't attempt to display HTML. It's primarily a convenience for testing and debugging applets. If the document doesn't contain `<applet>` tags, *appletviewer* does nothing.

PREVIOUS | HOME | NEXT
The Netscape Alternative | BOOK INDEX | The Java Language

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

4.2 Comments

Java supports both C-style block comments delimited by `/*` and `*/` and C++-style line comments indicated by `//`:

```
/*  This is a
        multiline
            comment.    */

// This is a single line comment
// and so // is this
```

As in C, block comments can't be nested. Single-line comments are delimited by the end of a line; extra `//` indicators inside a single line have no effect. Line comments are useful for short comments within methods because you can still wrap block comments around large chunks of code during development.

By convention, a block comment beginning with `/**` indicates a special "doc comment." A doc comment is commentary that is extracted by automated documentation generators, such as Sun's *javadoc* program that comes with the Java Development Kit. A doc comment is terminated by the next `*/`, just as with a regular block comment. Leading spacing up to a `*` on each line is ignored; lines beginning with `@` are interpreted as special tags for the documentation generator:

```
/**
 * I think this class is possibly the most amazing thing you will
 * ever see. Let me tell you about my own personal vision and
 * motivation in creating it.
 * <p>
 * It all began when I was a small child, growing up on the
 * streets of Idaho. Potatoes were the rage, and life was good...
 *
 * @see PotatoPeeler
 * @see PotatoMasher
 * @author John 'Spuds' Smith
 * @version 1.00, 19 Dec 1996
 */
```

javadoc creates HTML class documentation by reading the source code and the embedded comments. The author and version information is presented in the output and the `@see` tags make hypertext links to the appropriate class documentation. The compiler also looks at the doc comments; in particular, it is interested in the `@deprecated` tag, which means that the method has been declared obsolete and should be avoided in new programs. The compiler generates a warning message whenever it sees you use a deprecated feature in your code.

Doc comments can appear above class, method, and variable definitions, but some tags may not be applicable to all. For example, a variable declaration can contain only a `@see` tag. Table 4.1 summarizes the tags used in doc comments.

Table 4.1: Doc Comment Tags

Tag	**Description**	**Applies to**
`@see`	Associated class name	Class, method, or variable
`@author`	Author name	Class
`@version`	Version string	Class
`@param`	Parameter name and description	Method
`@return`	Description of return value	Method
`@exception`	Exception name and description	Method
`@deprecated`	Declares an item obsolete	Class, method, or variable

4.3 Types

The type system of a programming language describes how its data elements (variables and constants) are associated with actual storage. In a statically typed language, such as C or C++, the type of a data element is a simple, unchanging attribute that often corresponds directly to some underlying hardware phenomenon, like a register value or a pointer indirection. In a more dynamic language like Smalltalk or Lisp, variables can be assigned arbitrary elements and can effectively change their type throughout their lifetime. A considerable amount of overhead goes into validating what happens in these languages at run-time. Scripting languages like Tcl and awk achieve ease of use by providing drastically simplified type systems in which only certain data elements can be stored in variables, and values are unified into a common representation such as strings.

As I described in Chapter 1, *Yet Another Language?*, Java combines the best features of both statically and dynamically typed languages. As in a statically typed language, every variable and programming element in Java has a type that is known at compile-time, so the interpreter doesn't normally have to check the type validity of assignments while the code is executing. Unlike C or C++ though, Java also maintains run-time information about objects and uses this to allow safe run-time polymorphism.

Java data types fall into two categories. *Primitive types* represent simple values that have built-in functionality in the language; they are fixed elements like literal constants and numeric expressions. *Reference types* (or class types) include objects and arrays; they are called reference types because they are passed "by reference" as I'll explain shortly.

Primitive Types

Numbers, characters, and boolean values are fundamental elements in Java. Unlike some other (perhaps more pure) object-oriented languages, they are not objects. For those situations where it's desirable to treat a primitive value as an object, Java provides "wrapper" classes (see Chapter 7, *Basic Utility Classes*). One major advantage of treating primitive values as such is that the Java compiler can more readily optimize their usage.

Another advantage of working with the Java virtual-machine architecture is that primitive types are precisely defined. For example, you never have to worry about the size of an `int` on a particular platform; it's always a 32-bit, signed, two's complement number. Table 4.2 summarizes Java's primitive types.

Table 4.2: Java Primitive Data Types

Type	**Definition**
`boolean`	`true` or `false`
`char`	16-bit Unicode character
`byte`	8-bit signed two's complement integer
`short`	16-bit signed two's complement integer
`int`	32-bit signed two's complement integer
`long`	64-bit signed two's complement integer
`float`	32-bit IEEE 754 floating-point value
`double`	64-bit IEEE 754 floating-point value

If you think the primitive types look like an idealization of C scalar types on a byte-oriented 32-bit machine, you're absolutely right. That's how they're supposed to look. The 16-bit characters were forced by Unicode, and generic pointers were deleted for other reasons we'll touch on later, but in general the syntax and semantics of Java primitive types are meant to fit a C programmer's mental habits. If you're like most of this book's readers, you'll probably find this saves you a lot of mental effort in learning the language.

Declaration and initialization

Variables are declared inside of methods or classes in C style. For example:

```
int foo;
double d1, d2;
boolean isFun;
```

Variables can optionally be initialized with an appropriate expression when they are declared:

```
int foo = 42;
double d1 = 3.14, d2 = 2 * 3.14;
boolean isFun = true;
```

Variables that are declared as instance variables in a class are set to default values if they are not initialized. In this case, they act much like static variables in C or C++. Numeric types default to the appropriate flavor of zero, characters are set to the null character "\0," and boolean variables have the value false. Local variables declared in methods, on the other hand, must be explicitly initialized before they can be used.

Integer literals

Integer literals can be specified in octal (base 8), decimal (base 10), or hexadecimal (base 16). A decimal integer is specified by a sequence of digits beginning with one of the characters 1-9:

```
int i = 1230;
```

Octal numbers are distinguished from decimal by a leading zero:

```
int i = 01230;                // i = 664 decimal
```

(An interesting, but meaningless, observation is that this would make the number 0 an octal value in the eyes of the compiler.)

As in C, a hexadecimal number is denoted by the leading characters 0x or 0X (zero "x"), followed by digits and the characters a-f or A-F, which represent the decimal values 10-15 respectively:

```
int i = 0xFFFF;               // i = 65535 decimal
```

Integer literals are of type int unless they are suffixed with an L, denoting that they are to be produced as a long value:

```
long l = 13L;
long l = 13;                  // equivalent--13 is converted from type int
```

(The lowercase character l ("el") is also acceptable, but should be avoided because it often looks like the numeral 1).

When a numeric type is used in an assignment or an expression involving a type with a larger range, it can be promoted to the larger type. For example, in the second line of the above example, the number 13 has the default type of int, but it's promoted to type long for assignment to the long variable. Certain other numeric and comparison operations also cause this kind of arithmetic promotion. A numeric value can never be assigned to a type with a smaller range without an explicit (C-style) cast, however:

```
int i = 13;
byte b = i;                   // Compile time error--explicit cast needed
byte b = (byte) i;            // Okay
```

Conversions from floating point to integer types always require an explicit cast because of the potential loss of precision.

Floating-point literals

Floating-point values can be specified in decimal or scientific notation. Floating-point literals are of type `double` unless they are suffixed with an `f` denoting that they are to be produced as a `float` value:

```
double d = 8.31;
double e = 3.00e+8;
float f = 8.31F;
float g = 3.00e+8F;
```

Character literals

A literal character value can be specified either as a single-quoted character or as an escaped ASCII or Unicode sequence:

```
char a = 'a';
char newline = '\n';
char octalff = \u00ff;
```

Reference Types

In C, you can make a new, complex data type by creating a `structure`. In Java (and other object-oriented languages), you instead create a `class` that defines a new type in the language. For instance, if we create a new class called `Foo` in Java, we are also implicitly creating a new type called `Foo`. The type of an item governs how it's used and where it's assigned. An item of type `Foo` can, in general, be assigned to a variable of type `Foo` or passed as an argument to a method that accepts a `Foo` value.

In an object-oriented language like Java, a type is not necessarily just a simple attribute. Reference types are related in the same way as the classes they represent. Classes exist in a hierarchy, where a subclass is a specialized kind of its parent class. The corresponding types have a similar relationship, where the type of the child class is considered a subtype of the parent class. Because child classes always extend their parents and have, at a minimum, the same functionality, an object of the child's type can be used in place of an object of the parent's type. For example, if I create a new class, `Bar`, that extends `Foo`, there is a new type `Bar` that is considered a subtype of `Foo`. Objects of type `Bar` can then be used anywhere an object of type `Foo` could be used; An object of type `Bar` is said to be assignable to a variable of type `Foo`. This is called *subtype polymorphism* and is one of the primary features of an object-oriented language. We'll look more closely at classes and objects in Chapter 5, *Objects in Java*.

Primitive types in Java are used and passed "by value." In other words, when a primitive value is assigned or passed as an argument to a method, it's simply copied. Reference types, on the other hand, are always accessed "by reference." A *reference* is simply a handle or a name for an object. What a variable of a reference type holds is a reference to an object of its type (or of a subtype). A reference is like a pointer in C or C++, except that its type is strictly enforced and the reference value itself is a primitive entity that can't be examined directly. A reference value can't be created or changed other than through assignment to an appropriate object. When references are assigned or passed to methods, they are copied by value. You can think of a reference as a pointer type that is automatically dereferenced whenever it's mentioned.

Let's run through an example. We specify a variable of type `Foo`, called `myFoo`, and assign it an appropriate object:

```
Foo myFoo = new Foo();
Foo anotherFoo = myFoo;
```

`myFoo` is a reference type variable that holds a reference to the newly constructed `Foo` object. For now, don't worry about the details of creating an object; we'll cover that in Chapter 5, *Objects in Java*. We designate a second `Foo` type variable, `anotherFoo`, and assign it to the same object. There are now two identical references: `myFoo` and `anotherFoo`. If we change things in the state of the `Foo` object itself, we will see the same effect by looking at it with either reference. The comparable code in C++ would be:

```
// C++
Foo& myFoo = *(new Foo());
Foo& anotherFoo = myFoo;
```

We can pass one of the variables to a method, as in:

```
myMethod( myFoo );
```

An important, but sometimes confusing distinction to make at this point is that the reference itself is passed by value. That is, the argument passed to the method (a local variable from the method's point of view) is actually a third copy of the reference. The method can alter the state of the `Foo` object itself through that reference, but it can't change the caller's reference to `myFoo`. That is, the method can't change the caller's `myFoo` to point to a different Foo object. For the times we want a method to change a reference for us, we have to pass a reference to the object that contains it, as shown in Chapter 5, *Objects in Java*.

Reference types always point to objects, and objects are always defined by classes. However, there are two special kinds of reference types that specify the type of object they point to in a slightly different way. Arrays in Java have a special place in the type system. They are a special kind of object automatically created to hold a number of some other type of object, known as the base type. Declaring an array-type reference implicitly creates the new class type, as you'll see in the next section.

Interfaces are a bit sneakier. An interface defines a set of methods and a corresponding type. Any object that implements all methods of the interface can be treated as an object of that type. Variables and method arguments can be declared to be of interface types, just like class types, and any object that implements the interface can be assigned to them. This allows Java to cross the lines of the class hierarchy in a type safe way, as you'll see in Chapter 5, *Objects in Java*.

A Word About Strings

Strings in Java are objects; they are therefore a reference type. `String` objects do, however, have some special help from the Java compiler that makes them look more primitive. Literal string values in Java source code are turned into `String` objects by the compiler. They can be used directly, passed as arguments to methods, or assigned to `String` type variables:

```
System.out.println( "Hello World..." );
String s = "I am the walrus...";
String t = "John said: \"I am the walrus...\"";
```

The `+` symbol in Java is overloaded to provide string concatenation; this is the only overloaded operator in Java:

```
String quote = "Four score and " + "seven years ago,";
String more = quote + " our" + " fathers" +  " brought...";
```

Java builds a single `String` object from the concatenated strings and provides it as the result of the expression. We will discuss the `String` class in Chapter 7, *Basic Utility Classes*.

4.4 Statements and Expressions

Although the method declaration syntax of Java is quite different from that of C++, Java statement and expression syntax is very much like that of C. Again, the design intention was to make the low-level details of Java easily accessible to C programmers, so that they can concentrate on learning the parts of the language that are really different. Java *statements* appear inside of methods and class and variable initializers; they describe all activities of a Java program. Variable declarations and initializations like those in the previous section are statements, as are the basic language structures like conditionals and loops. *Expressions* are statements that produce a result that can be used as part of another statement. Method calls, object allocations, and, of course, mathematical expressions are examples of expressions.

One of the tenets of Java is to keep things simple and consistent. To that end, when there are no other constraints, evaluations and initializations in Java always occur in the order in which they appear in the code--from left to right. We'll see this rule used in the evaluation of assignment expressions, method calls, and array indexes, to name a few cases. In some other languages, the order of evaluation is more complicated or even implementation dependent. Java removes this element of danger by precisely and simply defining how the code is evaluated. This doesn't, however, mean you should start writing obscure and convoluted statements. Relying on the order of evaluation of expressions is a bad programming habit, even when it works. It produces code that is hard to read and harder to modify. Real programmers, however, are not made of stone, and you may catch me doing this once or twice when I can't resist the urge to write terse code.

Statements

As in C or C++, statements and expressions in Java appear within a *code block*. A code block is syntactically just a number of statements surrounded by an open curly brace (`{`) and a close curly brace (`}`). The statements in a code block can contain variable declarations:

```
{
    int size = 5;
    setName("Max");
    ...
}
```

Methods, which look like C functions, are in a sense code blocks that take parameters and can be called by name.

```
setupDog( String name ) {
    int size = 5;
    setName( name );
    ...
}
```

Variable declarations are limited in scope to their enclosing code block. That is, they can't be seen outside of the nearest set of braces:

```
{
    int i = 5;
}

i = 6;           // compile time error, no such variable i
```

In this way, code blocks can be used to arbitrarily group other statements and variables. The most common use of code blocks, however, is to define a group of statements for use in a conditional or iterative statement.

Since a code block is itself collectively treated as a statement, we define a conditional like an `if/else` clause as follows:

```
if ( condition )
    statement;
[ else
    statement; ]
```

Thus, `if/else` in Java has the familiar functionality of taking either of the forms:

```
if ( condition )
    statement;
```

or:

```
if ( condition )  {
    [ statement; ]
    [ statement; ]
    [ ... ]
}
```

Here the *condition* is a boolean expression. In the second form, the statement is a code block, and all of its enclosed statements are executed if the conditional succeeds. Any variables declared within that block are visible only to the statements within the successful branch of the condition. Like the if/else conditional, most of the remaining Java statements are concerned with controlling the flow of execution. They act for the most part like their namesakes in C or C++.

The do and while iterative statements have the familiar functionality, except that their conditional test is also a boolean expression. You can't use an integer expression or a reference type; in other words you must explicitly test your value. In other words, while i==0 is legitimate, i is not, unless i is boolean. Here are the forms of these two statements:

```
while ( conditional )
    statement;

do
    statement;
while ( conditional );
```

The for statement also looks like it does in C:

```
for ( initialization; conditional; incrementor )
    statement;
```

The variable initialization expression can declare a new variable; this variable is limited to the scope of the for statement:

```
for (int i = 0; i < 100; i++ ) {
    System.out.println( i )
    int j = i;
    ...
}
```

Java doesn't support the C comma operator, which groups multiple expressions into a single expression. However, you can use multiple, comma-separated expressions in the initialization and increment sections of the for loop. For example:

```
for (int i = 0, j = 10; i < j; i++, j-- ) {
   ... }
```

The Java `switch` statement takes an integer type (or an argument that can be promoted to an integer type) and selects among a number of alternative `case` branches[2] :

> [2] An object-based `switch` statement is desirable and could find its way into the language someday.

```
switch ( int expression ) {
    case int expression :
        statement;
    [ case int expression
        statement;
    ...
    default :
        statement;  ]
}
```

No two of the `case` expressions can evaluate to the same value. As in C, an optional `default` case can be specified to catch unmatched conditions. Normally, the special statement `break` is used to terminate a branch of the `switch`:

```
switch ( retVal ) {
    case myClass.GOOD :
        // something good
        break;
    case myClass.BAD :
        // something bad
        break;
    default :
        // neither one
        break;
}
```

The Java `break` statement and its friend `continue` perform unconditional jumps out of a loop or conditional statement. They differ from the corresponding statements in C by taking an optional label as an argument. Enclosing statements, like code blocks and iterators, can be labeled with identifier statements:

```
one:
    while ( condition ) {
        ...
        two:
            while ( condition ) {
                ...
                // break or continue point
            }
        // after two
    }
// after one
```

In the above example, a `break` or `continue` without argument at the indicated position would have the normal, C-style effect. A `break` would cause processing to resume at the point labeled "after two"; a `continue` would immediately cause the `two` loop to return to its condition test.

The statement `break two` at the indicated point would have the same effect as an ordinary `break`, but `break one` would break two levels and resume at the point labeled "after one." Similarly, `continue two` would serve as a normal `continue`, but `continue one` would return to the test of the `one` loop. Multilevel `break` and `continue` statements remove much of the need for the evil `goto` statement in C and C++.

There are a few Java statements we aren't going to discuss right now. The `try`, `catch`, and `finally` statements are used in exception handling, as we'll discuss later in this chapter. The `synchronized` statement in Java is used to coordinate access to statements among multiple threads of execution; see Chapter 6, *Threads* for a discussion of thread synchronization.

On a final note, I should mention that the Java compiler flags "unreachable" statements as compile-time errors. Of course, when I say unreachable, I mean those statements the compiler determines won't be called by a static look at compile-time.

Expressions

As I said earlier, expressions are statements that produce a result when they are evaluated. The value of an expression can be a numeric type, as in an arithmetic expression; a reference type, as in an object allocation; or the special type `void`, which results from a call to a method that doesn't return a value. In the last case, the expression is evaluated only for its side effects (i.e., the work it does aside from producing a value). The type of an expression is known at compile-time. The value produced at run-time is either of this type or, in the case of a reference type, a compatible (assignable) type.

Operators

Java supports almost all standard C operators. These operators also have the same precedence in Java as they do in C, as you can see in Table 4.3.

Table 4.3: Java Operators

Precedence	Operator	Operand Type	Description
1	++, --	Arithmetic	Increment and decrement
1	+, -	Arithmetic	Unary plus and minus
1	~	Integral	Bitwise complement
1	!	Boolean	Logical complement
1	(*type*)	Any	Cast
2	*, /, %	Arithmetic	Multiplication, division, remainder
3	+, -	Arithmetic	Addition and subtraction
3	+	String	String concatenation
4	<<	Integral	Left shift
4	>>	Integral	Right shift with sign extension
4	>>>	Integral	Right shift with no extension
5	<, <=, >, >=	Arithmetic	Numeric comparison
5	instanceof	Object	Type comparison
6	==, !=	Primitive	Equality and inequality of value
6	==, !=	Object	Equality and inequality of reference
7	&	Integral	Bitwise AND
7	&	Boolean	Boolean AND
8	^	Integral	Bitwise XOR
8	^	Boolean	Boolean XOR
9	|	Integral	Bitwise OR
9	|	Boolean	Boolean OR
10	&&	Boolean	Conditional AND
11	||	Boolean	Conditional OR
12	?:	NA	Conditional ternary operator
13	=	Any	Assignment
13	*=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, |=	Any	Assignment with operation

There are a few operators missing from the standard C collection. For example, Java doesn't support the comma operator for combining expressions, although the `for` statement allows you to use it in the initialization and increment sections. Java doesn't allow direct pointer manipulation, so it does not support the reference (`*`), dereference (`&`), and `sizeof` operators.

Java also adds some new operators. As we've seen, the + operator can be used with String values to perform string concatenation. Because all integral types in Java are signed values, the >> operator performs a right-shift operation with sign extension. The >>> operator treats the operand as an unsigned number and performs a right shift with no extension. The new operator is used to create objects; we will discuss it in detail shortly.

Assignment

While variable initialization (i.e., declaration and assignment together) is considered a statement, variable assignment alone is an expression:

```
int i, j;
i = 5;                           // expression
```

Normally, we rely on assignment for its side effects alone, but, as in C, an assignment can be used as a value in another part of an expression:

```
j = ( i = 5 );
```

Again, relying on order of evaluation extensively (in this case, using compound assignments in complex expressions) can make code very obscure and hard to read. Do so at your own peril.

null

The expression null can be assigned to any reference type. It has the meaning of "no reference." A null reference can't be used to select a method or variable and attempting to do so generates a NullPointerException at run-time.

Variable access

Using the dot (.) to access a variable in an object is a type of expression that results in the value of the variable accessed. This can be either a numeric type or a reference type:

```
int i;
String s;
i = myObject.length;
s = myObject.name;
```

A reference type expression can be used in further evaluations, by selecting variables or calling methods within it:

```
int len = myObject.name.length();
int initialLen = myObject.name.substring(5, 10).length();
```

Here we have found the length of our `name` variable by invoking the `length()` method of the `String` object. In the second case, we took an intermediate step and asked for a substring of the `name` string. The `substring` method of the `String` class also returns a `String` reference, for which we ask the length. (Chapter 7, *Basic Utility Classes* describes all of these `String` methods in detail.)

Method invocation

A method invocation is basically a function call, or in other words, an expression that results in a value, the type of which is the return type of the method. Thus far, we have seen methods invoked via their name in simple cases:

```
System.out.println( "Hello World..." );
int myLength = myString.length();
```

When we talk about Java's object-oriented features in Chapter 5, *Objects in Java*, we'll look at some rules that govern the selection of methods.

Like the result of any expression, the result of a method invocation can be used in further evaluations, as we saw above. Whether to allocate intermediate variables and make it absolutely clear what your code is doing or to opt for brevity where it's appropriate is a matter of coding style.

Object creation

Objects in Java are allocated with the `new` operator:

```
Object o = new Object();
```

The argument to `new` is a *constructor* that specifies the type of object and any required parameters to create it. The return type of the expression is a reference type for the created object.

We'll look at object creation in detail in Chapter 5, *Objects in Java*. For now, I just want to point out that object creation is a type of expression, and that the resulting object reference can be used in general expressions. In fact, because the binding of `new` is "tighter" than that of the dot-field selector, you can easily allocate a new object and invoke a method in it for the resulting expression:

```
int hours = new Date().getHours();
```

The `Date` class is a utility class that represents the current time. Here we create a new instance of `Date` with the `new` operator and call its `getHours()` method to retrieve the current hour as an integer value. The `Date` object reference lives long enough to service the method call and is then cut loose and garbage collected at some point in the future.

Calling methods in object references in this way is, again, a matter of style. It would certainly be clearer to allocate an intermediate variable of type `Date` to hold the new object and then call its `getHours()` method. However, some of us still find the need to be terse in our code.

instanceof

The `instanceof` operator can be used to determine the type of an object at run-time. `instanceof` returns a `boolean` value that indicates whether an object is an instance of a particular class or a subclass of that class:

```
Boolean b;
String str = "foo";
b = ( str instanceof String );     // true
b = ( str instanceof Object );     // also true
b = ( str instanceof Date );       // false--not a Date or subclass
```

`instanceof` also correctly reports if an object is of the type of an arry or a specified interface.

```
        if ( foo instanceof byte[] )
            ...
```

(See Chapter 5, *Objects in Java* for a full discussion of interfaces.)

It is also important to note that the value *null* is not considered an instance of any object. So the following test will return false, no matter what the declared type of the variable:

```
String s = null;
if ( s istanceof String )
    // won't happen
```

PREVIOUS | HOME | NEXT
Types | BOOK INDEX | Exceptions

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

4.5 Exceptions

> Do, or do not... There is no try.
>
> --Yoda (*The Empire Strikes Back*)

Java's roots are in embedded systems--software that runs inside specialized devices like hand-held computers, cellular phones, and fancy toasters. In those kinds of applications, it's especially important that software errors be handled properly. Most users would agree that it's unacceptable for their phone to simply crash or for their toast (and perhaps their house) to burn because their software failed. Given that we can't eliminate the possibility of software errors, a step in the right direction is to at least try to recognize and deal with the application-level errors that we can anticipate in a methodical and systematic way.

Dealing with errors in a language like C is the responsibility of the programmer. There is no help from the language itself in identifying error types, and there are no tools for dealing with them easily. In C and C++, a routine generally indicates a failure by returning an "unreasonable" value (e.g., the idiomatic `-1` or `null`). As the programmer, you must know what constitutes a bad result, and what it means. It's often awkward to work around the limitations of passing error values in the normal path of data flow.[3] An even worse problem is that certain types of errors can legitimately occur almost anywhere, and it's prohibitive and unreasonable to explicitly test for them at every point in the software.

> [3] The somewhat obscure `setjmp()` and `longjmp()` statements in C can save a point in the execution of code and later return to it unconditionally from a deeply buried location. In a limited sense, this is the functionality of exceptions in Java.

Java offers an elegant solution to these problems with exception handling. (Java exception handling is similar to, but not quite the same as, exception handling in C++.) An *exception* indicates an unusual condition or an error condition. Program control becomes unconditionally transferred or thrown to a specially designated section of code where it's caught and handled. In this way, error handling is somewhat orthogonal to the normal flow of the program. We don't have to have special return values for all our methods; errors are handled by a separate mechanism. Control can be passed long distance from a deeply nested routine and handled in a single location when that is desirable, or an error can be handled immediately at its source. There are still a few methods that return `-1` as a special value, but these are limited to situations in which there isn't really an error.[4]

> [4] For example, the `getHeight()` method of the `Image` class returns `-1` if the height isn't known yet. No error has occurred; the height will be available in the future. In this situation, throwing an exception would be inappropriate.

A Java method is required to specify the exceptions it can throw (i.e., the ones that it doesn't catch itself); this means that the compiler can make sure we handle them. In this way, the information about what errors a method can produce is promoted to the same level of importance as its argument and return types. You may still decide to punt and ignore obvious errors, but in Java you must do so explicitly.

Exceptions and Error Classes

Exceptions are represented by instances of the class `java.lang.Exception` and its subclasses. Subclasses of `Exception` can hold specialized information (and possibly behavior) for different kinds of exceptional conditions. However, more often they are simply "logical" subclasses that exist only to serve as a new exception type (more on that later). Figure 4.1 shows the subclasses of `Exception`; these classes are defined in various packages in the Java API, as indicated in the diagram.

Figure 4.1: Java exception classes

[Graphic: Figure 4-1]

An `Exception` object is created by the code at the point where the error condition arises. It can hold whatever information is necessary to describe the exceptional condition, including a full stack trace for debugging. The exception object is passed, along with the flow of control, to the handling block of code. This is where the terms "throw" and "catch" come from: the `Exception` object is thrown from one point in the code and caught by the other, where execution resumes.

The Java API also defines the `java.lang.Error` class for eggregious or unrecoverable errors. The subclasses of `Error` are shown in Figure 4.2. You needn't worry about these errors (i.e., you do not have to catch them); they normally indicate linkage problems or virtual machine errors. An error of this kind usually causes the Java interpreter to display a message and exit.

Figure 4.2: Java error classes

[Graphic: Figure 4-2]

Exception Handling

The `try/catch` guarding statements wrap a block of code and catch designated types of exceptions that occur within it:

```
try {
    readFromFile("foo");
    ...
}
```

```
catch ( Exception e ) {
    // Handle error
    System.out.println( "Exception while reading file: " + e );
    ...
}
```

In the above example, exceptions that occur within the body of the `try` statement are directed to the `catch` clause for possible handling. The `catch` clause acts like a method; it specifies an argument of the type of exception it wants to handle, and, if it's invoked, the `Exception` object is passed into its body as an argument. Here we receive the object in the variable `e` and print it along with a message.

A `try` statement can have multiple `catch` clauses that specify different specific types (subclasses) of `Exception`:

```
try {
    readFromFile("foo");
    ...
}
catch ( FileNotFoundException e ) {
    // Handle file not found
    ...
}
catch ( IOException e ) {
    // Handle read error
    ...
}
catch ( Exception e ) {
    // Handle all other errors
    ...
}
```

The `catch` clauses are evaluated in order, and the first possible (assignable) match is taken. At most one `catch` clause is executed, which means that the exceptions should be listed from most specific to least. In the above example, we'll assume that the hypothetical `readFromFile()` can throw two different kinds of exceptions: one that indicates the file is not found; the other indicates a more general read error. Any subclass of `Exception` is assignable to the parent type `Exception`, so the third `catch` clause acts like the `default` clause in a `switch` statement and handles any remaining possibilities.

It should be obvious, but one beauty of the `try/catch` statement is that any statement in the `try` block can assume that all previous statements in the block succeeded. A problem won't arise suddenly because a programmer forgot to check the return value from some method. If an earlier statement fails, execution jumps immediately to the `catch` clause; later statements are never executed.

Bubbling Up

What if we hadn't caught the exception? Where would it have gone? Well, if there is no enclosing `try/catch` statement, the exception pops to the top of the method in which it appeared and is, in turn, thrown from that method. In this way, the exception bubbles up until it's caught, or until it pops out of the top of the program, terminating it with a run-time error message. There's a bit more to it than that because, in this case, the compiler would have reminded us to deal with it, but we'll get back to that in a moment.

Let's look at another example. In Figure 4.3, the method `getContent()` invokes the method `openConnection()` from within a `try/catch` statement. `openConnection()`, in turn, invokes the method `sendRequest()`, which calls the method `write()` to send some data.

Figure 4.3: Exception propagation

[Graphic: Figure 4-3]

In this figure, the second call to `write()` throws an `IOException`. Since `sendRequest()` doesn't contain a `try/catch` statement to handle the exception, it's thrown again, from the point that it was called in the method `openConnection()`. Since `openConnection()` doesn't catch the exception either, it's thrown once more. Finally it's caught by the `try` statement in `getContent()` and handled by its `catch` clause.

Since an exception can bubble up quite a distance before it is caught and handled, we may need a way to determine exactly where it was thrown. All exceptions can dump a *stack trace* that lists their method of origin and all of the nested method calls that it took to arrive there, using the printStackTrace() method.

```
try {
    // complex task
} catch ( Exception e ) {
    // dump information about exactly where the exception ocurred
    e.printStackTrack( System.err );
    ...
}
```

The *throws* Clause and *checked* Exceptions

I mentioned earlier that Java makes us be explicit about our error handling. But Java is programmer-friendly, and it's not possible to require that every conceivable type of error be handled in every situation. So, Java exceptions are divided into two categories: *checked exceptions* and *unchecked exceptions*. Most application level exceptions are checked, which means that any method that throws one, either by generating it itself (as we'll discuss below) or by passively ignoring one that occurs within it, must declare that it can throw that type of exception in a special `throws` clause in its method declaration. We haven't yet talked in detail about declaring methods; we'll cover that in Chapter 5, *Objects in Java*. For now all you need know is that methods have to declare the checked exceptions they can throw or allow to be thrown.

Again in Figure 4.3, notice that the methods `openConnection()` and `sendRequest()` both specify that they can throw an `IOException`. If we had to throw multiple types of exceptions we could declare them separated with commas:

```
void readFile( String s ) throws IOException, InterruptedException {
    ...
}
```

The throws clause tells the compiler that a method is a possible source of that type of checked exception and that anyone calling that method must be prepared to deal with it. The caller may use a *try/catch* block to catch it, or it may, itself, declare that it can throw the exception.

Exceptions that are subclasses of the `java.lang.RuntimeException` class are unchecked. See Figure 4.1 for the subclasses of `RuntimeException`. It's not a compile-time error to ignore the possibility of these exceptions being thrown; additionally, methods don't have to declare they can throw them. In all other respects, run-time exceptions behave the same as other exceptions. We are perfectly free to catch them if we wish; we simply aren't required to.

Checked exceptions

> Exceptions a reasonable application should try to handle gracefully.

Unchecked exception (Runtime exceptions)

> Exceptions from which we would not normally expect our software to try to recover.

The category of checked exceptions includes application-level problems like missing files and unavailable hosts. As good programmers (and upstanding citizens), we should design software to recover gracefully from these kinds of conditions. The category of unchecked exceptions includes problems such as "out of memory" and "array index out of bounds." While these may indicate application-level programming errors, they can occur almost anywhere and aren't generally easy to recover from. Fortunately, because there are unchecked exceptions, you don't have to wrap every one of your array-index operations in a `try/catch` statement.

Throwing Exceptions

We can throw our own exceptions: either instances of `Exception` or one of its predefined subclasses, or our own specialized subclasses. All we have to do is create an instance of the `Exception` and throw it with the `throw` statement:

```
throw new Exception();
```

Execution stops and is transferred to the nearest enclosing `try/catch` statement. (Note that there is little point in keeping a reference to the `Exception` object we've created here.) An alternative constructor of the `Exception` class lets us specify a string with an error message:

```
throw new Exception("Something really bad happened");
```

By convention, all types of Exception have a String constructor like this. Note that the String message above is somewhat facetious and vague. Normally you won't be throwing a plain old Exception, but a more specific subclass. For example:

```
public void checkRead( String s ) {
    if ( new File(s).isAbsolute() || (s.indexOf("..") != -1) )
        throw new SecurityException(
           x"Access to file : "+ s +" denied.");
}
```

In the above, we partially implement a method to check for an illegal path. If we find one, we throw a SecurityException, with some information about the transgression.

Of course, we could include whatever other information is useful in our own specialized subclasses of `Exception` (or SecurityException). Often though, just having a new type of exception is good enough, because it's sufficient to help direct the flow of control. For example, if we are building a parser, we might want to make our own kind of exception to indicate a particular kind of failure.

```
class ParseException extends Exception {
    ParseException() {
        super(); }
    ParseException( String desc ) {
        super( desc ) };
}
```

See Chapter 5, *Objects in Java* for a full description of classes and class constructors. The body of our exception class here simply allows a ParseException to be created in the conventional ways that we have created exceptions above. Now that we have our new exception type, we we might guard for it in the following kind of situation:

```
// Somewhere in our code
...
try {
    parseStream( input );
} catch ( ParseException pe ) {
    // Bad input...
} catch ( IOException ioe ) {
    // Low level communications problem
}
```

As you can see, although our new exception doesn't currently hold any specialized information about the problem (it certainly could), it does let us distinguish a parse error from an arbitrary communications error in the same chunk of code. You might call this kind of specialization of an exception to be making a "logical" exception.

Re-throwing exceptions

Sometimes you'll want to take some action based on an exception and then turn around and throw a new exception in its place. For example, suppose that we want to handle an IOException by freeing up some resources before allowing the failure to pass on to the rest of the application. You can do this in the obvious way, by simply catching the exception and then throwing it again or throwing a new one.

```
  *** I was going to say something about fillInStackTrack() here ***
```

Try Creep

The `try` statement imposes a condition on the statements they guard. It says that if an exception occurs within it, the remaining statements will be abandoned. This has consequences for local variable initialization. If the compiler can't determine whether a local variable assignment we placed inside a `try/catch` block will happen, it won't let us use the variable:

```
void myMethod() {
    int foo;

    try {
        foo = getResults();
    }
    catch ( Exception e ) {
        ...
    }

    int bar = foo;  // Compile time error--foo may not
                    // have been initialized
```

In the above example, we can't use `foo` in the indicated place because there's a chance it was never assigned a value. One obvious option is to move the assignment inside the `try` statement:

```
try {
    foo = getResults();
    int bar = foo;  // Okay because we only get here
                    // if previous assignment succeeds
}
catch ( Exception e ) {
    ...
}
```

Sometimes this works just fine. However, now we have the same problem if we want to use `bar` later in `myMethod()`. If we're not careful, we might end up pulling everything into the `try` statement. The situation changes if we transfer control out of the method in the `catch` clause:

```
try {
    foo = getResults();
}
catch ( Exception e ) {
    ...
    return;
}

int bar = foo;  // Okay because we only get here
                // if previous assignment succeeds
```

Your code will dictate its own needs; you should just be aware of the options.

The finally Clause

What if we have some clean up to do before we exit our method from one of the `catch` clauses? To avoid duplicating the code in each `catch` branch and to make the cleanup more explicit, Java supplies the `finally` clause. A `finally` clause can be added after a `try` and any associated `catch` clauses. Any statements in the body of the `finally` clause are guaranteed to be executed, no matter why control leaves the `try` body:

```
try {
    // Do something here
}
catch ( FileNotFoundException e ) {
    ...
}
catch ( IOException e ) {
    ...
}
catch ( Exception e ) {
    ...
}
finally {
    // Cleanup here
}
```

In the above example the statements at the cleanup point will be executed eventually, no matter how control leaves the `try`. If control transfers to one of the `catch` clauses, the statements in `finally` are executed after the `catch` completes. If none of the `catch` clauses handles the exception, the `finally` statements are executed before the exception propagates to the next level.

If the statements in the `try` execute cleanly, or even if we perform a `return`, `break`, or `continue`, the statements in the `finally` clause are executed. To perform cleanup operations, we can even use `try` and `finally` without any `catch` clauses:

```
try {
    // Do something here
    return;
}
finally {
    System.out.println("Whoo-hoo!");
}
```

Exceptions that occur in a `catch` or `finally` clause are handled normally; the search for an enclosing `try/catch` begins outside the offending `try` statement.

PREVIOUS
Statements and Expressions

HOME
BOOK INDEX

NEXT
Arrays

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

4.6 Arrays

An array is a special type of object that can hold an ordered collection of elements. The type of the elements of the array is called the *base type* of the array; the number of elements it holds is a fixed attribute called its *length*. (For a collection with a variable length, see the discussion of `Vector` objects in Chapter 7, *Basic Utility Classes*.) Java supports arrays of all numeric and reference types.

The basic syntax of arrays looks much like that of C or C++. We create an array of a specified length and access the elements with the special index operator, `[]`. Unlike other languages, however, arrays in Java are true, first-class objects, which means they are real objects within the Java language. An array is an instance of a special Java array class and has a corresponding type in the type system. This means that to use an array, as with any other object, we first declare a variable of the appropriate type and then use the `new` operator to create an instance of it.

Array objects differ from other objects in Java in three respects:

- Java implicitly creates a special array class for us whenever we declare an array type variable. It's not strictly necessary to know about this process in order to use arrays, but it helps in understanding their structure and their relationship to other objects in Java.

- Java lets us use the special `[]` operator to access array elements, so that arrays look as we expect. We could implement our own classes that act like arrays, but because Java doesn't have user-defined operator overloading, we would have to settle for having methods like `get()` and `put()` instead of using the special `[]` notation.

- Java provides a corresponding special form of the `new` operator that lets us construct an instance of an array and specify its length with the `[]` notation.

Array Types

An array type variable is denoted by a base type followed by empty brackets `[]`. Alternatively, Java accepts a C-style declaration, with the brackets placed after the array name. The following are equivalent:

```
int [] arrayOfInts;
int arrayOfInts [];
```

In each case, `arrayOfInts` is declared as an array of integers. The size of the array is not yet an issue, because we are declaring only the array type variable. We have not yet created an actual instance of the array class, with its associated storage. It's not even possible to specify the length of an array as part of its type.

An array of objects can be created in the same way:

```
String [] someStrings;
Button someButtons [];
```

Array Creation and Initialization

Having declared an array type variable, we can now use the `new` operator to create an instance of the array. After the `new` operator, we specify the base type of the array and its length, with a bracketed integer expression:

```
arrayOfInts = new int [42];
someStrings = new String [ number + 2 ];
```

We can, of course, combine the steps of declaring and allocating the array:

```
double [] someNumbers = new double [20];
Component widgets [] = new Component [12];
```

As in C, array indices start with zero. Thus, the first element of `someNumbers []` is `0` and the last element is `19`. After creation, the array elements are initialized to the default values for their type. For numeric types, this means the elements are initially zero:

```
int [] grades = new int [30];
grades[0] = 99;
grades[1] = 72;
// grades[2] == 0
```

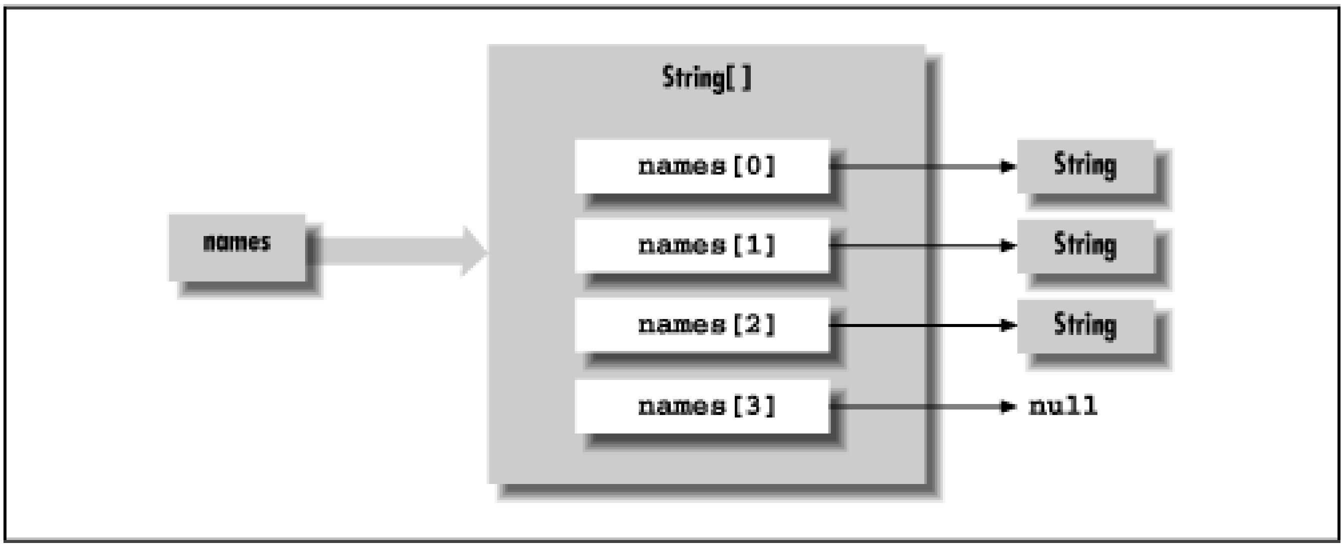
The elements of an array of objects are references to the objects, not actual instances of the objects. The default value of each element is therefore `null`, until we assign instances of appropriate objects:

```
String names [] = new String [4];
names [0] = new String();
names [1] = "Boofa";
names [2] = someObject.toString();
// names[3] == null
```

This is an important distinction that can cause confusion. In many other languages, the act of creating an array is the same as allocating storage for its elements. In Java, an array of objects actually contains only reference variables and those variables, have the value `null` until they are assigned to real objects.[5] Figure 4.4 illustrates the `names` array of the previous example:

> [5] The analog in C or C++ would be an array of pointers to objects. However, pointers in C or C++ are themselves two- or four-byte values. Allocating an array of pointers is, in actuality, allocating the storage for some number of those pointer objects. An array of references is conceptually similar, although references are not themselves objects. We can't manipulate references or parts of references other than by assignment, and their storage requirements (or lack thereof) are not part of the high-level language specification.

Figure 4.4: The names array



`names` is a variable of type `String[]` (i.e., a string array). The `String[]` object can be thought of as containing four `String` type variables. We have assigned `String` objects to the first three array elements. The fourth has the default value `null`.

Java supports the C-style curly braces `{}` construct for creating an array and initializing its elements when it is declared:

```
int [] primes = { 1, 2, 3, 5, 7, 7+4 };    // primes[2] == 3
```

An array object of the proper type and length is implicitly created and the values of the comma-separated list of expressions are assigned to its elements.

We can use the {} syntax with an array of objects. In this case, each of the expressions must evaluate to an object that can be assigned to a variable of the base type of the array, or the value null. Here are some examples:

```
String [] verbs = { "run", "jump", someWord.toString() };
Button [] controls = { stopButton, new Button("Forwards"),
    new Button("Backwards") };
// all types are subtypes of Object
Object [] objects = { stopButton, "A word", null };
```

You should create and initialize arrays in whatever manner is appropriate for your application. The following are equivalent:

```
Button [] threeButtons = new Button [3];
Button [] threeButtons = { null, null, null };
```

Using Arrays

The size of an array object is available in the public variable length:

```
char [] alphabet = new char [26];
int alphaLen = alphabet.length;             // alphaLen == 26

String [] musketeers = { "one", "two", "three" };
int num = musketeers.length;                // num == 3
```

length is the only accessible field of an array; it is a variable, not a method.

Array access in Java is just like array access in C; you access an element by putting an integer-valued expression between brackets after the name of the array. The following example creates an array of Button objects called keyPad and then fills the array with Button objects:

```
Button [] keyPad = new Button [ 10 ];
for ( int i=0; i < keyPad.length; i++ )      keyPad[ i ] = new Button(
Integer.toString( i ) );
```

Attempting to access an element that is outside the range of the array generates an ArrayIndexOutOfBoundsException. This is a type of RuntimeException, so you can either catch it and handle it yourself, or ignore it, as we already discussed:

```
String [] states = new String [50];

try {
    states[0] = "California";
    states[1] = "Oregon";
    ...
    states[50] = "McDonald's Land";  // Error--array out of bounds
}
catch ( ArrayIndexOutOfBoundsException err ) {
    System.out.println( "Handled error: " + err.getMessage() );
}
```

It's a common task to copy a range of elements from one array into another. Java supplies the `arraycopy()` method for this purpose; it's a utility method of the `System` class:

```
System.arraycopy( source, sourceStart, destination,
                  destStart, length );
```

The following example doubles the size of the `names` array from an earlier example:

```
String [] tmpVar = new String [ 2 * names.length ];
System.arraycopy( names, 0, tmpVar, 0, names.length );
names = tmpVar;
```

A new array, twice the size of `names`, is allocated and assigned to a temporary variable `tmpVar`. `arraycopy()` is used to copy the elements of `names` to the new array. Finally, the new array is assigned to `names`. If there are no remaining references to the old array object after `names` has been copied, it will be garbage collected on the next pass.

Anonymous Arrays

You often want to create "throw-away" arrays: arrays that are only used in one place, and never referenced anywhere else. Such arrays don't need to have a name, because you never need to refer to them again in that context. For example, you may want to create a collection of objects to pass as an argument to some method. It's easy enough to create a normal, named array--but if you don't actually work with the array (if you use the array only as a holder for some collection), you shouldn't have to. Java makes it easy to create "anonymous" (i.e., unnamed) arrays.

Let's say you need to call a method named `setPets()`, which takes an array of `Animal` objects as arguments. `Cat` and `Dog` are subclasses of `Animal`. Here's how to call `setPets()` using an anonymous array:

```
Dog pokey = new Dog ("gray");
Cat squiggles = new Cat ("black");
Cat jasmine = new Cat ("orange");
setPets ( new Animal [] { pokey, squiggles, jasmine });
```

The syntax looks just like the initialization of an array in a variable declaration. We implicitly define the size of the array and fill in its elements using the curly brace notation. However, since this is not a variable declaration we have to explicitly use the new operator to create the array object.

You can use anonymous arrays to simulate variable length argument lists (often called VARARGS), a feature of many programming languages that Java doesn't provide. The advantage of anonymous arrays over variable length argument lists is that it allows stricter type checking; the compiler always knows exactly what arguments are expected, and therefore can verify that method calls are correct.

Multidimensional Arrays

Java supports multidimensional arrays in the form of arrays of array type objects. You create a multidimensional array with C-like syntax, using multiple bracket pairs, one for each dimension. You also use this syntax to access elements at various positions within the array. Here's an example of a multidimensional array that represents a chess board:

```
ChessPiece [][] chessBoard;
chessBoard = new ChessPiece [8][8];
chessBoard[0][0] = new ChessPiece( "Rook" );
chessBoard[1][0] = new ChessPiece( "Pawn" );
...
```

Here `chessBoard` is declared as a variable of type `ChessPiece[][]` (i.e., an array of `ChessPiece` arrays). This declaration implicitly creates the type `ChessPiece[]` as well. The example illustrates the special form of the `new` operator used to create a multidimensional array. It creates an array of `ChessPiece[]` objects and then, in turn, creates each array of `ChessPiece` objects. We then index `chessBoard` to specify values for particular `ChessPiece` elements. (We'll neglect the color of the pieces here.)

Of course, you can create arrays of with more than two dimensions. Here's a slightly impractical example:

```
Color [][][] rgbCube = new Color [256][256][256];
rgbCube[0][0][0] = Color.black;
rgbCube[255][255][0] = Color.yellow;
```

```
...
```

As in C, we can specify the initial index of a multidimensional array to get an array type object with fewer dimensions. In our example, the variable `chessBoard` is of type `ChessPiece[][]`. The expression `chessBoard[0]` is valid and refers to the first element of `chessBoard`, which is of type `ChessPiece[]`. For example, we can create a row for our chess board:

```
ChessPiece [] startRow =  {
    new ChessPiece("Rook"), new ChessPiece("Knight"),
    new ChessPiece("Bishop"), new ChessPiece("King"),
    new ChessPiece("Queen"), new ChessPiece("Bishop"),
    new ChessPiece("Knight"), new ChessPiece("Rook")
};

chessBoard[0] = startRow;
```

We don't necessarily have to specify the dimension sizes of a multidimensional array with a single `new` operation. The syntax of the `new` operator lets us leave the sizes of some dimensions unspecified. The size of at least the first dimension (the most significant dimension of the array) has to be specified, but the sizes of any number of the less significant array dimensions may be left undefined. We can assign appropriate array type values later.

We can create a checkerboard of `boolean` values (which is not quite sufficient for a real game of checkers) using this technique:

```
boolean [][] checkerBoard;
checkerBoard = new boolean [8][];
```

Here, `checkerBoard` is declared and created, but its elements, the eight `boolean[]` objects of the next level, are left empty. Thus, for example, `checkerBoard[0]` is `null` until we explicitly create an array and assign it, as follows:

```
checkerBoard[0] = new boolean [8];
checkerBoard[1] = new boolean [8];
...
checkerBoard[7] = new boolean [8];
```

The code of the previous two examples is equivalent to:

```
boolean [][] checkerBoard = new boolean [8][8];
```

One reason we might want to leave dimensions of an array unspecified is so that we can store arrays given to us by another method.

Note that since the length of the array is not part of its type, the arrays in the checkerboard do not necessarily have to be of the same length. Here's a defective (but perfectly legal) checkerboard:

```
checkerBoard[2] = new boolean [3];
checkerBoard[3] = new boolean [10];
```

Since Java implements multidimensional arrays as arrays of arrays, multidimensional arrays do not have to be rectangular. For example, here's how you could create and initialize a triangular array:

```
int [][][] triangle = new int [5][];
for (int i = 0; i < triangle.length; i++) {
    triangle[i] = new int [i + 1];
    for (int j = 0; j < i + 1; j++)
        triangle[i][j] = i + j;    }
```

Inside Arrays

I said earlier that arrays are instances of special array classes in the Java language. If arrays have classes, where do they fit into the class hierarchy and how are they related? These are good questions; however, we need to talk more about the object-oriented aspects of Java before I can answer them. For now, take it on faith that arrays fit into the class hierarchy; details are in Chapter 5, *Objects in Java*.

PREVIOUS | HOME | NEXT
Exceptions | BOOK INDEX | Objects in Java

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

5.2 Methods

Methods appear inside class bodies. They contain local variable declarations and other Java statements that are executed by a calling thread when the method is invoked. Method declarations in Java look like ANSI C-style function declarations with two restrictions:

- A method in Java always specifies a return type (there's no default). The returned value can be a primitive numeric type, a reference type, or the type `void`, which indicates no returned value.
- A method always has a fixed number of arguments. The combination of method overloading and true arrays removes most of the need for a variable number of arguments. These techniques are type-safe and easier to use than C's variable argument list mechanism.

Here's a simple example:

```
class Bird {
    int xPos, yPos;

    double fly ( int x, int y ) {
        double distance = Math.sqrt( x*x + y*y );
        flap( distance );
        xPos = x;
        yPos = y;
        return distance;
    }
    ...
}
```

In this example, the class `Bird` defines a method, `fly()`, that takes as arguments two integers: `x` and `y`. It returns a `double` type value as a result.

Local Variables

The `fly()` method declares a local variable called `distance` that it uses to compute the distance flown. A local variable is temporary; it exists only within the scope of its method. Local variables are allocated and initialized when a method is invoked; they are normally destroyed when the method returns. They can't be referenced from outside the method itself. If the method is executing concurrently in different threads, each thread has its own copies of the method's local variables. A method's arguments also serve as local variables within the scope of the method.

An object created within a method and assigned to a local variable may or may not persist after the method has returned. As with all objects in Java, it depends on whether any references to the object remain. If an object is created, assigned to a local variable, and never used anywhere else, that object will no longer be referenced when the local variable is destroyed, so garbage collection will remove the object. If, however, we assign the object to an instance variable, pass it as an argument to another method, or pass it back as a return value, it may be saved by another variable holding its reference. We'll discuss object creation and garbage collection in more detail shortly.

Shadowing

If a local variable and an instance variable have the same name, the local variable *shadows* or hides the name of the instance variable within the scope of the method. In the following example, the local variables `xPos` and `yPos` hide the instance variables of the same name:

```
class Bird {
    int xPos, yPos;
    int xNest, yNest;
    ...
    double flyToNest() {
        int xPos = xNest;
        int yPos = yNest:
        return ( fly( xPos, yPos ) );
    }
    ...
}
```

When we set the values of the local variables in `flyToNest()`, it has no effect on the values of the instance variables.

this

The special reference `this` refers to the current object. You can use it any time you need to refer explicitly to the current object instance. Often, you don't need to use `this` because the reference to the current object is implicit; this is the case with using instance variables and methods inside of a class. But we can use `this` to refer explicitly to instance variables in the object, even if they are shadowed.

The subsequent example shows how we can use `this` to allow us argument names that shadow instance variable names. This is a fairly common technique, as it saves your having to deliberately make up alternate names (as we'll try to emphasize in this book, names are important). Here's how we could implement our `fly()` method with shadowed variables:

```
class Bird {
    int xPos, yPos;

    double fly ( int xPos, int yPos ) {
        double distance = Math.sqrt( xPos*xPos + yPos*yPos );
        flap( distance );
        this.xPos = xPos;
        this.yPos = yPos;
        return distance;
    }
    ...
}
```

In this example, the expression `this.xPos` refers to the instance variable `xPos` and assigns it the value of the local variable `xPos`, which would otherwise hide its name. The only reason we need to use `this` in the above example is because we've used argument names that hide our instance variables, and we want to refer to the instance variables.

Static Methods

Static methods (class methods), like static variables, belong to the class and not to an individual instance of the class. What does this mean? Well, foremost, a static method lives outside of any particular class instance. It can be invoked by name, through the class name, without any objects around. Because it is not bound to a particular object instance, a static method can only directly access other static members of classes. It can't directly see any instance variables or call any instance methods, because to do so we'd have to ask: "on which instance?" Static methods can be called from instances, just like instance methods, but the important thing is that they can also be used independently.

Our `fly()` method uses a static method: `Math.sqrt()`. This method is defined by the `java.lang.Math` class; we'll explore this class in detail in Chapter 7, *Basic Utility Classes*. For now, the important thing to note is that `Math` is the name of a class and not an instance of a `Math` object (you can't even make an instance of `Math`). Because `static` methods can be invoked wherever the class name is available, class methods are closer to normal C-style functions. Static methods are particularly useful for utility methods that perform work that might be useful either independently of instances of the class or in creating instances of the class.

For example, in our `Bird` class we can enumerate all types of birds that can be created:

```
class Bird {
    ...
    static String [] getBirdTypes( ) {
    String [] types;
    // Create list...
        return types;
    }
    ...
}
```

Here we've defined a `static` method `getBirdTypes()` that returns an array of strings containing bird names. We can use `getBirdTypes()` from within an instance of `Bird`, just like an instance method. However, we can also call it from other classes, using the `Bird` class name as a reference:

```
String [] names = Bird.getBirdTypes();
```

Perhaps a special version of the `Bird` class constructor accepts the name of a bird type. We could use this list to decide what kind of bird to create.

Local Variable Initialization

In the `flyToNest()` example, we made a point of initializing the local variables `xPos` and `yPos`. Unlike instance variables, local variables must be initialized before they can be used. It's a compile-time error to try to access a local variable without first assigning it a value:

```
void myMethod() {
    int foo = 42;
    int bar;

    // bar += 1;       // Compile time error, bar uninitialized

    bar = 99;
    bar += 1;       // ok here
}
```

Notice that this doesn't imply local variables have to be initialized when declared, just that the first time they are referenced must be in an assignment. More subtle possibilities arise when making assignments inside of conditionals:

```
void myMethod {
    int foo;

    if ( someCondition ) {
        foo = 42;
        ...
    }

    foo += 1;                  // Compile time error
                               // foo may not have been initialized
```

In the above example, `foo` is initialized only if `someCondition` is `true`. The compiler doesn't let you make this wager, so it flags the use of `foo` as an error. We could correct this situation in several ways. We could initialize the variable to a default value in advance or move the usage inside of the conditional. We could also make sure the path of execution doesn't reach the uninitialized variable through some other means, depending on what makes sense for our particular application. For example, we could return from the method abruptly:

```
int foo;
...
if ( someCondition ) {
    foo = 42;
    ...
} else
    return;

foo += 1;
```

In this case, there's no chance of reaching `foo` in an unused state and the compiler allows the use of `foo` after the conditional.

Why is Java so picky about local variables? One of the most common (and insidious) sources of error in C or C++ is forgetting to initialize local variables, so Java tries to help us out. If it didn't, Java would suffer the same potential irregularities as C or C++.[2]

[2] As with `malloc`'ed storage in C or C++, Java objects and their instance variables are allocated on a heap, which allows them default values once, when they are created. Local variables, however, are allocated on the Java virtual machine stack. As with the stack in C and C++, failing to initialize these could mean successive method calls could receive garbage values, and program execution might be inconsistent or implementation dependent.

Argument Passing and References

Let's consider what happens when you pass arguments to a method. All primitive data types (e.g., `int`, `char`, `float`) are passed by value. Now you're probably used to the idea that reference types (i.e., any kind of object, including arrays and strings) are used through references. An important distinction (that we discussed briefly in Chapter 4) is that the references themselves (the pointers to these objects) are actually primitive types, and are passed by value too.

Consider the following piece of code:

```
// somewhere
    int i = 0;
    SomeKindOfObject obj = new SomeKindOfObject();
    myMethod( i, obj );
    ...
void myMethod(int j, SomeKindOfObject o) {
    ...
}
```

The first chunk of code calls `myMethod()`, passing it two arguments. The first argument, `i`, is passed by value; when the method is called, the value of `i` is copied into the method's parameter `j`. If `myMethod()` changes the value of `i`, it's changing only its copy of the local variable.

In the same way, a copy of the reference to `obj` is placed into the reference variable `o` of `myMethod()`. Both references refer to the same object, of course, and any changes made through either reference affect the actual (single) object instance, but there are two copies of the pointer. If we change the value of, say, `o.size`, the change is visible through either reference. However, if `myMethod()` changes the reference `o` itself--to point to another object--it's affecting only its copy. In this sense, passing the reference is like passing a pointer in C and *unlike* passing by reference in C++.

What if `myMethod()` needs to modify the calling method's notion of the `obj` reference as well (i.e., make `obj` point to a different object)? The easy way to do that is to wrap `obj` inside some kind of object. A good candidate would be to wrap the object up as the lone element in an array:

```
SomeKindOfObject [] wrapper = { obj };
```

All parties could then refer to the object as wrapper[0] and would have the ability to change the reference. This is not very asthetically pleasing, but it does illustrate that what is needed is the level of indirection. Another possibility is to use `this` to pass a reference to the calling object.

Let's look at another piece of code that could be from an implementation of a linked list:

```
class Element {
    public Element nextElement;

    void addToList( List list ) {
        list.addToList( this );
    }
}

class List {
    void addToList( Element element ) {
        ...
        element.nextElement = getNextElement();
    }
}
```

Every element in a linked list contains a pointer to the next element in the list. In this code, the `Element` class represents one element; it includes a method for adding itself to the list. The `List` class itself contains a method for adding an arbitrary `Element` to the list. The method `addToList()` calls `addToList()` with the argument `this` (which is, of course, an `Element`). `addToList()` can use the `this` reference to modify the `Element`'s `nextElement` instance variable. The same technique can be used in conjunction with interfaces to implement callbacks for arbitrary method invocations.

Method Overloading

Method overloading is the ability to define multiple methods with the same name in a class; when the method is invoked, the compiler picks the correct one based on the arguments passed to the method. This implies, of course, that overloaded methods must have different numbers or types of arguments. In a later section we'll look at method overriding, which occurs when we declare methods with identical signatures in different classes.

Method overloading is a powerful and useful feature. It's another form of polymorphism (ad-hoc polymorphism). The idea is to create methods that act in the same way on different types of arguments and have what appears to be a single method that operates on any of the types. The Java `PrintStream`'s `print()` method is a good example of method overloading in action. As you've probably deduced by now, you can print a string representation of just about anything using the expression:

```
System.out.print( argument )
```

The variable `out` is a reference to an object (a `PrintStream`) that defines nine different versions of the `print()` method. They take, respectively, arguments of the following types: `Object`, `String`, char[], char, int, long, float, double, and boolean.

```
class PrintStream {
    void print( Object arg ) { ... }
    void print( String arg ) { ... }
    void print( char [] arg ) { ... }
    ...
}
```

You can invoke the print() method with any of these types as an argument, and it's printed in an appropriate way. In a language without method overloading, this would require something more cumbersome, such as a separate method for printing each type of object. Then it would be your responsibility to remember what method to use for each data type.

In the above example, print() has been overloaded to support two reference types: Object and String. What if we try to call print() with some other reference type? Say, perhaps, a Date object? The answer is that since Date is a subclass of Object, the Object method is selected. When there's not an exact type match, the compiler searches for an acceptable, assignable match. Since Date, like all classes, is a subclass of Object, a Date object can be assigned to a variable of type Object. It's therefore an acceptable match, and the Object method is selected.

But what if there's more than one possible match? Say, for example, we tried to print a subclass of String called MyString. (Of course, the String class is final, so it can't be subclassed, but allow me this brief transgression for purposes of explanation.) MyString is assignable to either String or to Object. Here the compiler makes a determination as to which match is "better" and selects that method. In this case it's the String method.

The intuitive explanation is that the String class is closer to MyString in the inheritance hierarchy. It is a *more specific* match. A more rigorous way of specifying it would be to say that a given method is more specific than another method with respect to some arguments it wants to accept if the argument types of the first method are all assignable to the argument types of the second method. In this case, the String method is more specific to a subclass of String than the Object method because type String is assignable to type Object. The reverse is obviously not true.

If you're paying close attention, you may have noticed I said that the compiler resolves overloaded methods. Method overloading is not something that happens at run-time; this is an important distinction. It means that the selected method is chosen once, when the code is compiled. Once the overloaded method is selected, the choice is fixed until the code is recompiled, even if the class containing the called method is later revised and an even more specific overloaded method is added. This is in contrast to overridden (virtual) methods, which are located at run-time and can be found even if they didn't exist when the calling class was compiled. We'll talk about method overriding later in the chapter.

One last note about overloading. In earlier chapters, we've pointed out that Java doesn't support programmer-defined overloaded operators, and that + is the only system-defined overloaded operator. If you've been wondering what an overloaded operator is, I can finally clear up that mystery. In a language like C++, you can customize operators such as + and * to work with objects that you create. For example, you could create a class `Complex` that implements complex numbers, and then overload methods corresponding to + and * to add and multiply `Complex` objects. Some people argue that operator overloading makes for elegant and readable programs, while others say it's just "syntactic sugar" that makes for obfuscated code. The Java designers clearly espoused the later opinion when they chose not to support programmer-defined overloaded operators.

5.3 Object Creation

Objects in Java are allocated from a system heap space, much like `malloc`'ed storage in C or C++. Unlike C or C++, however, we needn't manage that memory ourselves. Java takes care of memory allocation and deallocation for you. Java explicitly allocates storage for an object when you create it with the `new` keyword. More importantly, objects are removed by garbage collection when they're no longer referenced.

Constructors

You allocate an object by specifying the `new` operator with an object *constructor*. A constructor is a special method with the same name as its class and no return type. It's called when a new class instance is created, which gives the class an opportunity to set up the object for use. Constructors, like other methods, can accept arguments and can be overloaded (they are not, however, inherited like other methods; we'll discuss inheritance later).

```
class Date {
    long time;

    Date() {
        time = currentTime();
    }

    Date( String date ) {
        time = parseDate( date );
    }
    ...
}
```

In the above example, the class `Date` has two constructors. The first takes no arguments; it's known as the default constructor. Default constructors play a special role in that, if we don't define any constructors for a class, an empty default constructor is supplied for us. The default constructor is what gets called whenever you create an object by calling its constructor with no arguments. Here we have implemented the default constructor so that it sets the instance variable `time` by calling a hypothetical method: `currentTime()`, which resembles the functionality of the real `java.util.Date` class.

The second constructor takes a `String` argument. Presumably, this `String` contains a string representation of the time that can be parsed to set the `time` variable.

Given the constructors above, we create a `Date` object in the following ways:

```
Date now = new Date();
Date christmas = new Date("Dec 25, 1997");
```

In each case, Java chooses the appropriate constructor at compile-time based on the rules for overloaded method selection.

If we later remove all references to an allocated object, it'll be garbage collected, as we'll discuss shortly:

```
christmas = null;          // fair game for the garbage collector
```

Setting the above reference to `null` means it's no longer pointing to the "Dec 25, 1997" object. Unless that object is referenced by another variable, it's now inaccessible and can be garbage collected. Actually, setting `christmas` to any other value would have the same results, but using the value `null` is a clear way to indicate that `christmas` no longer has a useful value.

A few more notes about constructors. Constructors can't be `abstract`, `synchronized`, or `final`. Constructors can, however, be declared with the visibility modifiers public, private, or protected, to control their accessibility. We'll talk in detail about visibility modifiers later in the chapter.

Working with Overloaded Constructors

A constructor can refer to another constructor in the same class or the immediate superclass using special forms of the `this` and `super` references. We'll discuss the first case here, and return to that of the superclass constructor again after we have talked more about subclassing and inheritance.

A constructor can invoke another, overloaded constructor in its class using the reference `this()` with appropriate arguments to select the desired constructor. If a constructor calls another constructor, it must do so as its first statement (we'll explain why in a bit):

```
class Car {
    String model;
    int doors;
    Car( String m, int d ) {
        model = m;
        doors = d;
        // other, complicated setup
        ...
    }

    Car( String m ) {
        this( m, 4 );
    }
    ...
}
```

In the example above, the class `Car` has two overloaded constructors. The first, more explicit one, accepts arguments specifying the car's model and its number of doors and uses them to set up the object. We have also provided a simpler constructor that takes just the model as an argument and, in turn, calls the first constructor with a default value of four doors. The advantage of this approach is that you can have a single constructor do all the complicated setup work; other auxiliary constructors simply feed the appropriate arguments to that constructor.

The important point is the call to `this()`, which must appear as the first statement our second constructor. The syntax is restricted in this way because there's a need to identify a clear chain of command in the calling of constructors. At one end of the chain, Java invokes the constructor of the superclass (if we don't do it explicitly) to ensure that inherited members are initialized properly before we proceed. There's also a point in the chain, just after the constructor of the superclass is invoked, where the initializers of the current class's instance variables are evaluated. Before that point, we can't even reference the instance variables of our class. We'll explain this situation again in complete detail after we have talked about inheritance.

For now, all you need to know is that you can invoke a second constructor only as the first statement of another constructor. In addition, you can't do anything at that point other than pass along arguments of the current constructor. For example, the following is illegal and causes a compile-time error:

```
Car( String m ) {
    int doors = determineDoors();
    this( m, doors );   // Error
}                       // Constructor call must be first statement
```

The simple model name constructor can't do any additional setup before calling the more explicit constructor. It can't even refer to an instance member for a constant value:

```
class Car {
    ...
    final int default_doors = 4;
    ...

    Car( String m ) {
        this( m, default_doors ); // Error
                                  // Referencing uninitialized variable
    }
    ...
}
```

The instance variable `defaultDoors` above is not initialized until a later point in the chain of constructor calls, so the compiler doesn't let us access it yet. Fortunately, we can solve this particular problem by making the identifier `static` as well:

```
class Car {
    ...
    static final int DEFAULT_DOORS = 4;
    ...

    Car( String m ) {
        this( m, DEFAULT_DOORS );  // Okay now
    }
    ...
}
```

The `static` members of our class have been initialized for some time (since the class was first loaded), so it's safe to access them.

Static and Nonstatic Code Blocks

It's possible to declare a code block (some statements within curly braces) directly within the scope of a class. This code block doesn't belong to any method; instead, it's executed just once, at the time the object is constructed, or, in the case of a code block marked static, at the time the class is loaded.

Nonstatic code blocks can be thought of as just an extension of instance variable initialization. They are called at the time the instance variable's initializers are evaluated (after superclass construction), in the textual order in which they appear in the class source.

```
class MyClass {
    Properties myProps = new Properties();
    // set up myProps
    {
        myProps.put("foo", "bar");
        myProps.put("boo", "gee);
    }
    int a = 5;
    ...
```

You can use static code blocks to initialize `static` class members in this way. So the static members of a class can have complex initialization just like objects:

```
class ColorWheel {
    static Hashtable colors = new Hashtable();

    // set up colors
    static {
        colors.put("Red", Color.red );
        colors.put("Green", Color.green );
        colors.put("Blue", Color.blue );
        ...
    }
    ...
}
```

In the above example, the class `ColorWheel` provides a variable `colors` that maps the names of colors to `Color` objects in a `Hashtable`. The first time the class `ColorWheel` is referenced and loaded, the `static` components of `ColorWheel` are evaluated, in the order they appear in the source. In this case, the `static` code block simply adds elements to the `colors` `Hashtable`.

PREVIOUS | HOME | NEXT
Methods | BOOK INDEX | Object Destruction

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

5.5 Subclassing and Inheritance

Classes in Java exist in a class hierarchy. A class in Java can be declared as a *subclass* of another class using the `extends` keyword. A subclass *inherits* variables and methods from its *superclass* and uses them as if they're declared within the subclass itself:

```
class Animal {
    float weight;
    ...
    void eat() {
        ...
    }
    ...
}

class Mammal extends Animal {
    int heartRate;
    // inherits weight
    ...
    void breathe() {
        ...
    }
    // inherits eat()
}
```

In the above example, an object of type `Mammal` has both the instance variable `weight` and the method `eat()`. They are inherited from `Animal`.

A class can extend only one other class. To use the proper terminology, Java allows *single inheritance* of class implementation. Later we'll talk about interfaces, which take the place of *multiple inheritance* as it's primarily used in C++.

A subclass can, of course, be further subclassed. Normally, subclassing specializes or refines a class by adding variables and methods:

```
class Cat extends Mammal {
    boolean longHair;
    // inherits weight and heartRate
    ...
    void purr() {
        ...
    }
    // inherits eat() and breathe()
}
```

The `Cat` class is a type of `Mammal` that is ultimately a type of `Animal`. `Cat` objects inherit all the characteristics of `Mammal` objects and, in turn, `Animal` objects. `Cat` also provides additional behavior in the form of the `purr()` method and the `longHair` variable. We can denote the class relationship in a diagram, as shown in Figure 5.3.

Figure 5.3: A class hierarchy

[Graphic: Figure 5-3]

A subclass inherits all members of its superclass not designated as `private`. As we'll discuss shortly, other levels of visibility affect what inherited members of the class can be seen from outside of the class and its subclasses, but at a minimum, a subclass always has the same set of visible members as its parent. For this reason, the type of a subclass can be considered a subtype of its parent, and instances of the subtype can be used anywhere instances of the supertype are allowed. For example:

```
Cat simon = new Cat();
Animal creature = simon;
```

The `Cat` `simon` in the above example can be assigned to the `Animal` type variable `creature` because `Cat` is a subtype of `Animal`.

Shadowed Variables

In the previous section on methods, we saw that a local variable of the same name as an instance variable hides the instance variable. Similarly, an instance variable in a subclass can shadow an instance variable of the same name in its parent class, as shown in Figure 5.4.

Figure 5.4: The scope of shadowed variables

[Graphic: Figure 5-4]

In Figure 5.4, the variable `weight` is declared in three places: as a local variable in the method `foodConsumption()` of the class `Mammal`, as an instance variable of the class `Mammal`, and as an instance variable of the class `Animal`. The actual variable selected depends on the scope in which we are working.

In the above example, all variables were of the same type. About the only reason for declaring a variable with the same type in a subclass is to provide an alternate initializer. A more important use of shadowed variables involves changing their types. We could, for example, shadow an `int` variable with a `double` variable in a subclass that needs decimal values instead of integer values. We do this without changing the existing code because, as its name suggests, when we shadow variables, we don't replace them but instead mask them. Both variables still exist; methods of the superclass see the original variable, and methods of the subclass see the new version. The determination of what variables the various methods see is static and happens at compile-time.

Here's a simple example:

```
class IntegerCalculator {
    int sum;
    ...
}

class DecimalCalculator extends IntegerCalculator {
    double sum;
    ...
}
```

In this example, we override the instance variable sum to change its type from int to double.[3] Methods defined in the class IntegerCalculator see the integer variable sum, while methods defined in DecimalCalculator see the decimal variable sum. However, both variables actually exist for a given instance of DecimalCalculator, and they can have independent values. In fact, any methods that DecimalCalculator inherits from IntegerCalculator actually see the integer variable sum.

[3] Note that a better way to design our calculators would be to have an abstract Calculator class with two subclasses: IntegerCalculator and DecimalCalculator.

Since both variables exist in DecimalCalculator, we need to reference the variable inherited from IntegerCalculator. We do that using the super reference:

```
int s = super.sum
```

Inside of DecimalCalculator, the super keyword used in this manner refers to the sum variable defined in the superclass. I'll explain the use of super more fully in a bit.

Another important point about shadowed variables has to do with how they work when we refer to an object by way of a less derived type. For example, we can refer to a DecimalCalculator object as an IntegerCalculator. If we do so and then access the variable sum, we get the integer variable, not the decimal one:

```
DecimalCalculator dc = new DecimalCalculator();
IntegerCalculator ic = dc;

int s = ic.sum;                  // Accesses IntegerCalculator sum
```

After this detailed explanation, you may still be wondering what shadowed variables are good for. Well, to be honest, the usefulness of shadowed variables is limited, but it's important to understand the concepts before we talk about doing the same thing with methods. We'll see a different and more dynamic type of behavior with method shadowing, or more correctly, *method overriding*.

Overriding Methods

In a previous section, we saw we could declare overloaded methods (i.e., methods with the same name but a different number or type of arguments) within a class. Overloaded method selection works the way I described on all methods available to a class, including inherited ones. This means that a subclass can define some overloaded methods that augment the overloaded methods provided by a superclass.

But a subclass does more than that; it can define a method that has exactly the *same* method signature (arguments and return type) as a method in its superclass. In that case, the method in the subclass *overrides* the method in the superclass and effectively replaces its implementation, as shown in Figure 5.5. Overriding methods to change the behavior of objects is another form of polymorphism (sub-type polymorphism): the one most people think of when they talk about the power of object-oriented languages.

Figure 5.5: Method overriding

[Graphic: Figure 5-5]

In Figure 5.5, `Mammal` overrides the `reproduce()` method of `Animal`, perhaps to specialize the method for the peculiar behavior of `Mammals` giving live birth.[4] The `Cat` object's sleeping behavior is overridden to be different from that of a general `Animal`, perhaps to accommodate cat naps. The `Cat` class also adds the more unique behaviors of purring and hunting mice.

[4] We'll ignore the platypus, which is an obscure nonovoviviparous mammal.

From what you've seen so far, overridden methods probably look like they shadow methods in superclasses, just as variables do. But overridden methods are actually more powerful than that. An overridden method in Java acts like a `virtual` method in C++. When there are multiple implementations of a method in the inheritance hierarchy of an object, the one in the most derived class always overrides the others, even if we refer to the object by way of a less derived type. In other words, if we have a `Cat` instance assigned to a variable of the more general type `Animal` and we call its `sleep()` method, we get the `sleep()` method implemented in the `Cat` class, not the one in `Animal`:

```
Cat simon = new Cat();
Animal creature = simon;

creature.sleep();          // Accesses Cat sleep();
```

In other respects, the variable `creature` looks like an `Animal`. For example, access to a shadowed variable would find the implementation in the `Animal` class, not the `Cat` class. However, because methods are virtual, the appropriate method in the `Cat` class can be located, even though we are dealing with an `Animal` object. This means we can deal with specialized objects as if they were more general types of objects and still take advantage of their specialized implementations of behavior.

Much of what you'll be doing when you're writing a Java applet or application is overriding methods defined by various classes in the Java API. For example, think back to the applets we developed in the tutorial in Chapter 2, *A First Applet*. Almost all of the methods we implemented for those applets were overridden methods. Recall that we created a subclass of `Applet` for each of the examples. Then we overrode various methods: `init()` set up our applet, `mouseDrag()` to handle mouse movement, and `paint()` to draw our applet.

A common programming error in Java (at least for me) is to miss and accidentally overload a method when trying to override it. Any difference in the number or type of arguments or the return type of a method produces two overloaded methods instead of a single, overridden method. Make it a habit to look twice when overriding methods.

Overridden methods and dynamic binding

In a previous section, I mentioned that overloaded methods are selected by the compiler at compile-time. Overridden methods, on the other hand, are selected dynamically at run-time. Even if we create an instance of a subclass our code has never seen before (perhaps a new object type loaded from the network), any overridden methods that it contains will be located and invoked at run-time to replace those that existed when we last compiled our code.

In contrast, if we load a new class that implements an additional, more specific overloaded method, our code will continue to use the implementation it discovered at compile-time. Another effect of this is that casting (i.e., explicitly telling the compiler to treat an object as one of its assignable types) affects the selection of overloaded methods, but not overridden methods.

Static method binding

Static methods do not belong to any object instance, they are accessed directly through a class name, so they are not dynamically selected at run-time like instance methods. That is why static methods are called "static"--they are always bound at compile time.

A `static` method in a superclass can be shadowed by another `static` method in a subclass, as long as the original method was not declared `final`. However, you can't override a `static` method with a non`static` method. In other words, you can't change a `static` method into an instance method in a subclass.

Dynamic method selection and peformance

When Java has to dynamically search for overridden methods in subclasses, there's a small performance penalty. In languages like C++, the default is for methods to act like shadowed variables, so you have to explicitly declare the methods you want to be virtual. Java is more dynamic, and the default is for all instance methods to be virtual. In Java you can, however, go the other direction and explicitly declare that an instance method can't be overridden, so that it will not be subject to dynamic binding and will not suffer in terms of performance. This is done with the `final` modifier. We have seen `final` used with variables to effectively make them constants. When `final` is applied to a method, it means that that method can't be overridden (in some sense, its implementation is constant). `final` can also be applied to an entire class, which means the class can't be subclassed.

Compiler optimiziations

When *javac*, the Java compiler, is run with the `-O` switch, it performs certain optimizations. It can inline `final` methods to improve performance (while slightly increasing the size of the resulting class file). `private` methods, which are effectively `final`, can also be inlined, and `final` classes may also benefit from more powerful optimizations.

Another kind of optimization allows you to include debugging code in your Java source without penalty. Java doesn't have a pre-processor, to explicitly control what source is included, but you can get some of the same effects by making a block of code conditional on a constant (i.e., `static` and `final`) variable. The Java compiler is smart enough to remove this code when it determines that it won't be called. For example:

```
static final boolean DEBUG = false;
...
static void debug (String message) {
    if (DEBUG) {
        System.err.println(message);
        // do other stuff
        ...
    }
}
```

If we compile the above code using the -O switch, the compiler will recognize that the condition on the DEBUG variable is always false, and the body of the debug() method will be optimized away. But that's not all--since debug() itself is also final it can be inlined, and an empty inlined method generates no code at all. So, when we compile with DEBUG set to false, calls to the debug() method will generate no residual code at all.

Method selection revisited

By now you should have a good, intuitive idea as to how methods are selected from the pool of potentially overloaded and overridden method names of a class. If, however, you are dying for a dry definition, I'll provide one now. If you are satisfied with your understanding, you may wish to skip this little exercise in logic.

In a previous section, I offered an inductive rule for overloaded method resolution. It said that a method is considered more specific than another if its arguments are polymorphically assignable to the arguments of the second method. We can now expand this rule to include the resolution of overridden methods by adding the following condition: to be more specific than another method, the type of the class containing the method must also be assignable to the type of the class holding the second method.

What does that mean? Well, the only classes whose types are assignable are classes in the same inheritance hierarchy. So, what we're talking about now is the set of all methods of the same name in a class or any of its parent or child classes. Since subclass types are assignable to superclass types, but not vice versa, the resolution is pushed, in the way that we expect, down the chain, towards the subclasses. This effectively adds a second dimension to the search, in which resolution is pushed down the inheritance tree towards more refined classes and, simultaneously, towards the most specific overloaded method within a given class.

Exceptions and overridden methods

When we talked about exception handling in Chapter 4, *The Java Language*, there's one thing I didn't mention because it wouldn't have made sense then. An important restriction on overridden methods is that they must adhere to the `throws` clause of the parent method's signature. If an overridden method throws an exception, the exception must be of the type specified by the parent or a subtype of that exception. Because the exception can be a subtype of the one specified by the parent, the overridden method can refine the type of exception thrown to go along with its new behavior. For example:

```
// A more refined exception
class MeatInedibleException extends InedibleException { ...
}
```

```
class Animal {
    void eat( Food f ) throws InedibleException { ...
}
class Herbivore extends Animal {
    void eat( Food f ) throws InedibleException {
        if ( f instanceof Meat )
            throw new MeatInedibleException();
        ....
    }
}
```

In the example above, Animal specifies that it can throw an InedibleException from its eat() method. Herbivore is a subclass Animal, so it must be able to do this too. However, Herbivore's eat() method actually throws a more specific exception: MeatInedibleException. It can do this because MeatInedibleException is a subtype of InedibleException (remember that Exceptions are classes too). Our calling code's *catch* clause can therefore be more specific:

```
Animal creature = ...
try {
    creature.eat( food );
} catch ( MeatInedibleException ) {
    // creature can't eat this food because it's meat
} catch ( InedibleException ) {
    // creature can't eat this food
}
```

this and *super*

The special references `this` and `super` allow you to refer to the members of the current object instance or those of the superclass, respectively. We have seen `this` used elsewhere to pass a reference to the current object and to refer to shadowed instance variables. The reference `super` does the same for the parents of a class. You can use it to refer to members of a superclass that have been shadowed or overridden. A common arrangement is for an overridden method in a subclass to do some preliminary work and then defer to the method of the superclass to finish the job.

```
class Animal {
    void eat( Food f ) throws InedibleException {
        // consume food
    }
}
```

```
class Herbivore extends Animal {
    void eat( Food f ) throws InedibleException {
        // check if edible
        ...
        super.eat( f );
    }
}
```

In the above example, our Herbivore class overrides the Animal eat() method to first do some checking on the food object. After doing its job it simply calls the (otherwise overridden) implementation of eat() in its superclass, using super.

super prompts a search for the method or variable to begin in the scope of the immediate superclass rather than the current class. The inherited method or variable found may reside in the immediate superclass, or in a more distant one. The usage of the super reference when applied to overridden methods of a superclass is special; it tells the method resolution system to stop the dynamic method search at the superclass, instead of in the most derived class (as it otherwise does). Without super, there would be no way to access overridden methods.

Casting

As in C++, a *cast* explicitly tells the compiler to change the apparent type of an object reference. Unlike in C++, casts in Java are checked both at compile- and at run-time to make sure they are legal. Attempting to cast an object to an incompatible type at run-time results in a ClassCastException. Only casts between objects in the same inheritance hierarchy (and as we'll see later, to appropriate interfaces) are legal in Java and pass the scrutiny of the compiler and the run-time system.

Casts in Java affect only the treatment of references; they never change the form of the actual object. This is an important rule to keep in mind. You never change the object pointed to by a reference by casting it; you change only the compiler's (or run-time system's) notion of it.

A cast can be used to *narrow* the type of a reference--to make it more specific. Often, we'll do this when we have to retrieve an object from a more general type of collection or when it has been previously used as a less derived type. (The prototypical example is using an object in a Vector or Hashtable, as you'll see in Chapter 7, *Basic Utility Classes*.) Continuing with our Cat example:

```
Animal creature = ...
Cat simon = ...

creature = simon;         // Okay
// simon = creature;      // Compile time error, incompatible type
simon = (Cat)creature;    // Okay
```

We can't reassign the reference in `creature` to the variable `simon` even though we know it holds an instance of a `Cat` (Simon). We have to perform the indicated cast. This is also called *downcasting* the reference.

Note that an implicit cast was performed when we went the other way to *widen* the reference `simon` to type `Animal` during the first assignment. In this case, an explicit cast would have been legal, but superfluous.

If casting seems complicated, here's a simple way to think about it. Basically, you can't lie about what an object is. If you have a `Cat` object, you can cast it to a less derived type (i.e., a type above it in the class hierarchy) such as `Animal` or even `Object`, since all Java classes are a subclass of `Object`. If you have an `Object` you know is a `Cat`, you can downcast the `Object` to be an `Animal` or a `Cat`. However, if you aren't sure if the `Object` is a `Cat` or a `Dog` at run-time, you should check it with `instanceof` before you perform the cast. If you get the cast wrong, Java throws a `ClassCastException`.

As I mentioned earlier, casting can affect the selection of compile-time items like variables and overloaded methods, but not the selection of overridden methods. Figure 5.6 shows the difference. As shown in the top half of the diagram, casting the reference `simon` to type `Animal` (widening it) affects the selection of the shadowed variable `weight` within it. However, as the lower half of the diagram indicates, the cast doesn't affect the selection of the overridden method `sleep()`.

Figure 5.6: Casting and its effect on method and variable selection

[Graphic: Figure 5-6]

Using superclass constructors

When we talked earlier about constructors, we discussed how the special statement `this()` invokes an overloaded constructor upon entry to another constructor. Similarly, the statement `super()` explicitly invokes the constructor of a superclass. Of course, we also talked about how Java makes a chain of constructor calls that includes the superclass's constructor, so why use `super()` explicitly? When Java makes an implicit call to the superclass constructor, it calls the default constructor. So, if we want to invoke a superclass constructor that takes arguments, we have to do so explicitly using `super()`.

If we are going to call a superclass constructor with `super()`, it must be the first statement of our constructor, just as `this()` must be the first call we make in an overloaded constructor. Here's a simple example:

```
class Person {
    Person ( String name ) {
        //  setup based on name
        ...
    }
    ...
}

class Doctor extends Person {
    Doctor ( String name, String specialty ) {
        super( name );
        // setup based on specialty
        ...
    }
    ...
}
```

In this example, we use `super()` to take advantage of the implementation of the superclass constructor and avoid duplicating the code to set up the object based on its name. In fact, because the class `Person` doesn't define a default (no arguments) constructor, we have no choice but to call `super()` explicitly. Otherwise, the compiler would complain that it couldn't find an appropriate default constructor to call. Said another way, if you subclass a class that has only constructors that take arguments, you have to invoke one of the superclass's constructors explicitly from your subclass constructor.

Instance variables of the class are initialized upon return from the superclass constructor, whether that's due to an explicit call via `super()` or an implicit call to the default superclass constructor.

We can now give the full story of how constructors are chained together and when instance variable initialization occurs. The rule has three parts and is applied repeatedly for each successive constructor invoked.

- If the first statement of a constructor is an ordinary statement--i.e., not a call to `this()` or `super()`--Java inserts an implicit call to `super()` to invoke the default constructor of the superclass. Upon returning from that call, Java initializes the instance variables of the current class and proceeds to execute the statements of the current constructor.

- If the first statement of a constructor is a call to a superclass constructor via `super()`, Java invokes the selected superclass constructor. Upon its return, Java initializes the current class's instance variables and proceeds with the statements of the current constructor.

- If the first statement of a constructor is a call to an overloaded constructor via `this()`, Java invokes the selected constructor and upon its return simply proceeds with the statements of the current constructor. The call to the superclass's constructor has happened within the overloaded constructor, either explicitly or implicitly, so the initialization of instance variables has already occurred.

Abstract Methods and Classes

As in C++, a method can be declared with the `abstract` modifier to indicate that it's just a prototype. An abstract method has no body; it's simply a signature definition followed by a semicolon. You can't directly use a class that contains an abstract method; you must instead create a subclass that implements the abstract method's body.

```
abstract vaporMethod( String name );
```

In Java, a class that contains one or more abstract methods must be explicitly declared as an abstract class, also using the `abstract` modifier :

```
abstract class vaporClass {
    ...
    abstract vaporMethod( String name );
    ...
}
```

An abstract class can contain other, nonabstract methods and ordinary variable declarations; however, it can't be instantiated. To be used, it must be subclassed and its abstract methods must be overridden with methods that implement a body. Not all abstract methods have to be implemented in a single subclass, but a subclass that doesn't override all its superclass's abstract methods with actual, concrete implementations must also be declared `abstract`.

Abstract classes provide a framework for classes that are to be "filled in" by the implementor. The `java.io.InputStream` class, for example, has a single abstract method called `read()`. Various subclasses of `InputStream` implement `read()` in their own ways to read from their own sources. The rest of the `InputStream` class, however, provides extended functionality built on the simple `read()` method. A subclass of `InputStream` inherits these nonabstract methods that provide functionality based on the simple `read()` method that the subclass implements.

It's often desirable to specify only the prototypes for a whole set of methods and provide no implementation. In C++, this would be a purely abstract class. In Java, you should instead use an *interface*. An interface is like a purely abstract class; it defines a set of methods a class must implement (i.e., the behavior of a class). However, unlike in C++, a class in Java can simply say that it `implements` an interface and go about implementing those methods. As we'll discuss later, a class that implements an interface doesn't have to inherit from any particular part of the inheritance hierarchy or use a particular implementation.

PREVIOUS | HOME | NEXT
Object Destruction | BOOK INDEX | Packages and Compilation Units

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

5.6 Packages and Compilation Units

A *package* is a name for a group of related classes. In Chapter 3, *Tools of the Trade*, we discussed how Java uses package names to locate classes during compilation and at run-time. In this sense, packages are somewhat like libraries; they organize and manage sets of classes. Packages provide more than just source code-level organization though. They also create an additional level of scope for their classes and the variables and methods within them. We'll talk about the visibility of classes in this section. In the next section, we'll discuss the effect that packages have on access to variables and methods between classes.

Compilation Units

The source code for a Java class is called a *compilation unit*. A compilation unit normally contains a single class definition and is named for that class. The definition of a class named `MyClass`, for instance, should appear in a file named *MyClass.java*. For most of us, a compilation unit is just a file with a *.java* extension, but in an integrated development environment, it could be an arbitrary entity. For brevity here, we'll refer to a compilation unit simply as a file.

The division of classes into their own compilation units is important because, as described in Chapter 3, *Tools of the Trade*, the Java compiler assumes much of the responsibility of a *make* utility. The compiler relies on the names of source files to find and compile dependent classes. It's possible (and common) to put more than one class definition into a single file, but there are some restrictions we'll discuss shortly.

A class is declared to belong to a particular package with the `package` statement. The `package` statement must appear as the first statement in a compilation unit. There can be only one `package` statement, and it applies to the entire file:

```
package mytools.text;

class TextComponent {
     ...
}
```

In the above example, the class `TextComponent` is placed in the package `mytools.text`.

A Word About Package Names

You should recall from Chapter 3, *Tools of the Trade* that package names are constructed in a hierarchical way, using a dot-separated naming convention. Package-name components construct a unique path for the compiler and run-time systems to locate files; however, they don't affect the contents directly in any other way. There is no such thing as a subpackage (the package name space is really flat, not hierarchical) and packages under a particular part of a package hierarchy are related only by association. For example, if we create another package called `mytools.text.poetry` (presumably for text classes specialized in some way to work with poetry), those classes would not be considered part of the `mytools.text` package and would have no special access to its members. In this sense, the package-naming convention can be misleading.

Class Visibility

By default, a class is accessible only to other classes within its package. This means that the class `TextComponent` is available only to other classes in the `mytools.text` package. To be visible elsewhere, a class must be declared as `public`:

```
package mytools.text;

public class TextEditor {
     ...
}
```

The class `TextEditor` can now be referenced anywhere. There can be only a single `public` class defined in a compilation unit; the file must be named for that class.

By hiding unimportant or extraneous classes, a package builds a subsystem that has a well-defined interface to the rest of the world. Public classes provide a facade for the operation of the system and the details of its inner workings can remain hidden, as shown in Figure 5.7. In this sense, packages hide classes in the way classes hide `private` members.

Figure 5.7: Class visibility and packages

[Graphic: Figure 5-7]

Figure 5.7 shows part of the the hypothetical `mytools.text` package. The classes `TextArea` and `TextEditor` are declared `public` and can be used elsewhere in an application. The class `TextComponent` is part of the implementation of `TextArea` and is not accessible from outside of the package.

Importing Classes

Classes within a package can refer to each other by their simple names. However, to locate a class in another package, we have to supply a qualifier. Continuing with the above example, an application refers directly to our editor class by its fully qualified name of `mytools.text.TextEditor`. But we'd quickly grow tired of typing such long class names, so Java gives us the `import` statement. One or more `import` statements can appear at the top of a compilation unit, beneath the `package` statement. The `import` statements list the full names of classes to be used within the file. Like a `package` statement, `import` statements apply to the entire compilation unit. Here's how you might use an `import` statement:

```
package somewhere.else;

import mytools.text.TextEditor;

class MyClass {
    TextEditor editBoy;
    ...
}
```

As shown in the example above, once a class is imported, it can be referenced by its simple name throughout the code. It's also possible to import all of the classes in a package using the `*` notation:

```
import mytools.text.*;
```

Now we can refer to all `public` classes in the `mytools.text` package by their simple names.

Obviously, there can be a problem with importing classes that have conflicting names. If two different packages contain classes that use the same name, you just have to fall back to using fully qualified names to refer to those classes. Other than the potential for naming conflicts, there's no penalty for importing classes. Java doesn't carry extra baggage into the compiled class files. In other words, Java class files don't contain other class definitions, they only reference them.

The Unnamed Package

A class that is defined in a compilation unit that doesn't specify a package falls into the large, amorphous unnamed package. Classes in this nameless package can refer to each other by their simple names. Their path at compile- and run-time is considered to be the current directory, so package-less classes are useful for experimentation, testing, and brevity in providing examples for books about Java.

PREVIOUS	HOME	NEXT
Subclassing and Inheritance	BOOK INDEX	Variable and Method Visibility

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

5.8 Interfaces

Interfaces are kind of like Boy Scout (or Girl Scout) merit badges. When a scout has learned to build a bird house, he can walk around wearing a little patch with a picture of one on his sleeve. This says to the world, "I know how to build a bird house." Similarly, an *interface* is a list of methods that define some set of behavior for an object. Any class that implements each of the methods listed in the interface can declare that it implements the interface and wear, as its merit badge, an extra type--the interface's type.

Interface types act like class types. You can declare variables to be of an interface type, you can declare arguments of methods to accept interface types, and you can even specify that the return type of a method is an interface type. In each of these cases, what is meant is that any object that implements the interface (i.e., wears the right merit badge) can fill that spot. In this sense, interfaces are orthogonal to the class hierarchy. They cut across the boundaries of what kind of object an item is and deal with it only in terms of what it can do. A class implements as many interfaces as it desires. In this way, interfaces in Java replace the need for multiple inheritance (and all of its messy side effects).

An interface looks like a purely `abstract` class (i.e., a class with only `abstract` methods). You define an interface with the `interface` keyword and list its methods with no bodies:

```
interface Driveable {
    boolean startEngine();
    void stopEngine();
    float accelerate( float acc );
    boolean turn( Direction dir );
}
```

The example above defines an interface called `Driveable` with four methods. It's acceptable, but not necessary, to declare the methods in an interface with the `abstract` modifier, so we haven't used it here. Interfaces define capabilities, so it's common to name interfaces after their capabilities in a passive sense. "Driveable" is a good example; "runnable" and "updateable" would be two more.

Any class that implements all the methods can then declare it implements the interface by using a special `implements` clause in its class definition:

```
class Automobile implements Driveable {
    ...
    boolean startEngine() {
        if ( notTooCold )
            engineRunning = true;
        ...
    }

    void stopEngine() {
        engineRunning = false;
    }

    float accelerate( float acc ) {
        ...
    }

    boolean turn( Direction dir ) {
        ...
    }
    ...
}
```

The class `Automobile` implements the methods of the `Driveable` interface and declares itself `Driveable` using an `implements` clause.

As shown in Figure 5.9, another class, such as `LawnMower`, can also implement the `Driveable` interface. The figure illustrates the `Driveable` interface being implemented by two different classes. While it's possible that both `Automobile` and `Lawnmower` could derive from some primitive kind of vehicle, they don't have to in this scenario. This is a significant advantage of interfaces over standard multiple inheritance as implemented in C++.

Figure 5.9: Implementing the Driveable interface

[Graphic: Figure 5-9]

After declaring the interface, we have a new type, `Driveable`. We can declare variables of type `Driveable` and assign them any instance of a `Driveable` object:

```
Automobile auto = new Automobile();
Lawnmower mower = new Lawnmower();
Driveable vehicle;

vehicle = auto;
vehicle.startEngine();
vehicle.stopEngine();
...
vehicle = mower;
vehicle.startEngine();
vehicle.stopEngine();
```

Both `Automobile` and `Lawnmower` implement `Driveable` and can be considered of that type.

Interfaces as Callbacks

Interfaces can be used to implement callbacks in Java. A *callback* is a situation where you'd like to pass a reference to some behavior and have another object invoke it later. In C or C++, this is prime territory for function pointers; in Java, we'll use interfaces instead.

Consider two classes: a `TickerTape` class that displays data and a `TextSource` class that provides an information feed. We'd like our `TextSource` to send any new text data. We could have `TextSource` store a reference to a `TickerTape` object, but then we could never use our `TextSource` to send data to any other kind of object. Instead, we'd have to proliferate subclasses of `TextSource` that dealt with different types. A more elegant solution is to have `TextSource` store a reference to an interface type, `TextUpdateable`:

```
interface TextUpdateable {
    receiveText( String text );
}

class TickerTape implements TextUpdateable {
    TextSource source;

    init() {
        source = new TextSource( this );
        ...
    }

    public receiveText( String text ) {
        scrollText( text ):
    }
    ...
}

class TextSource {
    TextUpdateable receiver;

    TextSource( TextUpdateable r ) {
        receiver = r;
    }

    private sendText( String s ) {
        receiver.receiveText( s );
    }
    ...
}
```

The only thing `TextSource` really cares about is finding the right method to invoke to send text. Thus, we can list that method in an interface called `TextUpdateable` and have our `TickerTape` implement the interface. A `TickerTape` object can then be used anywhere we need something of the type `TextUpdateable`. In this case, the `TextSource` constructor takes a `TextUpdateable` object and stores the reference in an instance variable of type `TextUpdateable`. Our `TickerTape` object simply passes a reference to itself as the callback for text updates, and the source can invoke its `receiveText()` method as necessary.

Interface Variables

Although interfaces allow us to specify behavior without implementation, there's one exception. An interface can contain constant variable identifiers; these identifiers appear in any class that implements the interface. This functionality allows for predefined parameters that can be used with the methods:

```
interface Scaleable {
    static final int BIG = 0, MEDIUM = 1, SMALL = 2;

    void setScale( int size );
}
```

The `Scaleable` interface defines three integers: `BIG`, `MEDIUM`, and `SMALL`. All variables defined in interfaces are implicitly `final` and `static`; we don't have to use the modifiers here but, for clarity, we recommend you do so.

A class that implements `Scaleable` sees these variables:

```
class Box implements Scaleable {

    void setScale( int size ) {
        switch( size ) {
            case BIG:
                ...
            case MEDIUM:
                ...
            case SMALL:
                ...
        }
    }
    ...
}
```

Empty Interfaces

Sometimes, interfaces are created just to hold constants; anyone who implements the interfaces can see the constant names, much as if they were included by a C/C++ include file. This is a somewhat degenerate, but acceptable use of interfaces.

Sometimes completely empty interfaces will be used to serve as a marker that a class has some special property. The java.io.Serializeable interface is a good example (See Chapter 8). Classes that implement Serializable don't add any methods or variables. Their additional type simply identifies them to Java as classes that want to be able to be serialized.

Interfaces and Packages

Interfaces behave like classes within packages. An interface can be declared `public` to make it visible outside of its package. Under the default visibility, an interface is visible only inside of its package. There can be only one `public` interface declared in a compilation unit.

Subinterfaces

An interface can extend another interface, just as a class can extend another class. Such an interface is called a *subinterface*:

```
interface DynamicallyScaleable extends Scaleable {
    void changeScale( int size );
}
```

The interface `DynamicallyScaleable` extends our previous `Scaleable` interface and adds an additional method. A class that implements `DynamicallyScaleable` must implement all methods of both interfaces.

Interfaces can't specify that they implement other interfaces, instead they are allowed to extend more than one interface. (This is multiple inheritence for interfaces). More than one superinterface can be specified with the comma operator:

```
    interface DynamicallyScaleable extends Scaleable, SomethingElseable {
        ...
```

Inside Arrays

At the end of Chapter 4, *The Java Language*, I mentioned that arrays have a place in the Java class hierarchy, but I didn't give you any details. Now that we've discussed the object-oriented aspects of Java, I can give you the whole story.

Array classes live in a parallel Java class hierarchy under the `Object` class. If a class is a direct subclass of `Object`, then an array class for that base type also exists as a direct subclass of `Object`. Arrays of more derived classes are subclasses of the corresponding array classes. For example, consider the following class types:

```
class Animal { ... }
class Bird extends Animal { ... }
class Penguin extends Bird { ... }
```

Figure 5.10 illustrates the class hierarchy for arrays of these classes.

Figure 5.10: Arrays in the Java class hierarchy

[Graphic: Figure 5-10]

Arrays of the same dimension are related to one another in the same manner as their base type classes. In our example, Bird is a subclass of Animal, which means that the Bird[] type is a subtype of Animal[]. In the same way a Bird object can be used in place of an Animal object, a Bird[] array can be assigned to an Animal[] array:

```
Animal [][] animals;
Bird [][] birds = new Bird [10][10];
birds[0][0] = new Bird();

// make animals and birds reference the same array object
animals = birds;
System.out.println( animals[0][0] );               // prints Bird
```

Because arrays are part of the class hierarchy, we can use instanceof to check the type of an array:

```
if ( birds instanceof Animal[][] )                 // yes
```

An array is a subtype of Object and can therefore be assigned to Object type variables:

```
Object something;
something = animals;
```

Since Java knows the actual type of all objects, you can also cast back if appropriate:

```
animals = (Animal [][])something;
```

Under unusual circumstances, Java may not be able to check the types of objects you place into arrays at compile-time. In those cases, it's possible to receive an ArrayStoreException if you try to assign the wrong type of object to an array element. Consider the following:

```
class Dog { ... }
class Poodle extends Dog { ... }
class Chihuahua extends Dog { ... }

Dog [] dogs;
Poodle [] poodles = new Poodle [10];

dogs = poodles;

dogs[3] = new Chihuahua();     // Run-time error, ArrayStoreException
```

Both Poodle and Chihuahua are subclasses of Dog, so an array of Poodle objects can therefore be assigned to an array of Dog objects, as I described previously. The problem is that an object assignable to an element of an array of type Dog[] may not be assignable to an element of an array of type Poodle. A Chihuahua object, for instance, can be assigned to a Dog element because it's a subtype of Dog, but not to a Poodle element.[6]

[6] In some sense this could be considered a tiny hole in the Java type system. It doesn't occur elsewhere in Java, only with arrays. This is because array objects exhibit *covariance* in overriding their assignment and extraction methods. Covariance allows array subclasses to override methods with arguments or return values that are subtypes of the overridden methods, where the methods would normally be overloaded or prohibited. This allows array subclasses to operate on their base types with type safety, but also means that subclasses have different capabilities than their parents, leading to the problem shown above.

PREVIOUS | HOME | NEXT
Variable and Method Visibility | BOOK INDEX | Inner Classes

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

5.10 The Object and Class Classes

`java.lang.Object` is the mother of all objects; it's the primordial class from which all other classes are ultimately derived. Methods defined in `Object` are therefore very important because they appear in every instance of any class, throughout all of Java. At last count, there were nine `public` methods in `Object`. Five of these are versions of `wait()` and `notify()` that are used to synchronize threads on object instances, as we'll discuss in Chapter 6, *Threads*. The remaining four methods are used for basic comparison, conversion, and administration.

Every object has a `toString()` method that is called when it's to be represented as a text value. `PrintStream` objects use `toString()` to print data, as discussed in Chapter 8, *Input/Output Facilities*. `toString()` is also used when an object is referenced in a string concatenation. Here are some examples:

```
MyObj myObject = new MyObj();
Answer theAnswer = new Answer();

System.out.println( myObject );
String s = "The answer is: " + theAnswer ;
```

To be friendly, a new kind of object should override `toString()` and implement its own version that provides appropriate printing functionality. Two other methods, `equals()` and `hashCode()`, may also require specialization when you create a new class.

Equality

`equals()` compares whether two objects are equivalent. Precisely what that means for a particular class is something that you'll have to decide for yourself. Two `String` objects, for example, are considered equivalent if they hold precisely the same characters in the same sequence:

```
String userName = "Joe";
```

```
...
if ( userName.equals( suspectName ) )
    arrest( userName );
```

Note that using `equals()` is *not* the same as:

```
// if ( userName == suspectName )      // Wrong!
```

The above code tests to see if the two `String` objects are the same object, which is sufficient but not necessary for them to be equivalent objects.

A class should override the `equals()` method if it needs to implement its own notion of equality. If you have no need to compare objects of a particular class, you don't need to override `equals()`.

Watch out for accidentally overloading `equals()` when you mean to override it. `equals()` takes an `Object` as an argument and returns a `boolean` value. While you'll probably want to check only if an object is equivalent to an object of its own type, in order to properly override `equals()`, the method should accept a generic `Object` as its argument. Here's an example of implementing `equals()`:

```
class Sneakers extends Shoes {
    public boolean equals( Object arg ) {
        if ( (arg != null) && (arg instanceof Sneakers) ) {
            // compare arg with this object to check equivalence
            // If comparison is okay...
            return true;
        }
        return false;
    }
    ...
}
```

A `Sneakers` object can now be properly compared by any current or future Java classes. If we had instead used a `Sneakers` type object as the argument to `equals()`, all would be well for classes that reference our objects as `Sneakers`, but methods that simply use `Shoes` would not see the overloaded method and would compare `Sneakers` against other `Sneakers` improperly.

Hashcodes

The `hashCode()` method returns an integer that is a hashcode for a class instance. A hashcode is like a signature for an object; it's an arbitrary-looking identifying number that is (with important exceptions) generally different for different instances of the class. Hashcodes are used in the process of storing objects in a `Hashtable`, or a similar kind of collection. The hashcode is essentially an index into the collection. See Chapter 7, *Basic Utility Classes* for a complete discussion of `Hashtable` objects and hashcodes.

The default implementation of `hashCode()` in `Object` assigns each object instance a unique number to be used as a hashcode. If you don't override this method when you create a new class, each instance of the class will have a unique hashcode. This is sufficient for most objects. However, if the class has a notion of equivalent objects, then you should probably override `hashCode()` so that equivalent objects are given the same hashcode.

java.lang.Class

The last method of `Object` we need to discuss is `getClass()`. This method returns a reference to the `Class` object that produced the object instance.

A good measure of the complexity of an object-oriented language is the degree of abstraction of its class structures. We know that every object in Java is an instance of a class, but what exactly is a class? In C++, objects are formulated by and instantiated from classes, but classes are really just artifacts of the compiler. Thus, you see only classes mentioned in C++ source code, not at run-time. By comparison, classes in Smalltalk are real, run-time entities in the language that are themselves described by "meta-classes" and "meta-class classes." Java strikes a happy medium between these two languages with what is, effectively, a two-tiered system that uses `Class` objects.

Classes in Java source code are represented at run-time by instances of the `java.lang.Class` class. There's a `Class` object for every class you use; this `Class` object is responsible for producing instances for its class. This may sound overwhelming, but you don't have to worry about any of it unless you are interested in loading new kinds of classes dynamically at run-time.

We can get the `Class` associated with a particular object with the `getClass()` method:

```
String myString = "Foo!"
Class c = myString.getClass();
```

We can also get the Class reference for a particular class statically, using the special `.class` notation:

```
Class c = String.class;
```

The .class reference looks like a static field that exists in every class. However, it is really resolved by the compiler.

One thing we can do with the `Class` object is to ask for the name of the object's class:

```
String s = "Boofa!";
Class strClass = s.getClass();
System.out.println( strClass.getName() ); // prints "java.lang.String"
```

Another thing that we can do with a `Class` is to ask it to produce a new instance of its type of object. Continuing with the above example:

```
try {
    String s2 = (String)strClass.newInstance();
}
catch ( InstantiationException e ) { ... }
catch ( IllegalAccessException e ) { ... }
```

`newInstance()` has a return type of `Object`, so we have to cast it to a reference of the appropriate type. A couple of problems can occur here. An `InstantiationException` indicates we're trying to instantiate an `abstract` class or an interface. `IllegalAccessException` is a more general exception that indicates we can't access a constructor for the object. Note that `newInstance()` can create only an instance of a class that has an accessible default constructor. There's no way for us to pass any arguments to a constructor.

All this becomes more meaningful when we add the capability to look up a `Class` by name. `forName()` is a `static` method of `Class` that returns a `Class` object given its name as a `String`:

```
try {
    Class sneakersClass = Class.forName("Sneakers");
}
catch ( ClassNotFoundException e ) { ... }
```

A `ClassNotFoundException` is thrown if the class can't be located.

Combining the above tools, we have the power to load new kinds of classes dynamically. When combined with the power of interfaces, we can use new data types by name in our applications:

```
interface Typewriter {
    void typeLine( String s );
    ...
}

class Printer implements Typewriter {
    ...
}
```

```
class MyApplication {
    ...
    String outputDeviceName = "Printer";

    try {
        Class newClass = Class.forName( outputDeviceName );
        Typewriter device = (Typewriter)newClass.newInstance();
        ...
        device.typeLine("Hello...");
    }
    catch ( Exception e ) {
}
```

7.2 Math Utilities

Java supports integer and floating-point arithmetic directly. Higher-level math operations are supported through the `java.lang.Math` class. Java provides wrapper classes for all primitive data types, so you can treat them as objects if necessary. Java also provides the `java.util.Random` class for generating random numbers.

Java handles errors in integer arithmetic by throwing an `ArithmeticException`:

```
int zero = 0;

try {
    int i = 72 / zero;
}
catch ( ArithmeticException e ) {       // division by zero
}
```

To generate the error in the above example, we created the intermediate variable `zero`. The compiler is somewhat crafty and would have caught us if we had blatantly tried to perform a division by zero.

Floating-point arithmetic expressions, on the other hand, don't throw exceptions. Instead, they take on the special out-of-range values shown in Table 7.3.

Table 7.3: Special Floating-Point Values

Value	**Mathematical representation**
`POSITIVE_INFINITY`	1.0/0.0
`NEGATIVE_INFINITY`	-1.0/0.0
`NaN`	0.0/0.0

The following example generates an infinite result:

```
double zero = 0.0;
double d = 1.0/zero;

if ( d == Double.POSITIVE_INFINITY )
    System.out.println( "Division by zero" );
```

The special value `NaN` indicates the result is "not a number." The value `NaN` has the special distinction of not being equal to itself (`NaN != NaN`). Use `Float.isNaN()` or `Double.isNaN()` to test for `NaN`.

java.lang.Math

The `java.lang.Math` class serves as Java's math library. All its methods are `static` and used directly ; you can't instantiate a `Math` object. We use this kind of degenerate class when we really want methods to approximate normal functions in C. While this tactic defies the principles of object-oriented design, it makes sense in this case, as it provides a means of grouping some related utility functions in a single class. Table 7.4 summarizes the methods in `java.lang.Math`.

Table 7.4: Methods in java.lang.Math

Method	**Argument type(s)**	**Functionality**
`Math.abs(a)`	`int`, `long`, `float`, `double`	Absolute value
`Math.acos(a)`	`double`	Arc cosine
`Math.asin(a)`	`double`	Arc sine
`Math.atan(a)`	`double`	Arc tangent
`Math.atan2(a,b)`	`double`	Converts rectangular to polar coordinates
`Math.ceil(a)`	`double`	Smallest whole number greater than or equal to `a`
`Math.cos(a)`	`double`	Cosine
`Math.exp(a)`	`double`	Exponential number to the power of `a`
`Math.floor(a)`	`double`	Largest whole number less than or equal to `a`
`Math.log(a)`	`double`	Natural logarithm of `a`
`Math.max(a, b)`	`int`, `long`, `float`, `double`	Maximum
`Math.min(a, b)`	`int`, `long`, `float`, `double`	Minimum
`Math.pow(a, b)`	`double`	`a` to the power of `b`

`Math.random()`	None	Random number generator
`Math.rint(a)`	`double`	Converts double value to integral value in double format
`Math.round(a)`	`float`, `double`	Rounds
`Math.sin(a)`	`double`	Sine
`Math.sqrt(a)`	`double`	Square root
`Math.tan(a)`	`double`	Tangent

`log()`, `pow()`, and `sqrt()` can throw an `ArithmeticException`. `abs()`, `max()`, and `min()` are overloaded for all the scalar values, `int`, `long`, `float`, or `double`, and return the corresponding type. Versions of `Math.round()` accept either `float` or `double` and return `int` or `long` respectively. The rest of the methods operate on and return `double` values:

```
double irrational = Math.sqrt( 2.0 );
int bigger = Math.max( 3, 4 );
long one = Math.round( 1.125798 );
```

For convenience, `Math` also contains the `static final double` values `E` and `PI`:

```
double circumference = diameter * Math.PI;
```

java.math

If a `long` or a `double` just isn't big enough for you, the `java.math` package provides two classes, `BigInteger` and `BigDecimal`, that support arbitrary-precision numbers. These are full-featured classes with a bevy of methods for performing arbitrary-precision math. In the following example, we use `BigInteger` to add two numbers together.

```
try {
    BigDecimal twentyone = new BigDecimal("21");
    BigDecimal seven = new BigDecimal("7");
    BigDecimal sum = twentyone.add(seven);

    int twentyeight = sum.intValue();
}
catch (NumberFormatException nfe) { }
catch (ArithmeticException ae) { }
```

Wrappers for Primitive Types

In languages like Smalltalk, numbers and other simple types are objects, which makes for an elegant language design, but has trade-offs in efficiency and complexity. By contrast, there is a schism in the Java world between class types (i.e., objects) and primitive types (i.e., numbers, characters, and boolean values). Java accepts this trade-off simply for efficiency reasons. When you're crunching numbers you want your computations to be lightweight; having to use objects for primitive types would seriously affect performance. For the times you want to treat values as objects, Java supplies a wrapper class for each of the primitive types, as shown in Table 7.5.

Table 7.5: Primitive Type Wrappers

Primitive	**Wrapper**
`void`	`java.lang.Void`
`boolean`	`java.lang.Boolean`
`char`	`java.lang.Character`
`byte`	`java.lang.Byte`
`short`	`java.lang.Short`
`int`	`java.lang.Integer`
`long`	`java.lang.Long`
`float`	`java.lang.Float`
`double`	`java.lang.Double`

An instance of a wrapper class encapsulates a single value of its corresponding type. It's an immutable object that serves as a container to hold the value and let us retrieve it later. You can construct a wrapper object from a primitive value or from a `String` representation of the value. The following code is equivalent:

```
Float pi = new Float( 3.14 );
Float pi = new Float( "3.14" );
```

Wrapper classes throw a `NumberFormatException` when there is an error in parsing from a string:

```
try {
    Double bogus = new Double( "huh?" );
}
catch ( NumberFormatException e ) {     // bad number
}
```

You should arrange to catch this exception if you want to deal with it. Otherwise, since it's a subclass of `RuntimeException`, it will propagate up the call stack and eventually cause a run-time error if not caught.

Sometimes you'll use the wrapper classes simply to parse the `String` representation of a number:

```
String sheep = getParameter("sheep");
int n = new Integer( sheep ).intValue();
```

Here we are retrieving the value of the `sheep` parameter. This value is returned as a `String`, so we need to convert it to a numeric value before we can use it. Every wrapper class provides methods to get primitive values out of the wrapper; we are using `intValue()` to retrieve an `int` out of `Integer`. Since parsing a `String` representation of a number is such a common thing to do, the `Integer` and `Long` classes also provide the `static` methods `Integer.parseInt()` and `Long.parseLong()` that read a `String` and return the appropriate type. So the second line above is equivalent to:

```
int n = Integer.parseInt( sheep );
```

All wrappers provide access to their values in various forms. You can retrieve scalar values with the methods `doubleValue()`, `floatValue()`, `longValue()`, and `intValue()`:

```
Double size = new Double ( 32.76 );

double d = size.doubleValue();
float f = size.floatValue();
long l = size.longValue();
int i = size.intValue();
```

The code above is equivalent to the primitive `double` value cast to the various types. For convenience, you can cast between the wrapper classes like `Double` class and the primitive data types.

Another common use of wrappers occurs when we have to treat a primitive value as an object in order to place it in a list or other structure that operates on objects. As you'll see shortly, a `Vector` is an extensible array of `Object`s. We can use wrappers to hold numbers in a `Vector`, along with other objects:

```
Vector myNumbers = new Vector();

Integer thirtyThree = new Integer( 33 );
myNumbers.addElement( thirtyThree );
```

Here we have created an `Integer` wrapper so that we can insert the number into the `Vector` using `addElement()`. Later, when we are taking elements back out of the `Vector`, we can get the number back out of the `Integer` as follows:

```
Integer theNumber = (Integer)myNumbers.firstElement();
int n = theNumber.intValue();           // n = 33
```

Random Numbers

You can use the `java.util.Random` class to generate random values. It's a pseudo-random number generator that can be initialized with a 48-bit seed.[1] The default constructor uses the current time as a seed, but if you want a repeatable sequence, specify your own seed with:

> [1] The generator uses a linear congruential formula. See *The Art of Computer Programming*, Volume 2 "Semi-numerical Algorithms," by Donald Knuth (Addison-Wesley).

```
long seed = mySeed;
Random rnums = new Random( seed );
```

This code creates a random-number generator. Once you have a generator, you can ask for random values of various types using the methods listed in Table 7.6.

Table 7.6: Random Number Methods

Method	**Range**
`nextInt()`	-2147483648 to 2147483647
`nextLong()`	-9223372036854775808 to 9223372036854775807
`nextFloat()`	-1.0 to 1.0
`nextDouble()`	-1.0 to 1.0

By default, the values are uniformly distributed. You can use the `nextGaussian()` method to create a Gaussian distribution of `double` values, with a mean of 0.0 and a standard deviation of 1.0.

The `static` method `Math.random()` retrieves a random `double` value. This method initializes a `private` random-number generator in the `Math` class, using the default `Random` constructor. So every call to `Math.random()` corresponds to a call to `nextDouble()` on that random number generator.

PREVIOUS HOME NEXT
Strings BOOK INDEX Dates

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

7.4 Vectors and Hashtables

Vectors and hashtables are *collection classes*. Each stores a group of objects according to a particular retrieval scheme. Aside from that, they are not particularly closely related things. A *hashtable* is a dictionary; it stores and retrieves objects by a key value. A *vector*, on the other hand, holds an ordered collection of elements. It's essentially a dynamic array. Both of these, however, have more subtle characteristics in common. First, they are two of the most useful aspects of the core Java distribution. Second, they both take full advantage of Java's dynamic nature at the expense of some of its more static type safety.

If you work with dictionaries or associative arrays in other languages, you should understand how useful these classes are. If you are someone who has worked in C or another static language, you should find collections to be truly magical. They are part of what makes Java powerful and dynamic. Being able to work with lists of objects and make associations between them is an abstraction from the details of the types. It lets you think about the problems at a higher level and saves you from having to reproduce common structures every time you need them.

java.util.Vector

A `Vector` is a dynamic array; it can grow to accommodate new items. You can also insert and remove elements at arbitrary positions within it. As with other mutable objects in Java, `Vector` is thread-safe. The `Vector` class works directly with the type `Object`, so we can use them with instances of any kind of class.[3] We can even put different kinds of `Object`s in a `Vector` together; the `Vector` doesn't know the difference.

> [3] In C++, where classes don't derive from a single `Object` class that supplies a base type and common methods, the elements of a collection would usually be derived from some common collectable class. This forces the use of multiple inheritance and brings its associated problems.

As you might guess, this is where things get tricky. To do anything useful with an `Object` after we take it back out of a Vector, we have to cast it back (narrow) it to its original type. This can be done with safety in Java because the cast is checked at run-time. Java throws a ClassCastException if we try to cast an object to the wrong type. However, this need for casting means that your code must remember types or methodically test them with instanceof. That is the price we pay for having a completely dynamic collection class that operates on all types.

You might wonder if you can subclass Vector to produce a class that looks like a Vector, but that works on just one type of element in a type-safe way. Unfortunately, the answer is no. We could override Vector's methods to make a Vector that rejects the wrong type of element at run-time, but this does not provide any new compile-time, static type safety. In C++, templates provide a safe mechanism for parameterizing types by restricting the types of objects used at compile-time. The keyword generic is a reserved word in Java. This means that it's possible that future versions might support C++-style templates, using generic to allow statically checked parameterized types.

We can construct a Vector with default characteristics and add elements to it using addElement() and insertElement():

```
Vector things = new Vector();

String one = "one";
String two = "two";
String three = "three";

things.addElement( one );
things.addElement( three );
things.insertElementAt( two, 1 );
```

things now contains three String objects in the order "one," "two," and "three." We can retrieve objects by their position with elementAt(), firstElement(), and lastElement():

```
String s1 = (String)things.firstElement();       // "one"
String s3 = (String)things.lastElement();        // "three"
String s2 = (String)things.elementAt(1);         // "two"
```

We have to cast each Object back to a String in order to assign it a String reference. ClassCastException is a type of RuntimeException, so we can neglect to guard for the exception if we are feeling confident about the type we are retrieving. Often, as in this example, you'll just have one type of object in the Vector. If we were unsure about the types of objects we were retrieving, we would want to be prepared to catch the ClassCastException or test the type explicitly with the instanceof operator.

We can search for an item in a Vector with the indexOf() method:

```
int i = things.indexOf( three );                 // i = 2
```

indexOf() returns a value of -1 if the object is not found. As a convenience, we can also use contains() simply to test for the presence of the object.

Finally, removeElement() removes a specified Object from the Vector:

```
things.removeElement( two );
```

The element formerly at position three now becomes the second element.

The size() method reports the number of objects currently in the Vector. You might think of using this to loop through all elements of a Vector, using elementAt() to get at each element. This works just fine, but there is a more general way to operate on a complete set of elements like those in a Vector.

java.util.Enumeration

The java.util.Enumeration interface can be used by any sort of set to provide serial access to its elements. An object that implements the Enumeration interface presents two methods: nextElement() and hasMoreElements(). nextElement() returns an Object type, so it can be used with any kind of collection. As with taking objects from a Vector, you need to know or determine what the objects are and cast them to the appropriate types before using them.

Enumeration is useful because any type of object can implement the interface and then use it to provide access to its elements. If you have an object that handles a set of values, you should think about implementing the Enumeration interface. Simply provide a hasMoreElements() test and a nextElement() iterator and declare that your class implements java.util.Enumeration. One advantage of an Enumeration is that you don't have to provide all values up front; you can provide each value as it's requested with nextElement(). And since Enumeration is an interface, you can write general routines that operate on all of the elements Enumeration.

An Enumeration does not guarantee the order in which elements are returned, however, so if order is important you don't want to use an Enumeration. You can iterate through the elements in an Enumeration only once; there is no way to reset it to the beginning or move backwards through the elements.

A Vector returns an Enumeration of its contents when we call the elements() method:

```
Enumeration e = things.elements();
```

```
while ( e.hasMoreElements() ) {
    String s = (String)e.nextElement();
    System.out.println( s ):
}
```

The above code loops three times, as call `nextElement()`, to fetch our strings. The actual type of object returned by `elements()` is a `VectorEnumeration`, but we don't have to worry about that. We can always refer to an `Enumeration` simply by its interface.

Note that `Vector` does not implement the `Enumeration` interface. If it did, that would put a serious limitation on `Vector` because we could cycle through the elements in it only once. That's clearly not the purpose of a `Vector`, which is why `Vector` instead provides a method that returns an `Enumeration`.

java.util.Hashtable

As I said earlier, a hashtable is a dictionary, similar to an associative array. A hashtable stores and retrieves elements with key values; they are very useful for things like caches and minimalist databases. When you store a value in a hashtable, you associate a key with that value. When you need to look up the value, the hashtable retrieves it efficiently using the key. The name hashtable itself refers to how the indexing and storage of elements is performed, as we'll discuss shortly. First I want to talk about how to use a hashtable.

The `java.util.Hashtable` class implements a hashtable that, like `Vector`, operates on the type `Object`. A `Hashtable` stores an element of type `Object` and associates it with a key, also of type `Object`. In this way, we can index arbitrary types of elements using arbitrary types as keys. As with `Vector`, casting is generally required to narrow objects back to their original type after pulling them out of a hashtable.

A `Hashtable` is quite easy to use. We can use the `put()` method to store items:

```
Hashtable dates = new Hashtable();

dates.put( "christmas",
    new GregorianCalendar( 1997, Calendar.DECEMBER, 25 ) );
dates.put( "independence",
    new GregorianCalendar( 1997, Calendar.JULY, 4 ) );
dates.put( "groundhog",
    new GregorianCalendar( 1997, Calendar.FEBRUARY, 2 ) );
```

First we create a new `Hashtable`. Then we add three `GregorianCalendar` objects to the hashtable, using `String` objects as keys. The key is the first argument to `put()`; the value is the second. Only one value can be stored per key. If we try to store a second object under a key that already exists in the `Hashtable`, the old element is booted out and replaced by the new one. The return value of the `put()` method is normally `null`, but if the call to `put()` results in replacing an element, the method instead returns the old stored `Object`.

We can now use the `get()` method to retrieve each of the above dates by name, using the `String` key by which it was indexed:

```
GregorianCalendar g = (GregorianCalendar)dates.get( "christmas" );
```

The `get()` method returns a `null` value if no element exists for the given key. The cast is required to narrow the returned object back to type `GregorianCalendar`. I hope you can see the advantage of using a `Hashtable` over a regular array. Each value is indexed by a key instead of a simple number, so unlike a simple array, we don't have to remember where each `GregorianCalendar` is stored.

Once we've put a value in a `Hashtable`, we can take it back out with the `remove()` method, again using the key to access the value:

```
dates.remove("christmas");
```

We can test for the existence of a key with `containsKey()`:

```
if ( dates.containsKey( "groundhog" ) ) {    // yes
```

Just like with a `Vector`, we're dealing with a set of items. Actually, we're dealing with two sets: keys and values. The `Hashtable` class has two methods, `keys()` and `elements()`, for getting at these sets. The `keys()` method returns an `Enumeration` of the keys for all of the elements in the `Hashtable`. We can use this `Enumeration` to loop through all of the keys:

```
for (Enumeration e = dates.keys(); e.hasMoreElements(); ) {
    String key = (String)e.nextElement();
    ...
}
```

Similarly, `elements()` provides an `Enumeration` of the elements themselves.

Hashcodes and key values

If you've used a hashtable before, you've probably guessed that there's more going on behind the scenes than I've let on so far. An element in a hashtable is not associated with its key by identity, but by something called a *hashcode*. Every object in Java has an identifying hashcode value determined by its `hashCode()` method, which is inherited from the `Object` class. When you store an element in a hashtable, the hashcode of the key object registers the element internally. Later, when you retrieve the item, that same hashcode looks it up efficiently.

A hashcode is usually a random-looking integer value based on the contents of an object, so it's different for different instances of a class. Two objects that have different hashcodes serve as unique keys in a hashtable; each object can reference a different stored object. Two objects that have the same hashcode value, on the other hand, appear to a hashtable as the same key. They can't coexist as keys to different objects in the hashtable.

Generally, we want our object instances to have unique hash codes, so we can put arbitrary items in a hashtable and index them with arbitrary keys. The default `hashCode()` method in the `Object` class simply assigns each object instance a unique number to be used as a hashcode. If a class does not override this method, each instance of the class will have a unique hashcode. This is sufficient for most objects.

However, it's also useful to allow equivalent objects to serve as equivalent keys. `String` objects provide a good example of this case. Although Java does its best to consolidate them, a literal string that appears multiple times in Java source code is often represented by different `String` objects at run-time. If each of these `String` objects has a different hash code, even though the literal value is the same, we could not use strings as keys in a hashtable, like we did the in above examples.

The solution is to ensure that equivalent `String` objects return the same hashcode value so that they can act as equivalent keys. The `String` class overrides the default `hashCode()` method so that equivalent `String` objects return the same hash code, while different `String` objects have unique hashcodes. This is possible because `String` objects are immutable; the contents can't change, so neither can the hashcode.

A few other classes in the Java API also override the default `hashCode()` method in order to provide equivalent hashcodes for equivalent objects. For example, each of the primitive wrapper classes provides a `hashCode()` method for this purpose. Other objects likely to be used as hashtable keys, such as `Color`, `Date`, `File`, and `URL`, also implement their own `hashCode()`methods.

So now maybe you're wondering when you need to override the default `hashCode()` method in your objects. If you're creating a class to use for keys in a hashtable, think about whether the class supports the idea of "equivalent objects." If so, you should implement a `hashCode()` method that returns the same hashcode value for equivalent objects.

To accomplish this, you need to define the hashcode of an object to be some suitably complex and arbitrary function of the contents of that object. The only criterion for the function is that it should be almost certain to provide different values for different contents of the object. Because the capacity of an integer is limited, hashcode values are not guaranteed to be unique. This limitation is not normally a problem though, as there are 2^32 possible hashcodes to choose from. The more sensitive the hashcode function is to small differences in the contents the better. A hashtable works most efficiently when the hashcode values are as randomly and evenly distributed as possible. As an example, you could produce a hashcode for a `String` object by adding the character values at each position in the string and multiplying the result by some number, producing a large random-looking integer.

java.util.Dictionary

`java.util.Dictionary` is the `abstract` superclass of `Hashtable`. It lays out the basic `get()`, `put()`, and `remove()` functionality for dictionary-style collections. You could derive other types of dictionaries from this class. For example, you could implement a dictionary with a different storage format, such as a binary tree.

7.5 Properties

The `java.util.Properties` class is a specialized hashtable for strings. Java uses the `Properties` object to replace the environment variables used in other programming environments. You can use a `Properties` table to hold arbitrary configuration information for an application in an easily accessible format. The `Properties` object can also load and store information using streams (see Chapter 8, *Input/Output Facilities* for information on streams).

Any string values can be stored as key/value pairs in a `Properties` table. However, the convention is to use a dot-separated naming hierarchy to group property names into logical structures, as is done with X resources on UNIX systems.[4] The `java.lang.System` class provides system-environment information in this way, through a system `Properties` table I'll describe shortly.

Create an empty `Properties` table and add `String` key/value pairs just as with any `Hashtable`:

```
Properties props = new Properties();
props.put("myApp.xsize", "52");
props.put("myApp.ysize", "79");
```

Thereafter, you can retrieve values with the `getProperty()`method:

```
String xsize = props.getProperty( "myApp.xsize" );
```

If the named property doesn't exist, `getProperty()` returns `null`. You can get an `Enumeration` of the property names with the `propertyNames()` method:

```
for ( Enumeration e = props.propertyNames(); e.hasMoreElements; ) {
    String name = e.nextElement();
```

[4] Unfortunately, this is just a naming convention right now, so you can't access logical groups of properties as you can with X resources.

```
    ...
}
```

Default Values

When you create a `Properties` table, you can specify a second table for default property values:

```
Properties defaults;
...
Properties props = new Properties( defaults );
```

Now when you call `getProperty()`, the method searches the default table if it doesn't find the named property in the current table. An alternative version of `getProperty()` also accepts a default value; this value is returned if the property is not found in the current list or in the default list:

```
String xsize = props.getProperty( "myApp.xsize", "50" );
```

Loading and Storing

You can save a `Properties` table to an `OutputStream` using the `save()` method. The property information is output in flat ASCII format. Continuing with the above example, output the property information to `System.out` as follows:

```
props.save( System.out, "Application Parameters" );
```

As we'll discuss in Chapter 8, *Input/Output Facilities*, `System.out` is a standard output stream similar to C's `stdout`. We could also save the information to a file by using a `FileOutputStream` as the first argument to `save()`. The second argument to `save()` is a `String` that is used as a header for the data. The above code outputs something like the following to `System.out`:

```
#Application Parameters
#Mon Feb 12 09:24:23 CST 1997
myApp.ysize=79
myApp.xsize=52
```

The `load()` method reads the previously saved contents of a `Properties` object from an `InputStream`:

```
FileInputStream fin;
...
Properties props = new Properties()
```

```
props.load( fin );
```

The `list()` method is useful for debugging. It prints the contents to an `OutputStream` in a format that is more human-readable but not retrievable by `load()`.

System Properties

The `java.lang.System` class provides access to basic system environment information through the `static System.getProperty()` method. This method returns a `Properties` table that contains system properties. System properties take the place of environment variables in other programming environments.

Table 7.7 summarizes system properties that are guaranteed to be defined in any Java environment.

Table 7.7: System Properties

System Property	**Meaning**
`java.vendor`	Vendor-specific string
`java.vendor.url`	URL of vendor
`java.version`	Java version
`java.home`	Java installation directory
`java.class.version`	Java class version
`java.class.path`	The class path
`os.name`	Operating-system name
`os.arch`	Operating-system architecture
`os.version`	Operating-system version
`file.separator`	File separator (such as "/" or " \")
`path.separator`	Path separator (such as ":" or ";")
`line.separator`	Line separator (such as "\n" or "\r\n")
`user.name`	User account name
`user.home`	User's home directory
`user.dir`	Current working directory

Applets are, by current Web browser conventions, prevented from reading the following properties: `java.home`, `java.class.path`, `user.name`, `user.home`, and `user.dir`. As you'll see in the next section, these restrictions are implemented by a `SecurityManager` object.

PREVIOUS
Vectors and Hashtables

HOME
BOOK INDEX

NEXT
The Security Manager

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

7.6 The Security Manager

As I described in Chapter 1, *Yet Another Language?*, a Java application's access to system resources, such as the display, the filesystem, threads, external processes, and the network, can be controlled at a single point with a *security manager*. The class that implements this functionality in the Java API is the `java.lang.SecurityManager` class.

An instance of the `SecurityManager` class can be installed once, and only once, in the life of the Java run-time environment. Thereafter, every access to a fundamental system resource is filtered through specific methods of the `SecurityManager` object by the core Java packages. By installing a specialized `SecurityManager`, we can implement arbitrarily complex (or simple) security policies for allowing access to individual resources.

When the Java run-time system starts executing, it's in a wide-open state until a `SecurityManager` is installed. The "null" security manager grants all requests, so the Java virtual environment can perform any activity with the same level of access as other programs running under the user's authority. If the application that is running needs to ensure a secure environment, it can install a `SecurityManager` with the `static System.setSecurityManager()` method. For example, a Java-enabled Web browser like Netscape Navigator installs a `SecurityManager` before it runs any Java applets.

`java.lang.SecurityManager` must be subclassed to be used. This class does not actually contain any `abstract` methods; it's `abstract` as an indication that its default implementation is not very useful. By default, each security method in `SecurityManager` is implemented to provide the strictest level of security. In other words, the default `SecurityManager` simply rejects all requests.

The following example, `MyApp`, installs a trivial subclass of `SecurityManager` as one of its first activities:

```
class FascistSecurityManager extends SecurityManager { }

public class MyApp {
    public static void main( Strings [] args ) {
        System.setSecurityManager( new FascistSecurityManager() );
        // No access to files, network, windows, etc.
        ...
    }
}
```

In the above scenario, `MyApp` does little aside from reading from `System.in` and writing to `System.out`. Any attempts to read or write files, access the network, or even open an window, results in a `SecurityException` being thrown.

After this draconian `SecurityManager` is installed, it's impossible to change the `SecurityManager` in any way. The security of this feature is not dependent on the `SecurityManager`; you can't replace or modify the `SecurityManager` under any circumstances. The upshot of this is that you have to install one that handles all your needs up front.

To do something more useful, we can override the methods that are consulted for access to various kinds of resources. Table 7.7 lists some of the more important access methods. You should not normally have to call these methods yourself, although you could. They are called by the core Java classes before granting particular types of access.

Table 7.8: SecurityManager Methods

Method	**Can I...?**
`checkAccess(Thread g)`	Access this thread?
`checkExit(int status)`	Execute a `System.exit()`?
`checkExec(String cmd)`	`exec()` this process?
`checkRead(String file)`	Read a file?
`checkWrite(String file)`	Write a file?
`checkDelete(String file)`	Delete a file?
`checkConnect(String host, int port)`	Connect a socket to a host?
`checkListen(int port)`	Create a server socket?
`checkAccept(String host, int port)`	Accept this connection?
`checkPropertyAccess(String key)`	Access this system property?
`checkTopLevelWindow(Object window)`	Create this new top-level window?

All these methods, with the exception of `checkTopLevelWindow()`, simply return to grant access. If access is not granted, they throw a `SecurityException`. `checkTopLevelWindow()` returns a `boolean` value. A value of `true` indicates the access is granted; a value of `false` indicates the access is granted with the restriction that the new window should provide a warning border that serves to identify it as an untrusted window.

Let's implement a silly `SecurityManager` that allows only files beginning with the name *foo* to be read:

```
class  FooFileSecurityManager extends SecurityManager {

    public void checkRead( String s ) {
        if ( !s.startsWith("foo") )
            throw new SecurityException("Access to non-foo file: " +
                s + " not allowed." );
    }
}
```

Once the `FooFileSecurityManager` is installed, any attempt to read a filename other than *foo** from any class will fail and cause a `SecurityException` to be thrown. All other security methods are inherited from `SecurityManager`, so they are left at their default restrictiveness.

All restrictions placed on applets by an applet-viewer application are enforced through a `SecurityManager`, which allows untrusted code loaded from over the network to be executed safely. The restrictions placed on applets are currently fairly harsh. As time passes and security considerations related to applets are better understood and accepted, the applet API will hopefully become more powerful and allow forms of persistence and access to designated public information.

PREVIOUS | HOME | NEXT
Properties | BOOK INDEX | Internationalization

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

8.2 Files

Unless otherwise restricted, a Java application can read and write to the host filesystem with the same level of access as the user who runs the Java interpreter. Java applets and other kinds of networked applications can, of course, be restricted by the `SecurityManager` and cut off from these services. We'll discuss applet access at the end of this section. First, let's take a look at the tools for basic file access.

Working with files in Java is still somewhat problematic. The host filesystem lies outside of Java's virtual environment, in the real world, and can therefore still suffer from architecture and implementation differences. Java tries to mask some of these differences by providing information to help an application tailor itself to the local environment; I'll mention these areas as they occur.

java.io.File

The `java.io.File` class encapsulates access to information about a file or directory entry in the filesystem. It gets attribute information about a file, lists the entries in a directory, and performs basic filesystem operations like removing a file or making a directory. While the `File` object handles these tasks, it doesn't provide direct access for reading and writing file data; there are specialized streams for that purpose.

File constructors

You can create an instance of `File` from a `String` pathname as follows:

```
File fooFile = new File( "/tmp/foo.txt" );
File barDir = new File( "/tmp/bar" );
```

You can also create a file with a relative path like:

```
File f = new File( "foo" );
```

In this case, Java works relative to the current directory of the Java interpreter. You can determine the current directory by checking the `user.dir` property in the `System Properties` list (`System.getProperty('user.dir')`).

An overloaded version of the `File` constructor lets you specify the directory path and filename as separate `String` objects:

```
File fooFile = new File( "/tmp", "foo.txt" );
```

With yet another variation, you can specify the directory with a `File` object and the filename with a `String`:

```
File tmpDir = new File( "/tmp" );
File fooFile = new File ( tmpDir, "foo.txt" );
```

None of the `File` constructors throw any exceptions. This means the object is created whether or not the file or directory actually exists; it isn't an error to create a `File` object for an nonexistent file. You can use the `exists()` method to find out whether the file or directory exists.

Path localization

One of the reasons that working with files in Java is problematic is that pathnames are expected to follow the conventions of the local filesystem. Java's designers intend to provide an abstraction that deals with most system-dependent filename features, such as the file separator, path separator, device specifier, and root directory. Unfortunately, not all of these features are implemented in the current version.

On some systems, Java can compensate for differences such as the direction of the file separator slashes in the above string. For example, in the current implementation on Windows platforms, Java accepts paths with either forward slashes or backslashes. However, under Solaris, Java accepts only paths with forward slashes.

Your best bet is to make sure you follow the filename conventions of the host filesystem. If your application is just opening and saving files at the user's request, you should be able to handle that functionality with the `java.awt.FileDialog` class. This class encapsulates a graphical file-selection dialog box. The methods of the `FileDialog` take care of system-dependent filename features for you.

If your application needs to deal with files on its own behalf, however, things get a little more complicated. The `File` class contains a few `static` variables to make this task easier. `File.separator` defines a `String` that specifies the file separator on the local host (e.g., "/" on UNIX and Macintosh systems and "\" on Windows systems), while `File.separatorChar` provides the same information in character form. `File.pathSeparator` defines a `String` that separates items in a path (e.g., ":" on UNIX systems; ";" on Macintosh and Windows systems); `File.pathSeparatorChar` provides the information in character form.

You can use this system-dependent information in several ways. Probably the simplest way to localize pathnames is to pick a convention you use internally, say "/", and do a `String` replace to substitute for the localized separator character:

```
// We'll use forward slash as our standard
String path = "mail/1995/june/merle";
path = path.replace('/', File.separatorChar);
File mailbox = new File( path );
```

Alternately, you could work with the components of a pathname and built the local pathname when you need it:

```
String [] path = { "mail", "1995", "june", "merle" };
```

```
StringBuffer sb = new StringBuffer(path[0]);
for (int i=1; i< path.length; i++)
    sb.append( File.separator + path[i] );
  File mailbox = new File( sb.toString() );
```

One thing to remember is that Java interprets the backslash character (\) as an escape character when used in a `String`. To get a backslash in a `String`, you have to use " \\".

File methods

Once we have a valid `File` object, we can use it to ask for information about the file itself and to perform standard operations on it. A number of methods lets us ask certain questions about the `File`. For example, `isFile()` returns `true` if the `File` represents a file, while `isDirectory()` returns `true` if it's a directory. `isAbsolute()` indicates whether the `File` has an absolute or relative path specification.

The components of the `File` pathname are available through the following methods: `getName()`, `getPath()`, `getAbsolutePath()`, and `getParent()`. `getName()` returns a `String` for the filename without any directory information; `getPath()` returns the directory information without the filename. If the `File` has an absolute path specification, `getAbsolutePath()` returns that path. Otherwise it returns the relative path appended to the current working directory. `getParent()` returns the parent directory of the `File`.

Interestingly, the string returned by `getPath()` or `getAbsolutePath()` may not be the same, case-wise, as the actual name of the file. You can retrieve the case-correct version of the file's path using `getCanonicalPath()`. In Windows 95, for example, you can create a `File` object whose `getAbsolutePath()` is `C:\Autoexec.bat` but whose `getCanonicalPath()` is `C:\AUTOEXEC.BAT`.

We can get the modification time of a file or directory with `lastModified()`. This time value is not useful as an absolute time; you should use it only to compare two modification times. We can also get the size of the file in bytes with `length()`. Here's a fragment of code that prints some information about a file:

```
File fooFile = new File( "/tmp/boofa" );

String type = fooFile.isFile() ? "File " : "Directory ";
String name = fooFile.getName();
long len = fooFile.length();
System.out.println(type + name + ", " + len + " bytes " );
```

If the `File` object corresponds to a directory, we can list the files in the directory with the `list()` method:

```
String [] files = fooFile.list();
```

`list()` returns an array of `String` objects that contains filenames. (You might expect that `list()` would return an `Enumeration` instead of an array, but it doesn't.)

If the `File` refers to a nonexistent directory, we can create the directory with `mkdir()` or `mkdirs()`. `mkdir()` creates a single directory; `mkdirs()` creates all of the directories in a `File` specification. Use `renameTo()` to rename a file or directory and `delete()` to delete a file or directory. Note that `File` doesn't provide a method to create a file; creation is handled with a `FileOutputStream` as we'll discuss in a moment.

Table 8.1 summarizes the methods provided by the `File` class.

Table 8.1: File Methods

Method	**Return type**	**Description**
`canRead()`	`boolean`	Is the file (or directory) readable?
`canWrite()`	`boolean`	Is the file (or directory) writable?
`delete()`	`boolean`	Deletes the file (or directory)
`exists()`	`boolean`	Does the file (or directory) exist?
`getAbsolutePath()`	`String`	Returns the absolute path of the file (or directory)
`getCanonicalPath()`	`String`	Returns the absolute, case-correct path of the file (or directory)
`getName()`	`String`	Returns the name of the file (or directory)
`getParent()`	`String`	Returns the name of the parent directory of the file (or directory)
`getPath()`	`String`	Returns the path of the file (or directory)
`isAbsolute()`	`boolean`	Is the filename (or directory name) absolute?
`isDirectory()`	`boolean`	Is the item a directory?
`isFile()`	`boolean`	Is the item a file?
`lastModified()`	`long`	Returns the last modification time of the file (or directory)
`length()`	`long`	Returns the length of the file
`list()`	`String []`	Returns a list of files in the directory
`mkdir()`	`boolean`	Creates the directory
`mkdirs()`	`boolean`	Creates all directories in the path
`renameTo(File dest)`	`boolean`	Renames the file (or directory)

File Streams

Java provides two specialized streams for reading and writing files in the filesystem: `FileInputStream` and `FileOutputStream`. These streams provide the basic `InputStream` and `OutputStream` functionality applied to reading and writing the contents of files. They can be combined with the filtered streams described earlier to work with files in the same way we do other stream communications.

Because `FileInputStream` is a subclass of `InputStream`, it inherits all standard `InputStream` functionality for reading the contents of a file. `FileInputStream` provides only a low-level interface to reading data, however, so you'll typically wrap another stream like a `DataInputStream` around the `FileInputStream`.

You can create a `FileInputStream` from a `String` pathname or a `File` object:

```
FileInputStream foois = new FileInputStream( fooFile );
FileInputStream passwdis = new FileInputStream( "/etc/passwd" );
```

When you create a `FileInputStream`, Java attempts to open the specified file. Thus, the `FileInputStream` constructors can throw a `FileNotFoundException` if the specified file doesn't exist, or an `IOException` if some other I/O error occurs. You should be sure to catch and handle these exceptions in your code. When the stream is first created, its `available()` method and the `File` object's `length()` method should return the same value. Be sure to call the `close()` method when you are done with the file.

To read characters from a file, you can wrap an `InputStreamReader` around a `FileInputStream`. If you want to use the default character encoding scheme, you can use the `FileReader` class instead, which is provided as a convenience. `FileReader` works just like `FileInputStream`, except that it reads characters instead of bytes and wraps a `Reader` instead of an `InputStream`.

The following class, `ListIt`, is a small utility that displays the contents of a file or directory to standard output:

```
import java.io.*;

class ListIt {
    public static void main ( String args[] ) throws Exception {
        File file =  new File( args[0] );

        if ( !file.exists() || !file.canRead() ) {
            System.out.println( "Can't read " + file );
            return;
        }

        if ( file.isDirectory() ) {
            String [] files = file.list();
            for (int i=0; i< files.length; i++)
                System.out.println( files[i] );
        }
        else
            try {
                FileReader fr = new FileReader ( file );
                BufferedReader in = new BufferedReader( fr );
                String line;
                while ((line = in.readLine()) != null)
                    System.out.println(line);
            }
            catch ( FileNotFoundException e ) {
                System.out.println( "File Disappeared" );
            }
    } }
```

`ListIt` constructs a `File` object from its first command-line argument and tests the `File` to see if it exists and is readable. If the `File` is a directory, `ListIt` prints the names of the files in the directory. Otherwise, `ListIt` reads and prints the file.

FileOutputStream is a subclass of OutputStream, so it inherits all the standard OutputStream functionality for writing to a file. Just like FileInputStream though, FileOutputStream provides only a low-level interface to writing data. You'll typically wrap another stream like a DataOutputStream or a PrintStream around the FileOutputStream to provide higher-level functionality. You can create a FileOutputStream from a String pathname or a File object. Unlike FileInputStream however, the FileOutputStream constructors don't throw a FileNotFoundException. If the specified file doesn't exist, the FileOutputStream creates the file. The FileOutputStream constructors can throw an IOException if some other I/O error occurs, so you still need to handle this exception.

If the specified file does exist, the FileOutputStream opens it for writing. When you actually call a write() method, the new data overwrites the current contents of the file. If you need to append data to an existing file, you should use a different constructor that accepts an append flag, as shown here:

```
FileInputStream foois = new FileInputStream( fooFile );
FileInputStream passwdis = new FileInputStream( "/etc/passwd" );
```

Antoher alternative for appending files is to use a RandomAccessFile, as I'll discuss shortly.

To write characters (instead of bytes) to a file, you can wrap an OutputStreamWriter around a FileOutputStream. If you want to use the default character encoding scheme, you can use the FileWriter class instead, which is provided as a convenience. FileWriter works just like FileOutputStream, except that it writes characters instead of bytes and wraps a Writer instead of an OutputStream.

The following example reads a line of data from standard input and writes it to the file */tmp/foo.txt*:

```
String s = new BufferedReader( new InputStreamReader( System.in ) ).readLine();

File out = new File( "/tmp/foo.txt" );
FileWriter fw = new FileWriter ( out );
PrintWriter pw = new PrintWriter( fw, true )
pw.println( s );
```

Notice how we have wrapped a PrintWriter around the FileWriter to facilitate writing the data. To be a good filesystem citizen, you need to call the close() method when you are done with the FileWriter.

java.io.RandomAccessFile

The java.io.RandomAccessFile class provides the ability to read and write data from or to any specified location in a file. RandomAccessFile implements both the DataInput and DataOutput interfaces, so you can use it to read and write strings and Java primitive types. In other words, RandomAccessFile defines the same methods for reading and writing data as DataInputStream and DataOutputStream. However, because the class provides random, rather than sequential, access to file data, it's not a subclass of either InputStream or OutputStream.

You can create a RandomAccessFile from a String pathname or a File object. The constructor also takes a second String argument that specifies the mode of the file. Use "r" for a read-only file or "rw" for a read-write file. Here's how to create a simple database to keep track of user information:

```
try {
    RandomAccessFile users = new RandomAccessFile( "Users", "rw" );
    ...
}
catch (IOException e) {
}
```

When you create a RandomAccessFile in read-only mode, Java tries to open the specified file. If the file doesn't exist, RandomAccessFile throws an IOException. If, however, you are creating a RandomAccessFile in read-write mode, the object creates the file if it doesn't exist. The constructor can still throw an IOException if some other I/O error occurs, so you still need to handle this exception.

After you have created a RandomAccessFile, call any of the normal reading and writing methods, just as you would with a DataInputStream or DataOutputStream. If you try to write to a read-only file, the write method throws an IOException.

What makes a RandomAccessFile special is the seek() method. This method takes a long value and uses it to set the location for reading and writing in the file. You can use the getFilePointer() method to get the current location. If you need to append data on the end of the file, use length() to determine that location. You can write or seek beyond the end of a file, but you can't read beyond the end of a file. The read methods throws a EOFException if you try to do this.

Here's an example of writing some data to our user database:

```
users.seek( userNum * RECORDSIZE );
users.writeUTF( userName );
users.writeInt( userID );
```

One caveat to notice with this example is that we need to be sure that the String length for userName, along with any data that comes after it, fits within the boundaries of the record size.

Applets and Files

For security reasons, applets are not permitted to read and write to arbitrary places in the filesystem. The ability of an applet to read and write files, as with any kind of system resource, is under the control of a SecurityManager object. A SecurityManager is installed by the application that is running the applet, such as an applet viewer or Java-enabled Web browser. All filesystem access must first pass the scrutiny of the SecurityManager. With that in mind, applet-viewer applications are free to implement their own schemes for what, if any, access an applet may have.

For example, Sun's HotJava Web browser allows applets to have access to specific files designated by the user in an access-control list. Netscape Navigator, on the other hand, currently doesn't allow applets any access to the filesystem.

It isn't unusual to want an applet to maintain some kind of state information on the system where it's running. But for a Java applet that is restricted from access to the local filesystem, the only option is to store data over the network on its server. Although, at the moment, the Web is a relatively static, read-only environment, applets have at their disposal powerful, general means for communicating data over networks, as you'll see in Chapter 9, *Network Programming*. The only limitation is that, by convention, an applet's network communication is restricted to the server that launched it. This limits the options for where the data will reside.

The only means of writing data to a server in Java is through a network socket. In Chapter 9, *Network Programming* we'll look at building networked applications with sockets in detail. With the tools of that chapter it's possible to build powerful client/server applications.

PREVIOUS	HOME	NEXT
Streams	BOOK INDEX	Serialization

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

9.3 Working with URLs

A URL points to an object on the Internet. It's a collection of information that identifies an item, tells you where to find it, and specifies a method for communicating with it or retrieving it from its source. A URL refers to any kind of information source. It might point to static data, such as a file on a local filesystem, a Web server, or an FTP archive; or it can point to a more dynamic object such as a news article on a news spool or a record in a WAIS database. URLs can even refer to less tangible resources such as Telnet sessions and mailing addresses.

A URL is usually presented as a string of text, like an address.[3] Since there are many different ways to locate an item on the Net, and different mediums and transports require different kinds of information, there are different formats for different kinds of URLs. The most common form specifies three things: a network host or server, the name of the item and its location on that host, and a protocol by which the host should communicate:

> [3] The term URL was coined by the Uniform Resource Identifier (URI) working group of the IETF to distinguish URLs from the more general notion of Uniform Resource Names or URNs. URLs are really just static addresses, whereas URNs would be more persistent and abstract identifiers used to resolve the location of an object anywhere on the Net. URLs are defined in RFC 1738 and RFC 1808.

```
protocol://hostname/location/item
```

`protocol` is an identifier such as "http," "ftp," or "gopher"; *`hostname`* is an Internet hostname; and the *`location`* and *`item`* components form a path that identifies the object on that host. Variants of this form allow extra information to be packed into the URL, specifying things like port numbers for the communications protocol and fragment identifiers that reference parts inside the object.

We sometimes speak of a URL that is relative to a base URL. In that case we are using the base URL as a starting point and supplying additional information. For example, the base URL might point to a directory on a Web server; a relative URL might name a particular file in that directory.

The URL class

A URL is represented by an instance of the `java.net.URL` class. A `URL` object manages all information in a URL string and provides methods for retrieving the object it identifies. We can construct a `URL` object from a

URL specification string or from its component parts:

```
try {
    URL aDoc = new URL( "../../../../foo.bar.com/documents/homepage.html" );
    URL sameDoc =
        new URL("http","foo.bar.com","documents/homepage.html");
}
catch ( MalformedURLException e ) { }
```

The two `URL` objects above point to the same network resource, the *homepage.html* document on the server foo.bar.com. Whether or not the resource actually exists and is available isn't known until we try to access it. At this point, the `URL` object just contains data about the object's location and how to access it. No connection to the server has been made. We can examine the `URL`'s components with the `getProtocol()`, `getHost()`, and `getFile()` methods. We can also compare it to another `URL` with the `sameFile()` method. `sameFile()` determines if two URLs point to the same resource. It can be fooled, but `sameFile` does more than compare the URLs for equality; it takes into account the possibility that one server may have several names, and other factors.

When a `URL` is created, its specification is parsed to identify the protocol component. If the protocol doesn't make sense, or if Java can't find a protocol handler for it, the URL constructor throws a `MalformedURLException`. A protocol handler is a Java class that implements the communications protocol for accessing the URL resource. For example, given an "http" URL, Java prepares to use the HTTP protocol handler to retrieve documents from the specified server.

Stream Data

The most general way to get data back from `URL` is to ask for an `InputStream` from the `URL` by calling `openStream()`. If you're writing an applet that will be running under Netscape, this is about your only choice. In fact, it's a good choice if you want to receive continuous updates from a dynamic information source. The drawback is that you have to parse the contents of an object yourself. Not all types of URLs support the `openStream()` method; you'll get an `UnknownServiceException` if yours doesn't.

The following code reads a single line from an HTML file:

```
try {
    URL url = new URL("../../../../server/index.html");
    DataInputStream dis = new DataInputStream( url.openStream() );
    String line = dis.readLine();
```

We ask for an `InputStream` with `openStream()`, and wrap it in a `DataInputStream` to read a line of text. Here, because we are specifying the "http" protocol in the URL, we still require the services of an HTTP protocol handler. As we'll discuss more in a bit, that brings up some questions about what handlers we have available to us and where. This example partially works around those issues because no content handler is involved; we read the data and interpret it as a content handler would. However, there are even more limitations on what applets can do right now. For the time being, if you construct `URL`s relative to the applet's `codeBase()`, you should be able to use them in applets as in the above example. This should guarantee that the needed protocol is available and accessible to the applet. Again, we'll discuss the more general issues a bit later.

Getting the Content as an Object

`openStream()` operates at a lower level than the more general content-handling mechanism implemented by the `URL` class. We showed it first because, until some things are settled, you'll be limited as to when you can use URLs in their more powerful role. When a proper content handler is available to Java (currently, only if you supply one with your standalone application), you'll be able to retrieve the object the `URL` addresses as a complete object, by calling the `URL`'s `getContent()` method. `getContent()` initiates a connection to the host, fetches the data for you, determines the data's MIME type, and invokes a content handler to turn the data into a Java object.

For example: given the URL http://foo.bar.com/index.html, a call to `getContent()` uses the HTTP protocol handler to receive the data and the HTML content handler to turn the data into some kind of object. A URL that points to a plain-text file would use a text-content handler that might return a `String` object. A GIF file might be turned into an `Image` object for display, using a GIF content handler. If we accessed the GIF file using an "ftp" URL, Java would use the same content handler, but would use the FTP protocol handler to receive the data.

`getContent()` returns the output of the content handler. Now we're faced with a problem: exactly what did we get? Since the content handler can return almost anything, the return type of `getContent()` is `Object`. Before doing anything meaningful with this `Object`, we must cast it into some other data type that we can work with. For example, if we expect a `String`, we'll cast the result of `getContent()` to a `String`:

```
String content;

try
    content = (String)myURL.getContent();
catch ( Exception e ) { }
```

Of course, we are presuming we will in fact get a `String` object back from this `URL`. If we're wrong, we'll get a `ClassCastException`. Since it's common for servers to be confused (or even lie) about the MIME types of the objects they serve, it's wise to catch that exception (it's a subclass of `RuntimeException`, so catching it is optional) or to check the type of the returned object with the `instanceof` operator:

```
if ( content instanceof String ) {
    String s = (String)content;
    ...
```

Various kinds of errors can occur when trying to retrieve the data. For example, `getContent()` can throw an `IOException` if there is a communications error; `IOException` is not a type of `RuntimeException`, so we must catch it explicitly, or declare the method that calls `getContent()` can throw it. Other kinds of errors can happen at the application level: some knowledge of how the handlers deal with errors is necessary.

For example, consider a `URL` that refers to a nonexistent file on an HTTP server. When requested, the server probably returns a valid HTML document that consists of the familiar "404 Not Found" message. An appropriate

HTML content handler is invoked to interpret this and return it as it would any other HTML object. At this point, there are several alternatives, depending entirely on the content handler's implementation. It might return a `String` containing the error message; it could also conceivably return some other kind of object or throw a specialized subclass of `IOException`. To find out that an error occurred, the application may have to look directly at the object returned from `getContent()`. After all, what is an error to the application may not be an error as far as the protocol or content handlers are concerned. "404 Not Found" isn't an error at this level; it's a perfectly valid document.

Another type of error occurs if a content handler that understands the data's MIME type isn't available. In this case, `getContent()` invokes a minimal content handler used for data with an unknown type and returns the data as a raw `InputStream`. A sophisticated application might specialize this behavior to try to decide what to do with the data on its own.

The `openStream()` and `getContent()` methods both implicitly create a connection to the remote `URL` object. For some applications, it may be necessary to use the `openConnection()` method of the `URL` to interact directly with the protocol handler. `openConnection()` returns a `URLConnection` object, which represents a single, active connection to the `URL` resource. We'll examine `URLConnection`s further when we start writing protocol handlers.

PREVIOUS Datagram Sockets | HOME | BOOK INDEX | NEXT Web Browsers and Handlers

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

9.4 Web Browsers and Handlers

The content- and protocol-handler mechanisms I've introduced can be used by any application that accesses data via URLs. This mechanism is extremely flexible; to handle a URL, you need only the appropriate protocol and content handlers. To extend a Java-built Web browser so that it can handle new and specialized kinds of URLs, you need only supply additional content and protocol handlers. Furthermore, Java's ability to load new classes over the Net means that the handlers don't even need to be a part of the browser. Content and protocol handlers could be downloaded over the Net, from the same site that supplies the data, and used by the browser. If you wanted to supply some completely new data type, using a completely new protocol, you could make your data file plus a content handler and a protocol handler available on your Web server; anyone using a Web browser built in Java would automatically get the appropriate handlers whenever they access your data. In short, Java lets you build automatically extendible Web browsers; instead of being a gigantic do-everything application, the browser becomes a lightweight scaffold that dynamically incorporates extensions as needed. Figure 9.3 shows the conceptual operation of a content handler; Figure 9.4 does the same for a protocol handler.

Figure 9.3: A content handler at work

[Graphic: Figure 9-3]

Figure 9.4: A protocol handler at work

[Graphic: Figure 9-4]

Sun's HotJava was the first browser to demonstrate these features. When HotJava encounters a type of content or a protocol it doesn't understand, it searches the remote server for an appropriate handler class. HotJava also interprets HTML data as a type of content; that is, HTML isn't a privileged data type built into the browser. HTML documents use the same content- and protocol-handler mechanisms as other data types.

Unfortunately, a few nasty flies are stuck in this ointment. Content and protocol handlers are part of the Java API: they're an intrinsic part of the mechanism for working with URLs. However, specific content and protocol handlers aren't part of the API; the `ContentHandler` class and the two classes that make up protocol handlers, `URLStreamHandler` and `URLConnection`, are all abstract classes. They define what an implementation must provide, but they don't actually provide an implementation. This is not as paradoxical as it sounds. After all, the API defines the `Applet` class, but doesn't include any specific applets. Likewise, the standard Java classes don't include content handlers for HTML, GIF, MPEG, or other common data types. Even this isn't a real problem, although a library of standard content handlers would be useful. (JDK provides some content and protocol handlers in the `sun.net.www.content` and `sun.net.www.protocol` packages, but these are undocumented and subject to change.) There are two real issues here:

- There isn't any standard that tells you what kind of object the content handler should return. I danced around the issue just above, but it's a real problem. It's common sense that GIF data should be turned into an `Image` object, but at the moment, that's an application-level decision. If you're writing your own application and your own content handlers, that isn't an issue: you can make any decision you want. But if you're writing content handlers that interface to arbitrary Web browsers, you need a standard that defines what the browser expects. You can use the `sun.net` classes to make a guess, but a real standard hasn't been worked out yet.

- There isn't any standard that tells you where to put content and protocol handlers so that an

> application (like a Web browser) can find them. Again, you can make application-level decisions about where to place your own handlers, but that doesn't solve the real problem: we want our content and protocol handlers to be usable by any browser. It's possible to make an educated guess about what the standard will be, but it's still a guess.

The next release of Sun's HotJava Web browser should certainly take full advantage of handlers,[4] but current versions of Netscape Navigator do not. When the next release of HotJava appears, it may resolve these questions, at least on a de facto basis. (It would certainly be in Sun's interest to publish some kind of standard as soon as possible.) Although we can't tell you what standards will eventually evolve, we can discuss how to write handlers for standalone applications. When the standards issues are resolved, revising these handlers to work with HotJava and other Web browsers should be simple.

> [4] Downloadable handlers will be part of HotJava 1.0, though they are not supported by the "pre-beta 1" release. The current release *does* support local content and protocol handlers. HotJava 1.0 also promises additional classes to support network applications.

The most common Java-capable platform, Netscape Navigator, doesn't use the content- and protocol-handler mechanisms to render Net resources. It's a classic monolithic application: knowledge about certain kinds of objects, like HTML and GIF files, is built-in. It can be extended via a plug-in mechanism, but plug-ins aren't portable (they're written in C) and can't be downloaded dynamically over the Net. Applets running in Netscape can use a limited version of the `URL` mechanism, but the browser imposes many restrictions. As I said earlier, you can construct URLs relative to the applet's code base, and use the `openStream()` method to get a raw input stream from which to read data, but that's it. For the time being, you can't use your own content or protocol handlers to work with applets loaded over the Net. Allowing this would be a simple extension, even without content- and protocol-handler support integrated into Netscape itself. We can only hope they add this support soon.

9.5 Writing a Content Handler

`getContent()` invokes a content handler whenever it's called to retrieve an object at some URL. The content handler must read the flat stream of data produced by the `URL`'s protocol handler (the data read from the remote source), and construct a well-defined Java object from it. By "flat," I mean that the data stream the content handler receives has no artifacts left over from retrieving the data and processing the protocol. It's the protocol handler's job to fetch and decode the data before passing it along. The protocol handler's output is your data, pure and simple.

The roles of content and protocol handlers do not overlap. The content handler doesn't care how the data arrives, or what form it takes. It's concerned only with what kind of object it's supposed to create. For example, if a particular protocol involves sending an object over the network in a compressed format, the protocol handler should do whatever is necessary to unpack it before passing the data on to the content handler. The same content handler can then be used again with a completely different protocol handler to construct the *same* type of object received via a *different* transport mechanism.

Let's look at an example. The following lines construct a `URL` that points to a GIF file on an FTP archive and attempt to retrieve its contents:

```
try {
    URL url = new URL ("ftp://ftp.wustl.edu/graphics/gif/a/apple.gif");
    Image img = (Image)url.getContent();
    ...
```

When we construct the `URL` object, Java looks at the first part of the URL string (i.e., everything prior to the colon) to determine the protocol and locate a protocol handler. In this case, it locates the FTP protocol handler, which is used to open a connection to the host and transfer data for the specified file.

After making the connection, the `URL` object asks the protocol handler to identify the resource's MIME type.[5] It does this through a variety of means, but in this case it probably just looks at the filename extension (*.gif*) and determines that the MIME type of the data is `image/gif`. The protocol handler then looks for the content handler responsible for the `image/gif` type and uses it to construct the right kind of object from the data. The content handler returns an `Image` object, which `getContent()` returns to us as an `Object`; we cast this `Object` back to the `Image` type so we can work with it.

[5] MIME stands for Multipurpose Internet Mail Extensions. It's a standard design to facilitate multimedia email, but it has become more widely used as a way to specify the treatment of data contained in a document.

In an upcoming section, we'll build a simple content handler. To keep things as simple as possible, our example will produce text as output; the `URL`'s `getContent()` method will return this as a `String` object.

Locating Content Handlers

As I said earlier, there's no standard yet for where content handlers should be located. However, we're writing code now and need to know what package to place our class files in. In turn, this determines where to place the class files in the local filesystem. Because we are going to write our own standalone application to use our handler, we'll place our classes in a package in our local class path and tell Java where they reside. However, we will follow the naming scheme that's likely to become the standard. If other applications expect to find handlers in different locations (either locally or on servers), you'll simply have to repackage your class files according to their naming scheme and put them in the correct place.

Package names translate to path names when Java is searching for a class. This holds for locating content-handler classes as well as other kinds of classes. For example, on a UNIX- or DOS-based system, a class in a package named `net.www.content` would live in a directory with *net/www/content/* as part of its pathname. To allow Java to find handler classes for arbitrary new MIME types, content handlers are organized into packages corresponding to the basic MIME type categories. The handler classes themselves are then named after the specific MIME type. This allows Java to map MIME types directly to class names.

According to the scheme we'll follow, a handler for the `image/gif` MIME type is called `gif` and placed in a package called `net.www.content.image`. The fully qualified name of the class would then be `net.www.content.image.gif`, and it would be located in the file *net/www/content/image/gif.class*, somewhere in the local class path or on a server. Likewise, a content handler for the `video/mpeg` MIME type would be called `mpeg`, and there would be an *mpeg.class* file located (again, on a UNIX-/DOS-like filesystem) in a *net/www/content/video/* directory somewhere in a local class path or on a server.

Many MIME type names include a dash (`-`), which is illegal in a class name. You should convert dashes and other illegal characters into underscores when building Java class and package names. Also note that there are no capital letters in the class names. This violates the coding convention used in most Java source files, in which class names start with capital letters. However, capitalization is not significant in MIME type names, so it's simpler to name the handler classes accordingly. Table 9.1 shows how some typical MIME types are converted to package and class names.[6]

[6] The "pre-beta 1" release of HotJava has a temporary solution that is compatible with the convention described here. In the HotJava *properties* file, add the line: `java.content.handler.pkgs=net.www.content`.

Table 9.1: Converting MIME Types to Class and Package Names

MIME type	**Package name**	**Class name**	**Class location**
`image/gif`	`net.www.content.image`	`gif`	*net/www/content/image/*
`image/jpeg`	`net.www.content.image`	`jpeg`	*net/www/content/image/*
`text/html`	`net.www.content.text`	`html`	*net/www/content/text/*

The application/x-tar Handler

In this section, we'll build a simple content handler that reads and interprets *tar* (tape archive) files. *tar* is an archival format widely used in the UNIX-world to hold collections of files, along with their basic type and attribute information.[7] A *tar* file is similar to a ZIP file, except that it's not compressed. Files in the archive are stored sequentially, in flat text or binary with no special encoding. In practice, *tar* files are usually compressed for storage using an application like UNIX *compress* or GNU *gzip* and then named with a filename extension like *.tar.gz* or *.tgz*.

[7] There are several slightly different versions of the *tar* format. This content handler understands the most widely used variant.

Most Web browsers, upon retrieving a *tar* file, prompt the user with a **File Save** dialog. The assumption is that if you are retrieving an archive, you probably want to save it for later unpacking and use. We would like to instead implement a *tar* content handler that allows an application to read the contents of the archive and give us a listing of the files that it contains. In itself, this would not be the most useful thing in the world, because we would be left with the dilemma of how to get at the archive's contents. However, a more complete implementation of our content handler, used in conjunction with an application like a Web browser, could generate output that lets us select and save individual files within the archive.

The code that fetches the *.tar* file and lists its contents looks like this:

```
try {
    URL listing =
        new URL("../../../../somewhere.an.edu/lynx/lynx2html.tar");
    String s = (String)listing.getContents();
    System.out.println( s );
     ...
```

We'll produce a listing similar to the UNIX *tar* application's output:

```
Tape Archive Listing:

0     Tue Sep 28 18:12:47 CDT 1993 lynx2html/
14773 Tue Sep 28 18:01:55 CDT 1993 lynx2html/lynx2html.c
470   Tue Sep 28 18:13:24 CDT 1993 lynx2html/Makefile
172   Thu Apr 01 15:05:43 CST 1993 lynx2html/lynxgate
3656  Wed Mar 03 15:40:20 CST 1993 lynx2html/install.csh
490   Thu Apr 01 14:55:04 CST 1993 lynx2html/new_globals.c
...
```

Our content handler dissects the file to read the contents and generates the listing. The `URL`'s `getContent()` method returns that information to our application as a `String` object.

First we must decide what to call our content handler and where to put it. The MIME-type hierarchy classifies the *tar* format as an "application type extension." Its proper MIME type is then `application/x-tar`. Therefore, our handler belongs to the `net.www.content.application` package, and goes into the class file *net/www/content/application/x_tar.class*. Note that the name of our class is `x_tar`, rather than `x-tar`; you'll remember the dash is illegal in a class name so, by convention, we convert it to an underscore.

Here's the code for the content handler; compile it and place it in the *net/www/content/application/* package, somewhere in your class path:

```
package net.www.content.application;

import java.net.*;
import java.io.*;
import java.util.Date;

public class x_tar extends ContentHandler {
    static int
        RECORDLEN = 512,
        NAMEOFF = 0, NAMELEN = 100,
        SIZEOFF = 124, SIZELEN = 12,
        MTIMEOFF = 136, MTIMELEN = 12;

    public Object getContent(URLConnection uc) throws IOException {
        InputStream is = uc.getInputStream();
        StringBuffer output =
            new StringBuffer( "Tape Archive Listing:\n\n" );
        byte [] header = new byte[RECORDLEN];
        int count = 0;

        while ( (is.read(header) == RECORDLEN) &&
                (header[NAMEOFF] != 0) ) {

            String name =
               new String(header, 0, NAMEOFF, NAMELEN).trim();
            String s = new String(header, 0, SIZEOFF, SIZELEN).trim();
            int size = Integer.parseInt(s, 8);
            s = new String(header, 0, MTIMEOFF, MTIMELEN).trim();
            long l = Integer.parseInt(s, 8);
            Date mtime = new Date( l*1000 );

            output.append( size + " " + mtime + " " + name + "\n" );

            count += is.skip( size ) + RECORDLEN;
            if ( count % RECORDLEN != 0 )
                count += is.skip ( RECORDLEN - count % RECORDLEN);
        }

        if ( count == 0 )
            output.append("Not a valid TAR file\n");

        return( output.toString() );
    }
}
```

The ContentHandler class

Our `x_tar` handler is a subclass of the abstract class `java.net.ContentHandler`. Its job is to implement one method: `getContent()`, which takes as an argument a special "protocol connection" object and returns a constructed Java `Object`. The `getContent()` method of the `URL` class ultimately uses this `getContent()` method when we ask for the contents of the URL.

The code looks formidable, but most of it's involved with processing the details of the *tar* format. If we remove these details, there isn't much left:

```
public class x_tar extends ContentHandler {

    public Object getContent( URLConnection uc ) throws IOException {
        // get input stream
        InputStream is = uc.getInputStream();

        // read stream and construct object
        // ...

        // return the constructed object
        return( output.toString() );
    }
}
```

That's really all there is to a content handler; it's relatively simple.

The URLConnection

The `java.net.URLConnection` object that `getContent()` receives represents the protocol handler's connection to the remote resource. It provides a number of methods for examining information about the `URL` resource, such as header and type fields, and for determining the kinds of operations the protocol supports. However, its most important method is `getInputStream()`, which returns an `InputStream` from the protocol handler. Reading this `InputStream` gives you the raw data for the object the `URL` addresses. In our case, reading the `InputStream` feeds `x_tar` the bytes of the *tar* file it's to process.

Constructing the object

The majority of our `getContent()` method is devoted to interpreting the stream of bytes of the *tar* file and building our output object: the `String` that lists the contents of the *tar* file. Again, this means that this example involves the particulars of reading *tar* files, so you shouldn't fret too much about the details.

After requesting an `InputStream` from the `URLConnection`, `x_tar` loops, gathering information about each file. Each archived item is preceded by a header that contains attribute and length fields. `x_tar` interprets each header and then skips over the remaining portion of the item. It accumulates the results (the file listings) in a `StringBuffer`. (See Chapter 7, *Basic Utility Classes* for a discussion of `StringBuffer`.) For each file, we add a line of text listing the name, modification time, and size. When the listing is complete, `getContent()` returns the `StringBuffer` as a `String` object.

The main `while` loop continues as long as it's able to read another header record, and as long as the record's "name" field isn't full of ASCII null values. (The *tar* file format calls for the end of the archive to be padded with an empty header record, although most *tar* implementations don't seem to do this.) The `while` loop retrieves the name, size, and modification times as character strings from fields in the header. The most common *tar* format stores its numeric values in octal, as fixed-length ASCII strings. We extract the strings and use `Integer.parseInt()` to parse them.

After reading and parsing the header, `x_tar` skips over the data portion of the file and updates the variable `count`, which keeps track of the offset into the archive. The two lines following the initial skip account for *tar*'s "blocking" of the data records. In other words, if the data portion of a file doesn't fit precisely into an integral number of blocks of RECORDLEN bytes, *tar* adds padding to make it fit.

Whew. Well, as I said, the details of parsing *tar* files are not really our main concern here. But x_tar does illustrate a few tricks of data manipulation in Java.

It may surprise you that we didn't have to provide a constructor; our content handler relies on its default constructor. We don't need to provide a constructor because there isn't anything for it to do. Java doesn't pass the class any argument information when it creates an instance of it. You might suspect that the URLConnection object would be a natural thing to provide at that point. However, when you are calling the constructor of a class that is loaded at run-time, you can't pass it any arguments, as we discussed in Chapter 5, *Objects in Java*.

Using our new handler

When I began this discussion of content handlers, I showed a brief example of how our x_tar content handler would work for us. We need to make a few brief additions to that code in order to use our new handler and fetch URLs that point to *.tar* files. Since we're writing a standalone application, we're not only responsible for writing handlers that obey the package/class naming scheme we described earlier; we are also responsible for making our application use the naming scheme.

In a standalone application, the mapping between MIME types and content-handler class names is done by a special java.net.ContentHandlerFactory object we must install. The ContentHandlerFactory accepts a String containing a MIME type and returns the appropriate content handler. It's responsible for implementing the naming convention and creating an instance of our handler. Note that you don't need a content-handler factory if you are writing handlers for use by remote applications; a browser like HotJava, that loads content handlers over the Net, has its own content-handler factory.

To make absolutely clear what's happening, we'll provide a simple factory that knows only about our x_tar handler and install it at the beginning of our application:

```
import java.net.*;
import java.io.*;

class OurContentHandlerFactory implements ContentHandlerFactory {

    public ContentHandler createContentHandler(String mimetype) {
        if ( mimetype.equalsIgnoreCase( "application/x-tar" ) )
            return new net.www.content.application.x_tar();
        else
            return null;
    }
}
```

```
public class TarURLTest {
    public static void main (String [] args) throws Exception {

        URLConnection.setContentHandlerFactory(new
                      OurContentHandlerFactory() );

        URL url = new URL( args[0] );
        String s = (String)url.getContent();
        System.out.println( s );
    }
}
```

The class `OurContentHandlerFactory` implements the `ContentHandlerFactory` interface. It recognizes the MIME-type `application/x-tar` and returns a new instance of our `x_tar` handler. `TarURLTest` uses the static method `URLConnection.setContentHandlerFactory()` to install our new `ContentHandlerFactory`. After it's installed, our factory is called every time we retrieve the contents of a URL object. If it returns a `null` value, Java looks for handlers in a default location.[8]

> [8] If we don't install a `ContentHandlerFactory` (or later, as we'll see a `URLStreamHandlerFactory` for protocol handlers), Java defaults to searching for a vendor-specific package name. If you have Sun's Java Development Kit, it searches for content handlers in the `sun.net.www.content` package hierarchy and protocol handler classes in the `sun.net.www.protocol` package hierarchy.

After installing the factory, `TarURLTest` reads a URL from the command line, opens that URL, and lists its contents. Now you have a portable *tar* command that can read its *tar* files from arbitrary locations on the Net. I'll confess that I was lazy about exception handling in this example. Of course, a real application would need to catch and handle the appropriate exceptions; but we already know how to do that.

A final design note. Our content handler returned the *tar* listing as a `String`. I don't want to harp on the point, but this isn't the only option. If we were writing a content handler to work in the context of a Web browser, we might want it to produce some kind of HTML object that might display the listing as hypertext. Again, knowing the right solution requires that we know what kind of object a browser expects to receive, and currently that's undefined.

In the next section, we'll turn the tables and look at protocol handlers. There we'll be building `URLConnection` objects and someone else will have the pleasure of reconstituting the data.

9.6 Writing a Protocol Handler

A `URL` object uses a protocol handler to establish a connection with a server and perform whatever protocol is necessary to retrieve data. For example, an HTTP protocol handler knows how to talk to an HTTP server and retrieve a document; an FTP protocol handler knows how to talk to an FTP server and retrieve a file. All types of URLs use protocol handlers to access their objects. Even the lowly "file" type URLs use a special "file" protocol handler that retrieves files from the local filesystem. The data a protocol handler retrieves is then fed to an appropriate content handler for interpretation.

While we refer to a protocol handler as a single entity, it really has two parts: a `java.net.URLStreamHandler` and a `java.net.URLConnection`. These are both `abstract` classes we will subclass to create our protocol handler. (Note that these are `abstract` classes, not interfaces. Although they contain abstract methods we are required to implement, they also contain many utility methods we can use or override.) The `URL` looks up an appropriate `URLStreamHandler`, based on the protocol component of the `URL`. The `URLStreamHandler` then finishes parsing the URL and creates a `URLConnection` when it's time to communicate with the server. The `URLConnection` represents a single connection with a server, and implements the communication protocol itself.

Locating Protocol Handlers

Protocol handlers are organized in a package hierarchy similar to content handlers. Unlike content handlers, which are grouped into packages by the MIME types of the objects that they handle, protocol handlers are given individual packages. Both parts of the protocol handler (the `URLStreamHandler` class and the `URLConnection` class) are located in a package named for the protocol they support.

For example, the classes for an FTP protocol handler would be found in the `net.www.protocol.ftp` package. The `URLStreamHandler` is placed in this package and given the name `Handler`; all `URLStreamHandler`s are named `Handler` and distinguished by the package in which they reside. The `URLConnection` portion of the protocol handler is placed in the same package, and can be given any name. There is no need for a naming convention because the corresponding `URLStreamHandler` is responsible for creating the `URLConnection` objects it uses. Table 9.2 gives the obvious examples.[9]

[9] The "pre-beta 1" release of HotJava has a temporary solution that is compatible with the

convention described here. In the HotJava *properties* file, add the line: `java.protocol.handler.pkgs=net.www.protocol`.

Table 9.2: Mapping Protocols into Package and Class Names

Protocol	**Package**	**URLStreamHandler**	**Handler**
	name	**class name**	**class location**
FTP	`net.www.protocol.ftp`	`Handler`	*net/www/protocol/ftp/*
HTTP	`net.www.protocol.http`	`Handler`	*net/www/protocol/http/*

URLs, Stream Handlers, and Connections

The `URL`, `URLStreamHandler`, `URLConnection`, and `ContentHandler` classes work together closely. Before diving into an example, let's take a step back, look at the parts a little more closely, and see how these things communicate. Figure 9.5 shows how these components relate to each other.

Figure 9.5: The protocol handler machinery

[Graphic: Figure 9-5]

We begin with the `URL` object, which points to the resource we'd like to retrieve. The `URLStreamHandler` helps the `URL` class parse the URL specification string for its particular protocol. For example, consider the following call to the `URL` constructor:

```
URL url = new URL("protocol_3A/foo.bar.com/file.ext");
```

The `URL` class parses only the protocol component; later, a call to the `URL` class's `getContent()` or `openStream()` method starts the machinery in motion. The `URL` class locates the appropriate protocol handler by looking in the protocol-package hierarchy. It then creates an instance of the appropriate `URLStreamHandler` class.

The `URLStreamHandler` is responsible for parsing the rest of the URL string, including hostname and filename, and possibly an alternative port designation. This allows different protocols to have their own variations on the format of the URL specification string. Note that this step is skipped when a URL is constructed with the "protocol," "host," and "file" components specified explicitly. If the protocol is straightforward, its `URLStreamHandler` class can let Java do the parsing and accept the default behavior. For this illustration, we'll assume that the `URL` string requires no special parsing. (If we use a nonstandard URL with a strange format, we're responsible for parsing it ourselves, as I'll show shortly.)

The `URL` object next invokes the handler's `openConnection()` method, prompting the handler to create a new `URLConnection` to the resource. The `URLConnection` performs whatever communications are necessary to talk to the resource and begins to fetch data for the object. At that time, it also determines the MIME type of the incoming object data and prepares an `InputStream` to hand to the appropriate content handler. This `InputStream` must send "pure" data with all traces of the protocol removed.

The `URLConnection` also locates an appropriate content handler in the content-handler package hierarchy. The `URLConnection` creates an instance of a content handler; to put the content handler to work, the `URLConnection`'s `getContent()` method calls the content handler's `getContent()` method. If this sounds confusing, it is: we have three `getContent()` methods calling each other in a chain. The newly created `ContentHandler` object then acquires the stream of incoming data for the object by calling the `URLConnection`'s `getInputStream()` method. (Recall that we acquired an `InputStream` in our `x_tar` content handler.) The content handler reads the stream and constructs an object from the data. This object is then returned up the `getContent()` chain: from the content handler, the `URLConnection`, and finally the `URL` itself. Now our application has the desired object in its greedy little hands.

To summarize, we create a protocol handler by implementing a `URLStreamHandler` class that creates specialized `URLConnection` objects to handle our protocol. The `URLConnection` objects implement the `getInputStream()` method, which provides data to a content handler for construction of an object. The base `URLConnection` class implements many of the methods we need; therefore, our `URLConnection` needs only to provide the methods that generate the data stream and return the MIME type of the object data.

Okay. If you're not thoroughly confused by all that terminology (or even if you are), let's move on to the example. It should help to pin down what all these classes are doing.

The crypt Handler

In this section, we'll build a *crypt* protocol handler. It parses URLs of the form:

```
crypt:type//hostname[:port]/location/item
```

`type` is an identifier that specifies what kind of encryption to use. The protocol itself is a simplified version of HTTP; we'll implement the `GET` command and no more. I added the *`type`* identifier to the URL to show how to parse a nonstandard URL specification. Once the handler has figured out the encryption type, it dynamically loads a class that implements the chosen encryption algorithm and uses it to retrieve the data.

Obviously, we don't have room to implement a full-blown public-key encryption algorithm, so we'll use the `rot13InputStream` class from Chapter 8, *Input/Output Facilities*. It should be apparent how the example can be extended by plugging in a more powerful encryption class.

The Encryption class

First, we'll lay out our plug-in encryption class. We'll define an abstract class called `CryptInputStream` that provides some essentials for our plug-in encrypted protocol. From the `CryptInputStream` we'll create a subclass, `rot13CryptInputStream`, that implements our particular kind of encryption:

```
package net.www.protocol.crypt;
import java.io.*;

abstract class CryptInputStream extends InputStream {
    InputStream in;
    OutputStream talkBack;
    abstract public void set( InputStream in, OutputStream talkBack );
}

class rot13CryptInputStream extends CryptInputStream {

    public void set( InputStream in, OutputStream talkBack ) {
        this.in = new example.io.rot13InputStream( in );
    }
    public int read() throws IOException {
        if ( in == null )
            throw new IOException("No Stream");

        return in.read();
    }
}
```

Our `CryptInputStream` class defines a method called `set()` that passes in the `InputStream` it's to translate. Our `URLConnection` calls `set()` after creating an instance of the encryption class. We need a `set()` method because we want to load the encryption class dynamically, and we aren't allowed to pass arguments to the constructor of a class when it's dynamically loaded. We ran into this same restriction in our content handler. In the encryption class, we also provide an `OutputStream`. A more complex kind of encryption might use the `OutputStream` to transfer public-key information. Needless to say, *rot13* doesn't, so we'll ignore the `OutputStream` here.

The implementation of `rot13CryptInputStream` is very simple. `set()` just takes the `InputStream` it receives and wraps it with the `rot13InputStream` filter we developed in Chapter 8, *Input/Output Facilities*. `read()` reads filtered data from the `InputStream`, throwing an exception if `set()` hasn't been called.

The URLStreamHandler

Next we'll build our URLStreamHandler class. The class name is Handler; it extends the abstract URLStreamHandler class. This is the class the Java URL looks up by converting the protocol name (*crypt*) into a package name (net.www.protocol.crypt). The fully qualified name of this class is net.www.protocol.crypt.Handler:

```
package net.www.protocol.crypt;

import java.io.*;
import java.net.*;

public class Handler extends URLStreamHandler {
    String cryptype;

    protected void parseURL(URL u, String spec, int start, int end) {
        int slash = spec.indexOf('/');
        cryptype = spec.substring(start, slash);
        start=slash;
        super.parseURL(u, spec, start, end);
    }

    protected URLConnection openConnection(URL url)
       throws IOException {

        return new CryptURLConnection( url, cryptype );
    }
}
```

Java creates an instance of our URLStreamHandler when we create a URL specifying the *crypt* protocol. Handler has two jobs: to assist in parsing the URL specification strings and to create CryptURLConnection objects when it's time to open a connection to the host.

Our parseURL(default.htm) method overrides the parseURL(default.htm) method in the URLStreamHandler class. It's called whenever the URL constructor sees a URL requesting the *crypt* protocol. For example:

```
URL url = new URL("crypt_3Arot13/foo.bar.com/file.txt");
```

parseURL(default.htm) is passed a reference to the URL object, the URL specification string, and starting and ending indexes that shows what portion of the URL string we're expected to parse. The URL class has already identified the protocol name, otherwise it wouldn't have found our protocol handler. Our version of parseURL(default.htm) retrieves our *type* identifier from the specification, stores it in the variable cryptype, and then passes the rest on to the superclass's parseURL(default.htm) method to complete the job. To find the encryption type, take everything between the starting index we were given and the first slash in the URL string. Before calling super.parseURL(default.htm), we update the start index, so that it points to the character just after the type specifier. This tells the superclass parseURL(default.htm) that we've already parsed everything prior to the first slash, and it's responsible for the rest.

Before going on, I'll note two other possibilities. If we hadn't hacked the URL string for our own purposes by adding a type specifier, we'd be dealing with a standard URL specification. In this case, we wouldn't need to override parseURL(default.htm); the default implementation would have been sufficient. It could have sliced the URL into host, port, and filename components normally. On the other hand, if we had created a completely bizarre URL format, we would need to parse the entire string. There would be no point calling super.parseURL(default.htm); instead, we'd have called the URLStreamHandler's protected method setURL(default.htm) to pass the URL's components back to the URL object.

The other method in our Handler class is openConnection(). After the URL has been completely parsed, the URL object calls openConnection() to set up the data transfer. openConnection() calls the constructor for our URLConnection with appropriate arguments. In this case, our URLConnection object is named CryptURLConnection, and the constructor requires the URL and the encryption type as arguments. parseURL(default.htm) picked the encryption type from the URL string and stored it in the cryptype variable. openConnection() returns a reference to our URLConnection, which the URL object uses to drive the rest of the process.

The URLConnection

Finally, we reach the real guts of our protocol handler, the URLConnection class. This is the class that opens the socket, talks to the server on the remote host, and implements the protocol itself. This class doesn't have to be public, so you can put it in the same file as the Handler class we just defined. We call our class Crypt-URLConnection; it extends the abstract URLConnection class. Unlike ContentHandler and StreamURLConnection, whose names are defined by convention, we can call this class anything we want; the only class that needs to know about the URLConnection is the URLStreamHandler, which we wrote ourselves.

```
class CryptURLConnection extends URLConnection {
    CryptInputStream cis;
    static int defaultPort = 80;

    CryptURLConnection ( URL url, String cryptype )
        throws IOException {
        super( url );
        try {
            String name = "net.www.protocol.crypt." + cryptype
                                   + "CryptInputStream";
            cis = (CryptInputStream)Class.forName(name).newInstance();
        }
        catch ( Exception e ) { }
    }

    synchronized public void connect() throws IOException {
        int port;
        if ( cis == null )
            throw new IOException("Crypt Class Not Found");
        if ( (port = url.getPort()) == -1 )
            port = defaultPort;
        Socket s = new Socket( url.getHost(), port );

        // Send the filename in plaintext
        OutputStream server = s.getOutputStream();
        new PrintStream( server ).println( "GET "+url.getFile() );

        // Initialize the CryptInputStream
        cis.set( s.getInputStream(), server );
        connected = true;
    }

    synchronized public InputStream getInputStream()
       throws IOException {
        if (!connected)
            connect();
        return ( cis );
    }

    public String getContentType() {
        return guessContentTypeFromName( url.getFile() );
    }
}
```

The constructor for our `CryptURLConnection` class takes as arguments the destination `URL` and the name of an encryption type. We pass the `URL` on to the constructor of our superclass, which saves it in a protected `url` instance variable. We could have saved the `URL` ourselves, but calling our parent's constructor shields us from possible changes or enhancements to the base class. We use `cryptype` to construct the name of an encryption class, using the convention that the encryption class is in the same package as the protocol handler (i.e., `net.www.protocol.crypt`); its name is the encryption type followed by the suffix `CryptInputStream`.

Once we have a name, we need to create an instance of the encryption class. To do so, we use the static method `Class.forName()` to turn the name into a `Class` object and `newInstance()` to load and instantiate the class. (This is how Java loads the content and protocol handlers themselves.) `newInstance()` returns an `Object`; we need to cast it to something more specific before we can work with it. Therefore, we cast it to our `CryptInputStream` class, the abstract class that `rot13CryptInputStream` extends. If we implement any additional encryption types as extensions to `CryptInputStream` and name them appropriately, they will fit into our protocol handler without modification.

We do the rest of our setup in the `connect()` method of the `URLConnection`. There, we make sure we have an encryption class and open a `Socket` to the appropriate port on the remote host. `getPort()` returns `-1` if the `URL` doesn't specify a port explicitly; in that case we use the default port for an HTTP connection (port 80). We ask for an `OutputStream` on the socket, assemble a `GET` command using the `getFile()` method to discover the filename specified by the URL, and send our request by writing it into the `OutputStream`. (For convenience, we wrap the `OutputStream` with a `PrintStream` and call `println()` to send the message.) We then initialize the `CryptInputStream` class by calling its `set()` method and passing it an `InputStream` from the `Socket` and the `OutputStream`.

The last thing `connect()` does is set the `boolean` variable `connected` to `true`. `connected` is a `protected` variable inherited from the `URLConnection` class. We need to track the state of our connection because `connect()` is a `public` method. It's called by the `URLConnection`'s `getInputStream()` method, but it could also be called by other classes. Since we don't want to start a connection if one already exists, we use `connected` to tell us if this is so.

In a more sophisticated protocol handler, `connect()` would also be responsible for dealing with any protocol headers that come back from the server. In particular, it would probably stash any important information it can deduce from the headers (e.g., MIME type, content length, time stamp) in instance variables, where it's available to other methods. At a minimum, `connect()` strips the headers from the data so the content handler won't see them. I'm being lazy and assuming that we'll connect to a minimal server, like the modified `TinyHttpd` daemon I discuss below, which doesn't bother with any headers.

The bulk of the work has been done; a few details remain. The `URLConnection`'s `getContent()` method needs to figure out which content handler to invoke for this `URL`. In order to compute the content handler's name, `getContent()` needs to know the resource's MIME type. To find out, it calls the `URLConnection`'s `getContentType()` method, which returns the MIME type as a `String`. Our protocol handler overrides `getContentType()`, providing our own implementation.

The `URLConnection` class provides a number of tools to help determine the MIME type. It's possible that the MIME type is conveyed explicitly in a protocol header; in this case, a more sophisticated version of `connect()` would have stored the MIME type in a convenient location for us. Some servers don't bother to insert the appropriate headers, though, so you can use the method `guessContentTypeFromName()` to examine filename extensions, like *.gif* or *.html*, and map them to MIME types. In the worst case, you can use `guessContentTypeFromStream()` to intuit the MIME type from the raw data. The Java developers call this method "a disgusting hack" that shouldn't be needed, but that is unfortunately necessary "in a world where HTTP servers lie about content types and extensions are often nonstandard." We'll take the easy way out and use the `guessContentTypeFromName()` utility of the `URLConnection` class to determine the MIME type from the filename extension of the URL we are retrieving.

Once the `URLConnection` has found a content handler, it calls the content handler's `getContent()`

method. The content handler then needs to get an `InputStream` from which to read the data. To find an `InputStream`, it calls the `URLConnection`'s `getInputStream()` method. `getInputStream()` returns an `InputStream` from which its caller can read the data after protocol processing is finished. It checks whether a connection is already established; if not, it calls `connect()` to make the connection. Then it returns a reference to our `CryptInputStream`.

A final note on getting the content type: if you read the documentation, it's clear that the Java developers had some ideas about how to find the content type. The `URLConnection`'s default `getContentType()` calls `getHeaderField()`, which is presumably supposed to extract the named field from the protocol headers (it would probably spit back information `connect()` had stored in protected variables). The problem is there's no way to implement `getHeaderField()` if you don't know the protocol, and since the Java developers were designing a general mechanism for working with protocols, they couldn't make any assumptions. Therefore, the default implementation of `getHeaderField()` returns `null`; you have to override it to make it do anything interesting. Why wasn't it an abstract method? I can only guess, but making `getHeaderField()` abstract would have forced everyone building a protocol handler to implement it, whether or not they actually needed it.

The application

We're almost ready to try out a crypt URL! We still need an application (a mini-browser, if you will) to use our protocol handler, and a server to serve data with our protocol. If HotJava were available, we wouldn't need to write the application ourselves; in the meantime, writing this application will teach us a little about how a Java-capable browser works. Our application is similar to the application we wrote to test the `x_tar` content handler.

Because we're working in a standalone application, we have to tell Java how to find our protocol-handler classes. Java relies on a `java.net.URLStreamHandlerFactory` object to take a protocol name and return an instance of the appropriate handler. The `URLStreamHandlerFactory` is very similar to the `ContentHandlerFactory` we saw earlier. We'll provide a trivial implementation that knows only our particular handler. Again, if we were using our protocol handler with HotJava, this step would not be necessary; HotJava has its own stream-handler factory that tells it where to find handlers. To get HotJava to read files with our new protocol, we'd only have to put our protocol handler in the right place. (Note too, that an applet running in HotJava can use any of the methods in the `URL` class and therefore can use the content- and protocol-handler mechanism; applets would also rely on HotJava's stream-handler and content-xhandler factories.)

Here's our `StreamHandlerFactory` and sample application:

```
import java.net.*;

class OurURLStreamHandlerFactory implements URLStreamHandlerFactory {
    public URLStreamHandler createURLStreamHandler(String protocol) {
        if ( protocol.equalsIgnoreCase("crypt") )
            return new net.www.protocol.crypt.Handler();
        else
            return null;
    }
}

class CryptURLTest {
    public static void main( String argv[] ) throws Exception {

        URL.setURLStreamHandlerFactory(
            new OurURLStreamHandlerFactory());

        URL url = new URL("crypt_3Arot13/foo.bar.com_3A1234/myfile.txt");
        System.out.println( url.getContent() );
    }
}
```

The `CryptURLTest` class installs our factory and reads a document via the new "crypt:rot13" URL. (In the example, we have assumed that a *rot13* server is running on port 1234 on the host foo.bar.com.) When the `CryptURLTest` application calls the `URL`'s `getContent()` method, it automatically finds our protocol handler, which decodes the file.

`OurURLStreamHandlerFactory` is really quite simple. It implements the `URLStreamHandlerFactory` interface, which requires a single method called `createURLStreamHandler()`. In our case, this method checks whether the protocol's name is *crypt* ; if so, the method returns an instance of our encryption protocol handler, `net.www.protocol.crypt.Handler`. For any other protocol name, it returns `null`. If we were writing a browser and needed to implement a more general factory, we would compute a class name from the protocol name, check to see if that class exists, and return an instance of that class.

The server

We still need a *rot13* server. Since the *crypt* protocol is nothing more than HTTP with some encryption added, we can make a *rot13* server by modifying one line of the `TinyHttpd` server we developed earlier, so that it spews its files in *rot13*. Just change the line that reads the data from the file:

```
f.read( data );
```

To instead read through a `rot13InputStream`:

```
new example.io.rot13InputStream( f ).read( data );
```

I assume you placed the `rot13InputStream` example in a package called `example.io`, and that it's somewhere in your class path. Now recompile and run the server. It automatically encodes the files before sending them; our sample application decodes them on the other end.

I hope that this example and the rest of this chapter have given you some food for thought. Content and protocol handlers are among the most exciting ideas in Java. It's unfortunate that we have to wait for future releases of HotJava and Netscape to take full advantage of them. In the meantime, you can experiment and implement your own applications.

10.2 Applets

If you've been waiting for a more detailed discussion of the applet class, here it is. For examples of writing applets, please see Chapter 2, *A First Applet* (the tutorial) and the examples in Chapter 11, *Using and Creating GUI Components* and throughout the book.

An `Applet` is something like a `Panel` with a mission. It is a GUI Container that has some extra structure to allow it to be used in an "alien" environment like a Web browser or appletviewer. Applets also have a life-cycle that lets them act more like an application than a static component. Although applets tend to be relatively simple, there's no inherent restriction on their complexity. There's no reason you couldn't write an air traffic control system (well, let's be less ambitious: a word processor) as an applet.

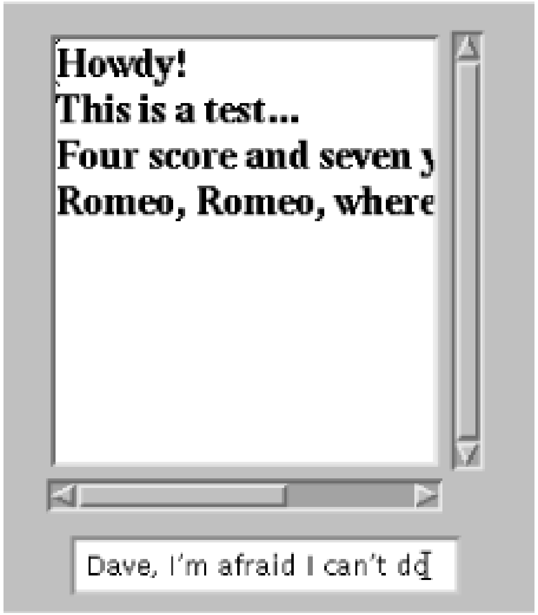
Structurally, an applet is a sort of wrapper for your Java code. In contrast to a standalone graphical Java application, which starts up from a `main()` method and creates a GUI, an applet is itself a Component that expects to be dropped into someone else's GUI. Thus, an applet can't run by itself; it runs in the context of a Web browser or an appletviewer. Instead of having your application create a `Frame` to hold your GUI, you stuff your application inside an `Applet` (which is itself a Container) and let someone else add the applet to their GUI.

Pragmatically, an applet is an intruder into someone else's environment, and therefore has to be treated with suspicion. The Web browsers that run applets impose restrictions on what the applet is allowed to do. The restrictions are enforced by a security manager, which the applet is not allowed to change. The browser also provides an "applet context," which is additional support that helps the applet live within its restrictions.

Aside from that top level structure and the security restrictions, there is no difference between an applet and an application. If your application can live within the restrictions imposed by a browser's security manager, you can easily structure it to function as an applet and a standalone application. (We'll show an example of an Applet that can also be run as a standalone below.) Conversely, if you can supply all of the things that an applet requires from its environment, you can use applets within your stand-alone applications and within other applets (though this requires a bit of work).

As we said a moment ago, an `Applet` expects to be embedded in GUI (perhaps a document) and used in a viewing environment that provides it with special resources. In all other respects, however, applets are just ordinary `Panel` objects. (See Figure 10.8.) Like a `Panel`, an `Applet` can contain user-interface components and implement all the basic drawing and event-handling capabilities of the `Component` class. We draw on an `Applet` by overriding its `paint()` method; we respond to events in the `Applet`'s display area by providing the appropriate event-listeners. The additional structure applets have helps them interact with the viewer environment.

Figure 10.8: The java.applet package



Howdy!
This is a test...
Four score and seven y
Romeo, Romeo, where

Dave, I'm afraid I can't do

Applet Control

The `Applet` class contains four methods an applet can override to guide it through its life cycle. The `init()`, `start()`, `stop()`, and `destroy()` methods are called by an applet viewer, such as a Web browser, to direct the applet's behavior. `init()` is called once, after the applet is created. The `init()` method is where you perform basic setup like parsing parameters, building a user interface, and loading resources. Given what we've said about objects, you might expect the `Applet`'s constructor would be the right place for such initialization. However, the constructor is meant to be called by the applet's environment, for simple creation of the applet. This might happen before the applet has access to certain resources, like information about its environment. Therefore, an applet doesn't normally do any work in its constructor; it relies on the default constructor for the `Applet` class and does its initialization in the `init()` method.

The `start()` method is called whenever the applet becomes visible; it shouldn't be a surprise then that the `stop()` method is called whenever the applet becomes invisible. `init()` is only called once in the life of an applet, but `start()` and `stop()` can be called any number of times (but always in the logical sequence). For example, `start()` is called when the applet is displayed, such as when it scrolls onto the screen; `stop()` is called if the applet scrolls off the screen or the viewer leaves the document. `start()` tells the applet it should be active. The applet may want to create threads, animate, or otherwise perform useful (or annoying) activity. `stop()` is called to let the applet know it should go dormant. Applets should cease CPU-intensive or wasteful activity when they are stopped and resume it when (and if) they are restarted. However, there's no requirement that an invisible applet stop computing; in some applications, it may be useful for the applet to continue running in the background. Just be considerate of your user, who doesn't want an invisible applet dragging down system performance. And for the users: be aware of the tools that will develop to let you monitor and squash rogue applets in your web browser.

Finally, the `destroy()` method is called to give the applet a last chance to clean up before it's removed--some time after the call to `stop()`. For example, an applet might want to close down suspended communications channels or remove graphics frames. Exactly when `destroy()` is called depends on the applet viewer; Netscape Navigator calls `destroy()` just prior to deleting the applet from its cache. This means that although an applet can cling to life after being told to `stop()`, how long it can go on is unpredictable. If you want to maintain your applet as the user progresses through other activities, consider putting it in an HTML frame (see "Driving the Browser" later in this chapter).

The Applet security Sandbox

Applets are quarantined within the browser by an applet `SecurityManager`. The `SecurityManager` is part of the application that runs the applet, e.g., the web browser or applet viewer. It is installed before the browser loads any applets and implements the basic restrictions that let you run untrusted applets safely. Remember that, aside from basic language robustness, there are no inherent security restrictions on a standalone Java application. It is the browser's responsibility to install a special security manager and limit what applets are allowed to do.

Most browsers impose the following restrictions on untrusted applets:

- Untrusted Applets cannot read or write files on the local host.
- Untrusted Applets can only open network connections (sockets) to the server from which they originated.
- Untrusted Applets cannot start other processes on the local host.
- Untrusted Applets cannot have native methods.

We discuss these restrictions in more detail in the relevant chapters in this book. However, the motivation for these restrictions should be fairly obvious: you clearly wouldn't want a program coming from some random Internet site to access your files, or run arbitrary programs. Although untrusted applets cannot directly read and write files on the client side or talk to arbitrary hosts on the network, applets can work with servers to store data and communicate. For example, an applet can use Java's RMI (Remote Method Invocation) facility to do processing on its server. An applet can communicate with other applets on the Net by proxy through its server.

Trusted Applets

The latest version of Java makes it possible to sign archive files that contain applets. Because a signature identifies the applet's origin unambiguously, we can now distinguish between "trusted" applets (i.e., applets that come from a site or person you trust not to do anything harmful) and run of the mill "untrusted" applets. In web browsers that support signing, trusted applets can be granted permission to "go outside" of the applet security sandbox. Trusted applets can be allowed to do most of the things that standalone Java applications can do: read and write files, open network connections to arbitrary machines, and interact with the local operating system by starting processes. Trusted applets still can't have native methods, but including native methods in an applet would make it unportable, and would therefore be a bad idea.

Chapter 3, *Tools of the Trade* discusses how to package your applet's class files and resources into a JAR file and sign it with your digital signature. Currently, HotJava is the only browser that supports signing, but Netscape Navigator, Internet Explorer, and others will probably catch up soon.

Getting Applet Resources

An applet needs to communicate with its applet viewer. For example, it needs to get its parameters from the HTML document in which it appears. An applet may also need to load images, audio clips, and other items. It may also want to ask the viewer about other applets on the same HTML page in order to communicate with them. To get resources from the applet viewer environment, applets use the `AppletStub` and `AppletContext` interfaces. Unless you're writing a browser or some other application that loads and runs applets, you won't have to implement these interfaces, but you do use them within your applet.

Applet Parameters

An applet gets its parameters from the parameter tags placed inside the `<applet>` tag in the HTML document. For example, the code below reads the "sheep" parameter from its HTML page:

```
String imageName = getParameter( "imageName" );
try  {
    int numberOfSheep = Integer.parseInt(getParameter( "sheep" ));
} catch ( NumberFormatException e ) { // use default }
```

A friendly applet will provide information about the parameters it accepts through its `getParameterInfo()` method. `getParameterInfo()` returns an array of string arrays, listing and describing the applet's parameters. For each parameter, three strings are provided: the parameter name, its possible values or value types, and a verbose description. For example:

```
public String [][] getParameterInfo() {
    String [][] appletInfo =
        {"logo",   "url",   "Main logo image"}
        {"timer", "int",   "Time to wait before becoming annoying"},
        {"flashing", "constant | intermittant",  "Flag for how to flash"},
    return appletInfo;
}
```

Applet Resources

An applet can find where it lives by calling the `getDocumentBase()` and `getCodeBase()` methods. `getDocumentBase()` returns the base URL of the document in which the applet appears; `getCodeBase()` returns the base URL of the `Applet`'s class files. An applet can use these to construct relative URLs from which to load other resources like images, sounds, and other data. The `getImage()` method takes a URL and asks for an image from the viewer environment. The image may be pulled from a cache or loaded asynchronously when later used. The `getAudioClip()` method, similarly, retrieves sound clips. See Chapter 9, *Network Programming* for a full discussion of how to work with URLs, and Chapter 11, *Using and Creating GUI Components* for examples of applets that load images.

The following example uses `getCodeBase()` to construct a URL and load a properties-configuration file, located at the same location as the applet's class file. (See Chapter 7, *Basic Utility Classes* for a discussion of properties.)

```
Properties props = new Properties();
try {
    URL url = new URL(getCodeBase(), "appletConfig.props");
    props.load( url.openStream() );
} catch ( IOException e ) { // failed }
```

A better way to load resources is by calling the `getResource()` and `getResourceAsStream()` methods of the `Class` class, which search the applet's JAR files (if any) as well as its codebase. See Chapter 8, *Input/Output Facilities* for a discussion of resource loading. The following code loads the properties file appletConfig.props:

```
Properties props = new Properties();
try {
    props.load( getClass().getResourceAsStream("appletConfig.props") );
} catch ( IOException e ) { // failed }
```

Driving the Browser

The status line is a blurb of text that usually appears somewhere in the viewer's display, indicating a current activity. An applet can request that some text be placed in the status line with the `showStatus()` method. (The browser isn't required to do anything in response to this call, but most browsers will oblige you.)

An applet can also ask the browser to show a new document. To do this, the applet makes a call to the showDocument(url) method of the `AppletContext`. You can get a reference to the `AppletContext` with the Applet's `getAppletContext()` method. `showDocument()` can take an additional `String` argument to tell the browser where to display the new URL:

```
getAppletContext().showDocument( url, name );
```

The `name` argument can be the name of an existing labeled HTML frame; the document referenced by the URL will be displayed in that frame. You can use this method to create an applet that "drives" the browser to new locations dynamically, but stays active on the screen in a separate frame. If the named frame doesn't exist, the browser will create a new top-level window to hold it. Alternatively, `name` can have one of the following special values:

_self	Show in the current Frame
_parent	Show in the parent of our Frame
_top	Show in outer-most (top level) frame.
_blank	Show in a new top level browser window.

Both `showStatus()` and `showDocument()` requests may be ignored by a cold-hearted viewer or Web browser.

```
    *** Missing Discussion of getApplet() ***
    *** Add a blurb about the upcoming InfoBus stuff ***
```

Applets vs. Standalone Applications

```
    *** Discuss getImage() and image loading from JAR files for applications ***
```

The following list summarizes the methods of the applet API:

```
// from the AppletStub
boolean isActive();
```

```
URL getDocumentBase();
URL getCodeBase();
String getParameter(String name);
AppletContext getAppletContext();
void appletResize(int width, int height);
// from the AppletContext
AudioClip getAudioClip(URL url);
Image getImage(URL url);
Applet getApplet(String name);
Enumeration getApplets();
void showDocument(URL url);
public void showDocument(URL url, String target);
void showStatus(String status);
```

These are the methods that are provided by the applet viewer environment. If your applet doesn't happen to use any of them, or if you can provide alternatives to handle special cases (such as loading images from JAR files), your applet could be made able to function as a standalone application as well as an applet. For example, our `HelloWeb` applet from Chapter 2, *A First Applet* was very simple. We can easily give it a `main()` method to allow it to be run as a standalone application:

```
public class HelloWeb extends Applet {

    public void paint( java.awt.Graphics gc ) {
        gc.drawString( "Hello Web!", 125, 95 );
    }

    public static void main( String [] args ) {
        Frame theFrame = new Frame();
        Applet helloWeb = new HelloWeb();
        theFrame.add("Center", helloWeb);
        theFrame.setSize(200,200);
        helloWeb.init();
        helloWeb.start();
        theFrame.show();
    }
}
```

Here we get to play "appletviewer" for a change. We have created an instance of `HelloWeb` using its constructor - something we don't normally do$mdash;and added it to our own `Frame`. We call its `init()` method to give the applet a chance to wake up and then call its `start()` method. In this example, `HelloWeb` doesn't implement these, `init()` and `start()`, so we're calling methods inherited from the `Applet` class. This is the procedure that an appletviewer would use to run an applet. (If we wanted to go further, we could implement our own `AppletContext` and `AppletStub`, and set them in the `Applet` before startup).

Trying to make your applets into applications as well often doesn't make sense, and is not always trivial. We show this only to get you thinking about the real differences between applets and applications. It is probably best to stay within the applet API until you have a need to go outside it. Remember that trusted applets can do almost all of the things that applications can. It is probably wiser to make an applet that requires trusted permissions than an application.

Events

We've spent a lot of time discussing the different kinds of objects in AWT--components, containers, and a few special containers like applets. But we've neglected communications between different objects. A few times, we've mentioned events, and we have even used them in the occasional program (like our applets in Chapter 2, *A First Applet*), but we have deferred a discussion of events until later. Now is the time to pay that debt.

AWT objects communicate by sending events. The way we talk about "firing" events and "handling" them makes it sound as if they are part of some special Java language feature. But they aren't. An event is simply an ordinary Java object that is delivered to its receiver by invoking an ordinary Java method. Everything else, however interesting, is purely convention. The entire Java event mechanism is really just a set of conventions for the kinds of descriptive objects that should be delivered; these conventions prescribe when, how, and to whom events should be delivered.

Events are sent from a single source object to one or more listeners (or "targets"). A listener implements specific event handling methods that enable it to receive a type of event. It then registers itself with a source of that kind of event. Sometimes there may be an "adapter" object interposed between the event source and the listener, but there is always a connection established before any events are delivered.

An event object is a subclass of `java.util.EventObject` that holds information about "something that's happened" to its source. The `EventObject` class serves mainly to identify event objects; the only information it contains is a reference to the event source (the object that sent the event). Components do not normally send or receive `EventObject`s as such; they work with subclasses that provide more specific information. `AWTEvent` is a subclass of `EventObject` that is used within AWT; further subclasses of `AWTEvent` provide information about specific event types.

For example, events of type `ActionEvent` are fired by buttons when they are pushed. `ActionEvent`s are also sent when a menu item is selected or when a user presses ENTER in a `TextField`. Similarly, `MouseEvent`s are generated when you operate your mouse within a component's area. You can gather the general meaning of these two events from their names; they are relatively self-descriptive. `ActionEvent`s correspond to a decisive "action" that a user has taken with the component--like pressing a button, or pressing ENTER to indicate that he has filled in a text field. An `ActionEvent` thus carries the name of an action to be performed (the "action command") by the program. `MouseEvent`s describe the state of the mouse, and therefore carry information like the x and y coordinates and the state of your mouse buttons at the time it was created. You might hear someone say that `ActionEvent` is at a "higher semantic level" than `MouseEvent`. This means that `ActionEvent` is an interpretation of something that happened and is, therefore, conceptually more powerful than the `MouseEvent`, which carries raw data. An `ActionEvent` lets us know that a component has done its job, while a `MouseEvent` simply confers a lot of information about the mouse at a given time. You could figure out that somebody clicked on a `Button` by examining mouse events, but it is simpler to work with action events. The precise meaning of an event, however, can depend on the context in which it is received. (More on that in a moment.)

Event Receivers and Listener Interfaces

An event is delivered by passing it as an argument to an event handler method in the receiving object. `ActionEvent`s, for example, are always delivered to a method called `actionPerformed()` in the receiver:

```
// Receiver
public void actionPerformed( ActionEvent e ) {
    ...
}
```

For each type of event, there is a corresponding listener interface that describes the methods it must provide to receive those events. In this case, any object that receives `ActionEvent`s must implement the `ActionListener` interface:

```
public interface ActionListener extends java.util.EventListener {
    public void actionPerformed( ActionEvent e );
}
// Reciever implements ActionListener
```

All listener interfaces are subinterfaces of `java.util.EventListener`, but `EventListener` is simply an empty interface. It exists only to help the compiler identify listener interfaces.

Listener interfaces are required for a number of reasons. First, they help to identify objects that are capable of receiving a given type of event. This way we can give the event handler methods friendly, descriptive names and still make it easy for documentation, tools, and humans to recognize them in a class. Next, listener interfaces are useful because there can be several methods specified for an event receiver. For example, the `FocusListener` interface contains two methods:

```
abstract void focusGained( FocusEvent e );
abstract void focusLost( FocusEvent e );
```

Athough these methods both take a `FocusEvent` as an argument, they correspond to different meanings for why the event was fired; in this case, whether the `FocusEvent` means that focus was received or lost. You could figure out what happened by inspecting the event; all `AWTEvent`s contain a constant specifying the event's subtype. By requiring two methods, the `FocusListener` interface saves you the effort: if `focusGained()` is called, you know the event type was `FOCUS_GAINED`. Similarly, the `MouseListener` interface defines five methods for receiving mouse events (and `MouseMotionListener` defines two more), each of which gives you some additional information about why the event occurred. In general, the listener interfaces group sets of related event handler methods; the method called in any given situation provides a context for the information in the event object.

There can be more than one listener interface for dealing with a particular kind of event. For example, the `MouseListener` interface describes methods for receiving `MouseEvent`s when the mouse enters or exits an area, or a mouse button is pressed or released. `MouseMotionListener` is an entirely separate interface that describes methods to get mouse events when the mouse is moved (no buttons pressed) or dragged (buttons pressed). By separating mouse events into these two categories, Java lets you be a little more selective about the circumstances under which you want to recieve `MouseEvent`s. You can register as a listener for mouse events without receiving mouse motion events; since mouse motion events are extremely common, you don't want to handle them if you don't need to.

Finally, we should point out two simple patterns in the naming of AWT event listener interfaces and handler methods:

- Event handler methods are public methods that return type `void` and take a single event object (a subclass of `java.util.EventObject` as an argument).[2]

[2] This rule is not complete. The full Java Beans allows event handler methods to take additional arguments when absolutely necessary and also to throw checked exceptions.

- Listener interfaces are subclasses of `java.util.EventListener` that are named with the suffix "Listener," e.g., FooListener.

These may seem pretty obvious, but they are important because they are our first hint of a *design pattern* governing how to build components that work with events.

Event Sources

The previous section described the machinery that an event receiver uses to accept events. In this section we'll describe how the receiver tells an event source to start sending it events as they occur.

To receive events, an eligible listener must register itself with an event source. It does this by calling an "add listener" method in the event source, and passing a reference (a callback) to itself. For example, the AWT `Button` class is a source of `ActionEvent`s. In order to receive these events, our code might do something like the following:

```
// source of ActionEvents
Button theButton = new Button("Belly");

// receiver of ActionEvents
class TheReceiver implements ActionListener {
   setupReceiver() {
      ...
      theButton.addActionListener( this );
   }
   public void actionPerformed( ActionEvent e ) {
      // Belly Button pushed...
   }
```

The receiver makes a call to `addActionListener()` to complete its setup and become eligible to receive `ActionEvent`s from the button when they occur. It passes the reference `this`, to add itself as the `ActionListener`.

To manage its listeners, an `ActionEvent` source (like the `Button`) always implements two methods:

```
// ActionEvent source
public void addActionListener(ActionListener listener) {
   ...
}
public void removeActionListener(ActionListener listener) {
   ...
}
```

The `removeActionListener()` method complements `addActionListener()` and does what you'd expect: it removes the listener from the list so that it will not receive future events from that source.

Now, you may be expecting an "event source" interface listing these two methods, but there isn't one. There are no event source interfaces in the current conventions. If you are analyzing a class and trying to determine what events it generates, you have to look for the add and remove methods. For example, the presence of the addActionListener() and removeActionListener() methods define the object as a source of ActionEvents. If you happen to be a human being, you can simply look at the documentation; but if the documentation isn't available, or if you're writing a program that needs to analyze a class (a process called "reflection"), you can look for this design pattern:

- A source of events for the FooListener interface must implement a pair of add/remove methods:
    - addFooListener(FooListener listener) (*)
    - removeFooListener(FooListener listener)
- If an event source can only support one event listener (unicast delivery), the add listener method can throw the checked exception java.util.TooManyListenersException.

So, what do all the naming patterns up to this point accomplish? Well, for one thing they make it possible for automated tools and integrated development environments to divine what are sources and what are sinks of particular events. Tools that work with Java Beans will use the Java reflection and introspection APIs to search for these kinds of design patterns and identify the events that can be fired and received by a component.

It also means that event hookups are strongly typed, just like the rest of Java. So, it's not easy to accidentally hook up the wrong kind of components; for example, you can't register to receive ItemEvents from a Button, because a button doesn't have an addItemListener() method. Java knows at compile time what types of events can be delivered to whom.

Event Delivery

AWT events are multicast; every event is associated with a single source, but can be delivered to any number of receivers. Events are registered and distributed using an observer/observable model. When an event listener registers itself with an event source, the event source adds the listener to a list. When an event is fired, it is delivered individually to each listener on the list.

Figure 10.9: Event delivery

There are no guarantees about the order in which events will be delivered. Neither are there any guarantees if you register yourself more than once with an event source; you may get the event more than once, or not. Similarly, you should assume that every listener receives the same event data. Events are "immutable"; they can't be changed by their listeners. There's one important exception to this rule, which we'll discuss later.

To be complete we could say that event delivery is synchronous with respect to the event source, but that follows from the fact that the event delivery is really just the invocation of a normal Java method. The source of the event calls the handler method of each listener. However, listeners shouldn't assume that all of the events will be sent in the same thread. An event source could decide to sent out events to all of the listeners in parallel.

How exactly an event source maintains its set of listeners, constructs, and fires the events is up to it. Often it is sufficient to use a `Vector` to hold the list. We'll show the code for a component that uses a custom event in Chapter 11, *Using and Creating GUI Components*. For efficiency, AWT components all use the `java.awt.AWTEventMulticaster` object, which maintains a linked tree of the listeners for the component. You can use that too, if you are firing standard AWT events. We'll describe the event multicaster in Chapter 11, *Using and Creating GUI Components* as well.

AWTEvents

All of the events used by AWT GUI components are subclasses of `java.awt.AWTEvent`. `AWTEvent` holds some common information that is used by AWT to identify and process events. You can use or subclass any of the `AWTEvent` types for use in your own components.

```
Use the event hierarchy from Java in a Nutshell or AWT Reference.
```

`ComponentEvent` is the base class for events that can be fired by any AWT component. This includes events that provide notification when a component changes its dimensions or visibility, as well as the other event types for mouse operation and key presses. `ContainerEvent`s are fired by AWT containers when components are added or removed.

java.awt.event.InputEvent

`MouseEvent`s, which track the state of the mouse, and `KeyEvent`s, which are fired when the user uses the keyboard, are types of `java.awt.event.InputEvent`. Input events from the mouse and keyboard are a little bit special. They are normally produced by the native Java machinery associated with the peers. When the user touches a key or moves the mouse within a component's area, the events are generated with that component as the source.

Input events and some other AWT events are placed on a special event queue that is managed by the AWT Toolkit. This gives the Toolkit control over how the events are delivered. First, under some circumstances, the Toolkit can decide to "compress" a sequence of the same type of event into a single event. This is done to make some event types more efficient--in particular, mouse events and some special internal events used to control repainting. Perhaps more important to us, input events are delivered with a special arrangement that lets listeners decide if the component itself should act on the event.

Consuming events

Normally, the native peer of a standard AWT component operates by receiving `InputEvent`s telling it about the mouse and keyboard. When you push a `Button`, the native `ButtonPeer` object receives a `MouseEvent` and does its job in native land to accomplish the button-depressing behavior. But for `InputEvent`s, the Toolkit first delivers the event to any listeners registered with the the component and gives those listeners a chance to mark the event as "consumed," effectively telling the peer to ignore it. An `InputEvent` is marked "consumed" by calling the `consume()` method. (Yes, this is a case where an event is not treated as immutable.)

So, we could stop our `Button` from working by registering a listener with it that catches "mouse button depressed" events. When it got one, we could call its `consume()` method to tell the `ButtonPeer` to ignore that event. This is particularly useful if you happen to be building a develoment environment in Java and you want to "turn off" components while the user arranges them.

If you need to, in a trusted application you can get access to the AWT event queue. The Toolkit uses an instance of `java.awt.EventQueue`. With it you can peek at pending AWT events or even to push in new ones.

Mouse and Key Modifiers on InputEvents

`InputEvent`s come with a set of flags for special modifiers. These let you detect if the SHIFT or ALT key was held down during a mouse button or key press, or if the second or third mouse buttons were pressed. The following are the flag values contained in `java.awt.event.InputEvent`:

- SHIFT_MASK
- CTRL_MASK
- META_MASK
- ALT_MASK
- BUTTON1_MASK
- BUTTON2_MASK
- BUTTON3_MASK

To check for these masks, you can simply do a boolean AND of the modifiers, returned by the `InputEvent`'s `getModifiers()` method and the flag or flags you're interested in:

```
public void mousePressed (MouseEvent e) {
    int mods = e.getModifiers();
    if ((mods & InputEvent.SHIFT_MASK) != 0)
        // Shifted Mouse Button press
}
```

In the list you'll notice there are three BUTTON flags. These can be used to detect if a particular mouse button was used in a mouse press on a two or three button mouse. Be warned, if you use these you run the risk that your program won't work on platforms without multibutton mice. Currently, BUTTON2_MASK is equivalent to ALT_MASK, and BUTTON3_MASK is equivalent to META_MASK. This means that pushing the second mouse button is equivalent to pressing the first (or only) button with the ALT key depressed, and the third button is equivalent to the first with the META key depressed. However, if you really want to guarantee portability, you should limit yourself to a single button, possibly in combination with keyboard modifiers, rather than relying on the button masks.

Key events provide one other situation in which events aren't immutable. You can change the character that the user typed by calling `setKeyChar()`, `setKeyCode()`, or `setKeyModifiers()`. A user's keystroke isn't displayed until the `KeyEvent` is delivered to the peer. Therefore, by changing the character in the `KeyEvent`, you can change the character displayed on the screen. This is a good way to implement a field that only displays uppercase characters, regardless of what the user types.

AWT Event Reference

The following tables summarize the AWT Events, which components fire them, and the methods of the listener interfaces that receive them:

AWT Component and Container Events

Event	**Fired by**	**Listener interface(s)**	**Handler Method(s)**

ComponentEvent	*All Components (*)*	ComponentListener	componentResized()
			componentMoved()
			componentShown()
			componentHidden()

FocusEvent		FocusListener	focusGained()
			focusLost()

KeyEvent		KeyListener	keyTyped()
			keyPressed()
			keyReleased()

MouseEvent		MouseListener	mouseClicked()
			mousePressed()
			mouseReleased()
			mouseEntered()
			mouseExited()

		MouseMotionListener	mouseDragged()
			mouseMoved()

ContainerEvent	*All Containers*	ContainerListener	componentAdded()
			componentRemoved()

Component specific AWT Events

Event	**Fired by**	**Listener interface(s)**	**Handler Method(s)**
ActionEvent	TextField MenuItem List Button	ActionListener	actionPerformed()

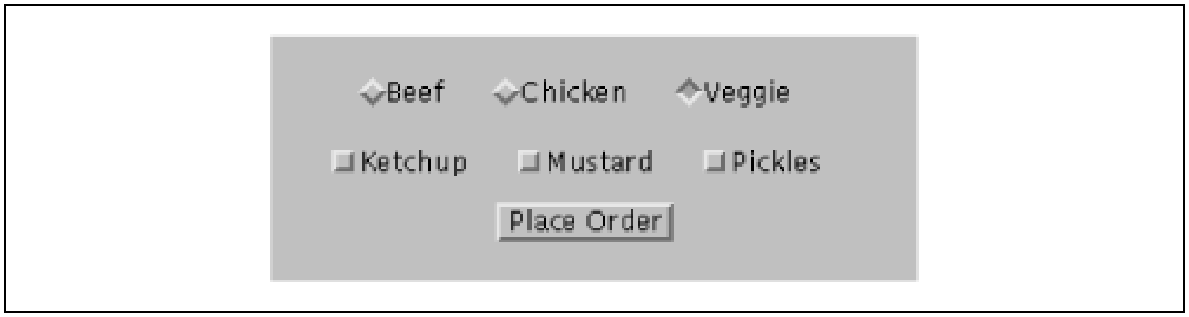
ItemEvent	List Checkbox Choice CheckboxMenuItem	ItemListener	itemStateChanged()
AdjustmentEvent	ScrollPane Scrollbar	AdjustmentListener	adjustmentValueChanged()
TextEvent	TextArea TextField	TextListener	textValueChanged()

WindowEvent	Frame, Dialog	WindowListener	windowOpened()
			windowClosing()
			windowClosed()
			windowIconified()
			windowDeiconified()
			windowActivated()
			windowDeactivated()

Adapter Classes

It's not ideal to have your application components implement a listener interface and receive events directly. Sometimes it's not even possible. Being an event receiver forces you to modify or subclass your objects to implement the appropriate event listener interfaces and add the code necessary to handle the events. A more subtle issue is that, since we are talking about AWT events here, you are, of necessity, building GUI logic into parts of your application that shouldn't have to know anything about the GUI. Let's look at an example:

Figure 10.10: Implementing the ActionListener interface



In Figure 10.10 we have drawn the plans for our Vegomatic food processor. Here we have made our Vegomatic object implement the `ActionListener` interface so that it can receive events directly from the three `Button` components: "Chop," "Puree," and "Frappe." The problem is that our Vegomatic object now has to know more than how to mangle food. It also has to be aware that it will be driven by three controls, specifically buttons that send action commands, and be aware of which methods in itself it should invoke for those commands. Our boxes labeling the GUI and Application code overlap in an unwholesome way. If the marketing people should later want to add or remove buttons, or perhaps just change the names, we have to be careful. We may have to modify the logic in our Vegomatic Object. All is not well.

An alternative is to place an "adapter" class between our event source and receiver. An adapter is a simple object whose sole purpose is to map an incoming event to an outgoing method.

Figure 10.11: A design using adapter classes

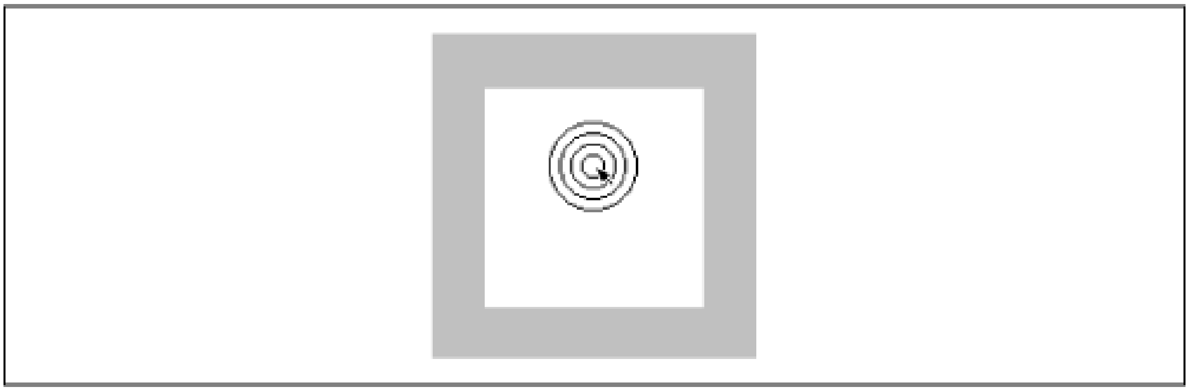


Figure 10.11 shows a better design that uses three adapter classes, one for each button. The implementation of the first adapter might look like:

```
class VegomaticAdapter1 implements actionListener {
    Vegotmatic vegomatic;
    VegomaticAdapter1 ( Vegotmatic vegomatic ) {
        this.vegomatic = vegomatic;
    }
    public void actionPerformed( ActionEvent e ) {
        vegomatic.chopFood();
    }
}
```

So somewhere in the code where we build our GUI, we could register our listener like so:

```
// building GUI for our Vegomatic
Vegomatic theVegomatic = ...;
Button chopButton = ...;
// make the hookup
chopButton.addActionListener( new VegomaticAdapter1(theVegomatic) );
```

We have completely separated the messiness of our GUI from the application code. However, we have added three new classes to our application, none of which does very much. Is that good? That depends on your vantage point.

Under different circumstances our buttons may have been able to share a common adapter class that was simply instantiated with different parameters. There are various trade-off that can be made between size, efficiency, and elegance of code. Often, adapter classes will be generated automatically by development tools. The way we have named our adapter classes "VegomaticAdapter1," "VegomaticAdapter2," and "VegomaticAdapter3" hints at this. More often, when hand coding, you'll use an inner class. At the other extreme, we can forsake Java's strong typing and use the reflection API to create a completely dynamic hookup betwen an event source and listener.

AWT Dummy Adapters

Many listener interfaces contain more than one event handler method. Unfortunately, this means that to register yourself as interested in any one of those events, you must implement the whole listener interface. And to accomplish this you might find yourself typing in dummy "stubbed out" methods, simply to complete the interface. There is really nothing wrong with this, but it is a bit tedious. To save you some trouble, AWT provides some helper classes that implement these dummy methods for you. For each listener interface containing more than one method there is an adapter class containing the stubbed methods. The adapter class serves as a base class for your own adapters. So, when you need a class to patch together your event source and listener, you can simply subclass the adapter and override only the methods you want.

For example, the `MouseAdapter` class implements the `MouseListener` interface and provides the following implementation:

```
public void mouseClicked(MouseEvent e) {};
public void mousePressed(MouseEvent e) {};
public void mouseReleased(MouseEvent e) {};
public void mouseEntered(MouseEvent e) {};
public void mouseExited(MouseEvent e) {};
```

This may not look like a tremendous time saver, and you're right. It's simply a bit of sugar. The primary advantage comes into play when we use the `MouseAdapter` as the base for your own adapter in an anonymous inner class. For example, suppose we want to catch a `mousePressed()` event in some component and blow up a building. We can use the following to make the hookup:

```
someComponent.addMouseListener( new MouseAdapter() {
    public void MousePressed(MouseEvent e) {
        building.blowUp();
    }
} );
```

We've taken artistic liberties with the formatting, but I think it's very readable, and I like it. It's a common enough activity that it's nice to avoid typing those extra few lines and perhaps stave off the onset of carpal tunnel syndrome for a few more hours. Remember that any time you use an inner class, the compiler is generating a class for you, so the messiness you've saved in your source still exists in the output classes.

Old Style and New Style Event Processing

Although Java is still a youngster, it has a bit of a legacy. Versions of Java before 1.1 used a different style of event delivery. Back in the old days (a few months ago) event types were limited and events were only delivered to the `Component` that generated it, or one of its parent containers. The old style component event handler methods (now deprecated) returned a boolean value declaring whether or not they had "handled" the event.

```
boolean handleEvent( Event e ) {
    ...
}
```

If the method returns false, the event is automatically redelivered to the component's container to give it a chance. If the container does not handle it, it is passed on to its parent container and so on. In this way, events were propogated up the containment hierarchy until they were either consumed or passed over to the component peer, just as current

InputEvents are ultimately interpreted used the peer if no registered event listeners have consumed them.

Although this style of event delivery was convenient for some simple applications, it is not very flexible. Events could only be handled by components, which meant that you always had to subclass a Component or Container type to handle events. This was a degenerate use of inheritance (bad design) that led to the creation of lots of unnecessary classes.

We could, alternatively, receive the events for many embedded components in a single parent container, but that would often lead to very convoluted logic in the container's event handling methods. It is also very costly to run every single AWT event through a guantlet of (often empty) tests as it traverses its way up the tree of containers. This is why Java now provides the more dynamic and general event source/listener model that we have described in this chapter. The old style events and event handler methods are, however, still with us.

Java is not allowed to simply change and break an established API. Instead, as we described in Chapter 1, *Yet Another Language?*, older ways of doing things are simply "deprecated" in favor of the new ones. This means that code using the old style event handler methods will still work; you may see old-style code around for a long time. The problem with relying on old-style event delivery, however, is that the old and new ways of doing things cannot be mixed.

By default, Java is obligated to perform the old behavior--offering events to the component and each of its parent containers. However, Java turns off the old style delivery whenever it thinks that we have elected to use the new style. Java determines whether a Component should recieve old style or new style events based on whether any event listeners are registered, or whether new style events have been explicitly enabled. When an AWT event listener is registered with a Component, new style events are implicitly turned on (a flag is set). Additionally, a mask is set telling the component the types of AWT events it should process. The mask allows components to be more selective about which events they process.

processEvent()

When new style events are enabled, all events are first routed to the dispatchEvent() method of the Component class. The dispatchEvent() method examines the component's event mask and decides whether the event should be processed or ignored. Events that have been "enabled" are sent to the processEvent() method, which simply looks at the event's type and delegates it to a "helper" processing method named for its type. The helper processing method finally dispatches the event to the set of registered listeners for its type.

This process closely parallels the way in which old style events are processed, and the way in which events are first directed to a single handleEvent() method that dispatches them to more specific handler methods in the Component class. The differences are that new style events are not delivered unless someone is listening for them, and the listener registration mechanism means that we don't have to subclass the component in order to override its event handler methods and insert our own code.

Enabling New Style Events Explicitly

Still, if you are subclassing a Component type for other reasons, or you really want to process all events in a single method, you should be aware that it is possible to emulate the old style event handling and override your component's event processing methods. You simply have to call the Component's enableEvents() method with the appropriate mask value to turn on processing for the given type of event. You can then override the corresponding method and insert your code. The mask values are found in the java.awt.AWTEvent class:

java.awt.AWTEvent mask	method
COMPONENT_EVENT_MASK	processComponentEvent()
FOCUS_EVENT_MASK	processFocusEvent()
KEY_EVENT_MASK	processKeyEvent()
MOUSE_EVENT_MASK	processMouseEvent()
MOUSE_MOTION_EVENT_MASK	processMouseMotionEvent()

For example:

```
    public void init() {
        ...
        enableEvent( AWTEvent.KEY_EVENT_MASK ):
    }
    public void processKeyEvent(KeyEvent e) {
        if ( e.getID() == KeyEvent.KEY_TYPED )
            // do work
        super.processKeyEvent(e);
    }
```

If you do this it is vital that you remember to make a call to `super.process...Event()` in order to allow normal event delegation to continue. Of course, by emulating old-style event handling, we're giving up the virtues of the new style; in particular, this code is a lot less flexible than the code we could write with the new event model. As we've seen, the user interface is hopelessly tangled with the actual work your program does. A compromise solution would be to have your subclass declare that it implements the appropriate listener interface and register itself, as we have done in the simpler examples in this book:

```
class MyApplet implements KeyListener ...

    public void init() {
        addKeyListener( this ):
        ...
    }
    public void keyTyped(KeyEvent e) {
        // do work
    }
```

11.2 Text Components

AWT gives us two basic text components: TextArea is a multiline text editor with vertical and horizontal scrollbars; TextField is a simple, single-line text editor. Both TextField and TextArea derive from the TextComponent class, which provides the functionality they have in common. This includes methods for setting and retrieving the displayed text, specifying whether the text is "editable" or read-only, manipulating the caret (cursor) position in the display, and manipulating the "selection text."

Both TextAreas and TextFields send TextEvents to listeners when their text is modified. To receive these events, you must implement the java.awt.TextListener interface and register by calling the component's addTextListener() method. In addition, TextField components generate an ActionEvent whenever the user presses the RETURN key within the field. To get these events, implement the ActionListener interface and call addActionListener() to register.

Here are a couple of simple applets that show you how to work with text areas and fields.

A TextEntryBox

Our first example, TextEntryBox, creates a TextArea and ties it to a TextField, as you can see in Figure 11.1.

When the user hits RETURN in the TextField, we receive an ActionEvent and add the line to the TextArea's display.

Try it out. You may have to click your mouse in the TextField to give it focus before typing in it. If you fill up the display with lines, you can test drive the scrollbar.

Figure 11.1: The TextEntryBox applet

[Graphic: Figure 11-1]

```
import java.awt.*;
import java.awt.event.*;
public class TextEntryBox extends java.applet.Applet
implements ActionListener {
    TextArea area;
    TextField field;

    public void init() {
        setLayout( new BorderLayout() );
        add( "Center", area = new TextArea() );
        area.setFont( new Font("TimesRoman",Font.BOLD,18) );
        area.setText("Howdy!\n");
        add( "South", field = new TextField() );
        field.addActionListener ( this );
    }
    public void actionPerformed(ActionEvent e) {
        area.append( field.getText() + '\n' );
        field.setText("");
    }
}
```

`TextEntryBox` is exceedingly simple; we've done a few things to make it more interesting. First, we set the applet's layout manager to `BorderLayout`. We use `BorderLayout` to position the `TextArea` above the `TextField`; the text area goes in the "North" region of the layout, and the text field is in the "South" region. We give the text area a bigger font using `Component`'s `setFont()` method; fonts are discussed in Chapter 11, *Using and Creating GUI Components*. Finally, we want to be notified whenever the user types RETURN in the text field, so we register our applet (`this`) as a listener for action events by calling `field.addActionListener(this)`.

Hitting RETURN in the `TextField` generates an action event, and that's where the fun begins. We handle the event in the `actionPerformed()` method of our container, the `TextEntryBox` applet. Then we use the `getText()` and `setText()` methods to manipulate the text the user has typed. These methods can be used for both `TextField` and `TextArea`, because these components are derived from the `TextComponent` class, and therefore have some common functionality.

Our event handler is called `actionPerformed()`, as required by the `ActionListener` interface. First, `actionPerformed()` calls `field.getText()` to read the text that the user typed into our `TextField`. It then adds this text to the `TextArea` by calling `area.append()`. Finally, we clear the text field by calling `field.setText("")`, preparing it for more input.

By default, `TextField` and `TextArea` are editable; you can type and edit in both text components. They can be changed to output-only areas with the `setEditable()` method. Both text components also support *selections*. A selection is a subset of text that is highlighted for copying and pasting in your windowing system. You select text by dragging the mouse over it; you can then copy and paste it into other text windows. You can get the selected text explicitly with `getSelectedText()`.

TextWatcher Applet

Our next example is a `TextWatcher` that consists of two linked text areas. Anything the user types into either area is reflected in both. It demonstrates how to handle a `TextEvent`, which is generated whenever the text changes in a `TextComponent`. It also demonstrates how to use an adapter class, which is a more realistic way of setting up event listeners. Registering the applet as a listener is okay for simple programs, but the technique shown here will serve you better in more complex situations.

Figure 11.2: The TextWatcher applet

[Graphic: Figure 11-2]

```
import java.awt.*;
import java.awt.event.*;
public class TextWatcher extends java.applet.Applet {
    TextArea area1, area2;

    public void init() {
        setLayout( new GridLayout(2,1) );
        add( area1 = new TextArea() );
        add( area2 = new TextArea() );
        area1.addTextListener ( new TextSyncAdapter( area2 ));
        area2.addTextListener ( new TextSyncAdapter( area1 ));
    }
    class TextSyncAdapter implements TextListener {
        TextArea targetArea;
        TextSyncAdapter( TextArea targetArea ) {
            this.targetArea = targetArea;
        }
        public void textValueChanged(TextEvent e) {
            TextArea sourceArea = (TextArea)e.getSource();
            if ( ! targetArea.getText().equals( sourceArea.getText() ) )
                targetArea.setText( sourceArea.getText() );
        }
    }
}
```

Setting up the display is simple. We use a `GridLayout` and add two text areas to the layout. Then we add our listeners for text events, which are generated whenever the text in a `TextComponent` is changed. There is one listener for each text area; both are `TextSyncAdapter` objects. One listens to events from `area1` and modifies the text in `area2`; the other listens to events from `area2` and modifies the text in `area1`.

All the real work is done by the `TextSyncAdapter`. This is an inner class; its definition is contained within `TextWatcher` and can't be referenced by classes outside of our `TextWatcher`. The adapter implements the `TextListener` interface, and therefore includes a `textValueChanged()` method.

`textValueChanged()` is the heart of the listener. First, it extracts the source area from the incoming event; this is the area that generated the event. The area whose text the listener is changing (the target area) was stored by the constructor. Then it tests whether or not the texts in both areas are the same. If they aren't, it calls the target area's `setText()` method to set its text equal to the source area's.

The one mysterious feature of this method is the test for equality. Why is it necessary? Why can't we just say "the text in one area changed, so set the text in the other?" A `TextEvent` is generated whenever a `TextComponent` is modified for any reason; this includes changes caused by software, not just changes that occur when a user types. So think about what happens when the user types into `area1`. Typing generates a `TextEvent`, which causes the adapter to change the text in `area2`. This generates another `TextEvent`, which if we didn't do any testing, would cause us to change area1 again, which would generate another `TextEvent`, ad infinitum. By checking whether or not the texts in our two areas are equivalent, we prevent an infinite loop in which text events ping-pong back and forth between the two areas.

Managing Text Yourself

Text areas and text fields do the work of handling keystrokes for you, but they're certainly not your only options for dealing with keyboard input. Any `Component` can register `KeyListener`s to recieve `KeyEvent`s that occur when their component has focus Chapter 10, *Understand the Abstract Windowing Toolkit*. We will provide an example here that shows how you might use these to make your own text gadgets.

Figure 11.3: The KeyWatcher applet

[Graphic: Figure 11-3]

```
import java.awt.*;
import java.awt.event.*;
public class KeyWatcher extends java.applet.Applet {
    StringBuffer text = new StringBuffer();
    public void init () {
        setFont( new Font("TimesRoman",Font.BOLD,18) );
        addKeyListener ( new KeyAdapter() {
            public void keyPressed( KeyEvent e ) {
                System.out.println(e);
                type( e.getKeyCode(), e.getKeyChar() );
            }
        } );
        requestFocus();
    }
    public void type(int code, char ch ) {
        switch ( code ) {
            case ( KeyEvent.VK_BACK_SPACE ):
                if (text.length() > 0)
                    text.setLength( text.length() - 1 );
                break;
            case ( KeyEvent.VK_ENTER ):
                    System.out.println( text );  // Process line
                    text.setLength( 0 );
                break;
            default:
                if ( (ch >= ' ') && (ch <= '~') )
                    text.append( ch );
        }
        repaint();
    }

    public void paint(Graphics g) {
        g.drawString(text.toString() + "_", 20, 20);
    }
}
```

Fundamentally, all we are doing is collecting text in a `StringBuffer` and using the `drawString()` method to display it on the screen. As you'd expect, paint() is responsible for managing the display.

In this applet, we're interested in receiving `KeyEvent`s, which occur whenever the user presses any key. To get these events, the applet calls its own `addKeyListener()` method. The `KeyListener` itself is an anonymous class. It doesn't have a name and can't be used anywhere else. We create this class by getting a new `KeyAdapter()`, and overriding its `keyPressed()` method. (Remember that `KeyAdapter` contains do-nothing implementations of the methods in the `KeyListener` interface.) All `keyPressed()` does is call our private method type(), with two arguments: the key code of the key that was pressed, and the logical character represented by the keystroke.

`type()` shows you how to deal with keystrokes. Each key event is identified with a code, which identifies the actual key typed, and a character, which identifies what the user meant to type. This sounds confusing, but it isn't. Basically, there is a constant for each key on the keyboard: `VK_ENTER` for the ENTER (return) key, `VK_A` for the letter A, and so on. However, the physical keystroke isn't usually the same as what the user types: the character capital A is made up of two keystrokes, while lower case a is made up of one.

Therefore, you can expect events for both physical keystrokes and typed characters. The `int` constant `VK_UNDEFINED` is used for the physical key code when the event doesn't correspond to a single keystroke. The `char` constant `CHAR_UNDEFINED` indicates that the event corresponds to a physical keystroke, but not a typed character.

The `type()` method is called with both the key constant and the character as arguments. The way we use them is relatively simple and would need more work for an industrial strength program. Simply, if the physical key is `VK_BACK_SPACE`, we delete the last character from the `StringBuffer` we're building. If it's `VK_ENTER`, we clear the `StringBuffer`. If the physical key has any other value, we look at the character the user typed. If this is a printable character, we add it to the `StringBuffer`. Anything else we ignore. Once we've handled the event, we call `repaint()` to update the screen. Using key codes to handle operations like "Backspace" or "Enter" is easier and less bug-prone than working with odd "Control" characters.

A final note on our anonymous adapter class: in essence our adapter is letting us write a "callback," by calling `type()` whenever `keyPressed()` is called. That's one important use for adapters: to map methods in the various listener interfaces into the methods that make sense for your class. Unlike C or C++, Java won't let us pass a method pointer as an argument, but it will let us create an anonymous class that calls our method and passes an instance of that class.

PREVIOUS | HOME | NEXT
Buttons and Labels | BOOK INDEX | Lists

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

11.3 Lists

A List is a step up on the evolutionary chain. Lists let the user choose from a group of alternatives. They can be configured to force the user to choose a single selection or to allow multiple choices. Usually, only a small group of choices are displayed at a time; a built-in scrollbar lets you move to the choices that aren't visible.

A List generates two kinds of events. If the user single clicks on a selection, the List generates an ItemEvent. If the user double-clicks, a List generates an ActionEvent. Therefore, a List can register both ItemListeners and ActionListeners. In either case, the listener can use the event to figure out what the user selected.

The applet below, SearchableListApplet, contains a List and a text field. Several of the items in the list aren't visible, because the list is too long for the space we've allotted for it (enough to display three items). When you type the name of an item into the text field, the applet displays the item you want and selects it. Of course, you could do this with a scrollbar, but then we wouldn't have the opportunity to demonstrate how to work with lists.

Figure 11.4: The SearchableList applet

[Graphic: Figure 11-4]

```
import java.awt.*;
import java.awt.event.*;
public class SearchableListApplet extends java.applet.Applet {
    public void init() {
        String [] items = { "One", "Two", "Three", "Four", "Five", "Six" };
        add( new SearchableList( items ) );
    }
}
class SearchableList extends Container implements ActionListener {
    List list;
    TextField field;
    SearchableList( String [] items ) {
        list = new List( 3 );  // Let some scroll for this example
        for(int i=0; i< items.length; i++)
            list.addItem( items[i] );
        field = new TextField();
        field.addActionListener( this );
        setLayout( new BorderLayout() );
        add("Center", list);
        add("South", field);
    }
    public void actionPerformed( ActionEvent e ) {
        String search = field.getText();
        for (int i=0; i< list.getItemCount(); i++)
            if ( list.getItem( i ).equals( search ) ) {
                list.select( i );
                list.makeVisible( i );  // Scroll it into view
                break;
            }
        field.setText(""); // clear the text field
    }
}
```

We create the `List` and the `TextField` in a new class, `SearchableList`; the applet itself only displays the `SearchableList`. `SearchableList` itself is a new kind of animal; it is a lightweight component that subclasses `Container` directly. We'll talk a little more about lightweight components later in the chapter.

In the constructor for `SearchableList`, we create our `List` by calling its constructor, setting it to display at most three components. We then call the `addItem()` method to add the possible selections to the list; these are the numbers "One" through "Six," passed to us in an array.

We then create our `TextField`, and register ourselves (i.e., the `SearchableList`) as an `ActionListener` for the field's events. Finally, we set the layout for `SearchableList` to a border layout, put the `List` in the center of the layout, and the `TextField` at the bottom.

The actionPerformed() method is called whenever the user presses RETURN in our TextField. In this method, we call getText() to extract the text the user typed. Then we loop through all the items in the list to see if there's a match. getItemCount() tells us the number of items in the list; getItem() gives us the text associated with any particular item. When we find a match, we call select() to make the matching item the selected item, and we call makeVisible() to make sure that this item is displayed.

By default, a List only allows a single selection. We've done nothing in this example to allow multiple selections, so whenever a user chooses an item, the previous selection is automatically dropped. If you want a list that supports multiple selections, call setMultipleMode(true). In this case, you must use the deselect() method to clear the user's selections.

11.4 Menus and Choices

A Menu is a standard, pull-down menu with a fixed name. Menus can hold other menus as submenu items, letting you implement complex menu structures. Menus come with several restrictions; they must be attached to a menu bar, and the menu bar can be attached only to a Frame (or another menu). You can't stick a Menu at an arbitrary position within a container. A top-level Menu has a name that is always visible in the menu bar. (An important exception to these rules is the PopupMenu, which we'll describe in the next section.)

A Choice is an item that lets you choose from a selection of alternatives. If this sounds like a menu, you're right. Choices are free-spirited relatives of menus. A Choice item can be positioned anywhere, in any kind of container. It looks something like a button, with the current selection as its label. When you press the mouse button on a choice, it unfurls to show possible selections.

Both menus and choices send action events when an item is selected. We'll create a little example that illustrates choices and menus and demonstrates how to work with the events they generate. Since a Menu has to be placed in the menu bar of a Frame, we'll take this opportunity to show off a Frame object as well. DinnerMenu pops up a window containing a **Food** choice and a menu of **Utensils**, as shown in Figure 11.2. DinnerMenu prints a message for each selection; choosing **Quit** from the menu removes the window. Give it a try.

Figure 11.5: The DinnerMenu applet

[Graphic: Figure 11-5]

```
import java.awt.*;
import java.awt.event.*;
import java.util.EventListener;
public class DinnerMenu extends java.applet.Applet {
    public void init() {
        new DinnerFrame().show();
    }
}
class DinnerFrame extends Frame implements ActionListener, ItemListener {
    DinnerFrame() {
        setTitle("Dinner Helper");
        setLayout( new FlowLayout() );
        add( new Label("Food") );
        Choice c = new Choice ();
        c.addItem("Chinese");
        c.addItem("Italian");
        c.addItem("American");
        c.addItemListener( this );
        add( c );
        Menu menu = new Menu("Utensils");
        menu.add( makeMenuItem("Fork") );
        menu.add( makeMenuItem("Knife") );
        menu.add( makeMenuItem("Spoon") );
        Menu subMenu = new Menu("Hybrid");
        subMenu.add( makeMenuItem("Spork") );
        subMenu.add( makeMenuItem("Spife") );
        subMenu.add( makeMenuItem("Knork") );
        menu.add( subMenu);
        menu.add( makeMenuItem("Quit") );
        MenuBar menuBar = new MenuBar();
        menuBar.add(menu);
        setMenuBar(menuBar);
        pack();
    }
    public void itemStateChanged(ItemEvent e) {
        System.out.println( "Choice set to: " + e.getItem() );
    }
    public void actionPerformed(ActionEvent e) {
        String command = e.getActionCommand();
        if ( command.equals( "Quit" ) )
            dispose();
        else
            System.out.println( "Menu selected: " + e.getActionCommand() );
    }
    private MenuItem makeMenuItem( String name ) {
        MenuItem m = new MenuItem( name );
        m.addActionListener( this );
        return m;
    }
}
```

Yes, I know. **Quit** doesn't belong in the **Utensils** menu. If it's driving you crazy, you can go back and add a **File** menu as an exercise when we're through.

So what do we have? Well, we've created a new kind of component called `DinnerFrame` that implements our palette of dinner options. We do our set-up work in the `DinnerFrame` constructor. `DinnerFrame` sets the name on its titlebar with `setTitle()`. The constructor also handles a few other miscellaneous details, such as setting the layout manager that places things side by side in the display area and later, resizing itself by calling `pack()`.

We make an instance of `Choice` and add three options to it with its `addItem()` method. `Choice` options are simple `String` objects. When one is picked, we get an action event with an argument that specifies the selected option name. We can also examine the currently selected item at any time with the `Choice`'s `getSelectedItem()` method. A `Choice` generates an `ItemEvent` when a user makes a selection, so we register the `DinnerFrame` as an `ItemEvent` listener by calling `addItemListener()`. (This means we must also implement the `ItemListener` interface and provide an `itemStateChanged()` method.) As with any component, we display the `Choice` by adding it to our applet's layout with `add()`.

The `Menu` has a few more moving parts. A `Menu` holds `MenuItem` objects. A simple `MenuItem` just has a `String` as a label. It sends this as an argument in an action event when it's selected. We can set its font with `setFont()`. We can also turn it on or off with `setEnabled()`; this method controls whether the `MenuItem` is available or not. A `Menu` object is itself a kind of `MenuItem`, and this allows us to use a menu as a submenu in another menu.

We construct the `Menu` with its name and call its `add()` method to give it three new `MenuItem` objects. To create the menu items, we call our own `makeMenuItem()` helper method. Next, we repeat this process to make a new `Menu` object, `subMenu`, and add it as the fourth option. Its name appears as the menu item along with a little arrow, indicating it's a submenu. When it's selected, the `subMenu` menu pops up to the side and we can select from it. Finally, we add one last simple menu item, to serve as a **Quit** option.

We use a private method, `makeMenuItem()`, to create our menu items for us. This method is convenient because, with menus, every item generates its own events. Therefore, we must register an `ActionListener` for every selection on the menu. Rather than write lots of code, we use a helper method to register our `DinnerFrame` (`this`) as the listener for every item. It should be no surprise then, that `DinnerFrame` must implement `ActionListener` and provide an `actionPerformed()` method.

Now we have the menu; to use it, we have to insert it in a menu bar. A `MenuBar` holds `Menu` objects. We create a `MenuBar` and set it as the menu bar for `DinnerFrame` with the `Frame.setMenuBar()` method. We can then add our menu to it with `menuBar.add()`:

```
MenuBar menuBar = new MenuBar();
menuBar.add(menu);
setMenuBar(menuBar);
```

Suppose our applet didn't have its own frame? Where could we put our menu? Ideally, you'd like the applet to be able to add a menu to the top-level menu bar of the Web browser or applet viewer. Unfortunately, as of Java 1.1, there is no standard for doing so. (There are obvious security considerations in allowing an applet to modify its viewer.) There has been talk of adding this ability as part of Java Beans, but so far, it's not possible.

One final note about the `DinnerMenu` example. As we said in the previous chapter, any time you use a `Frame`, and thus create a new top-level window, you should add code to destroy your frame whenever the user closes the window with his native window manager. To do so, you register your frame as a `WindowListener`, implement the `WindowListener` interface, and provide a `windowClosing()` method that calls `dispose()`. By calling `dispose()`, we indicate the window is no longer needed, so that it can release its window-system resources.

11.5 PopupMenus

One of the new components in Java 1.1 is the `PopupMenu`: a menu that automatically appears when you press the appropriate mouse button inside of a component. Which button you press depends on the platform you're using; fortunately you don't have to care.

The care and feeding of a `PopupMenu` is basically the same as any other menu. You use a different constructor (`PopupMenu()`) to create it, but otherwise, you build a menu and add elements to it the same way. The big difference is that you don't need to attach it to a `MenuBar`, and consequently don't need to worry about putting the `MenuBar` in a `Frame`. Instead, you call `add()` to add the `PopupMenu` to any component.

The `PopupColorMenu` applet contains three buttons. You can use a `PopupMenu` to set the color of each button or the applet itself, depending on where you press the mouse. (Setting the color of the applet also sets the buttons' colors). Figure 11.3 shows the applet in action; the user is preparing to change the color of the right-most button.

Figure 11.6: The PopupColorMenu Applet

[Graphic: Figure 11-6]

```
import java.awt.*;
import java.awt.event.*;
public class PopUpColorMenu extends java.applet.Applet
```

```
implements ActionListener {
    PopupMenu colorMenu;
    Component selectedComponent;
    public void init() {
        add( new Button("One") );
        add( new Button("Two") );
        add( new Button("Three") );

        colorMenu = new PopupMenu("Color");
        colorMenu.add( makeMenuItem("Red") );
        colorMenu.add( makeMenuItem("Green") );
        colorMenu.add( makeMenuItem("Blue") );
        addMouseListener( new MouseAdapter() {
            public void mousePressed(MouseEvent e) {
                if ( e.isPopupTrigger() ) {
                    selectedComponent = getComponentAt( e.getX(), e.getY() );
                    colorMenu.show(e.getComponent(), e.getX(), e.getY());
                }
            }
        } );
        add(colorMenu);
    }
    public void actionPerformed(ActionEvent e) {
        String color = e.getActionCommand();
        if ( color.equals("Red") )
            selectedComponent.setBackground( Color.red );
        else if ( color.equals("Green") )
            selectedComponent.setBackground( Color.green );
        else if ( color.equals("Blue") )
            selectedComponent.setBackground( Color.blue );
    }
    private MenuItem makeMenuItem(String label) {
        MenuItem item = new MenuItem(label);
        item.addActionListener( this );
        return item;
    }
}
```

Because the popup menu is triggered by mouse events, we need to register a `MouseListener`. In our call to `addMouseListener()`, we create an anonymous inner class based on the `MouseAdapter`. In this class, we override the `mousePressed()` method to display the popup menu when we get an appropriate event. How do we know what an "appropriate event" is? Fortunately, we don't need to worry about the specifics of our user's platform; we just need to call the event's `isPopupTrigger()` method. If this method returns `true`, we know the user has done whatever normally displays a popup menu on his system.

Once we know that the user wants to raise a popup menu, we figure out which component the mouse is over by calling `getComponentAt()`, with the coordinates of the mouse click (given by `e.getX()` and `e.getY()`).

Then we display the popup menu by calling its `show()` method, again with the mouse coordinates as arguments.

If we wanted to provide different menus for different types of components or the background, we could add a test within the check for the popup trigger:

```
if ( e.isPopupTrigger() ) {
    selectedComponent = getComponentAt( e.getX(), e.getY() );

    if ( selectedComponent instanceof Button )
         colorMenu.show(e.getComponent(),
                        e.getX(), e.getY());
    else if ( selectedComponent instanceof Applet )
            // show a menu for the background
    else if ( selectedComponent instanceof someOtherComponent )
            // show another menu
}
```

The only thing left is to handle the action events from the popup menu items. As in our earlier example, we use a helper method called `makeMenuItem()` to register the applet as an action listener for every item we add. The applet implements `ActionListener` and has the required `actionPerformed()` method. This method reads the action command from the event, which is equal to the selected menu item's label by default. It then sets the background color of the selected component appropriately.

11.6 Checkboxes and CheckboxGroups

A `Checkbox` is a labeled toggle button. A group of such toggle buttons can be made mutually exclusive by tethering them together with a `CheckboxGroup` object. By now you're probably well into the swing of things and could easily master the `Checkbox` on your own. We'll throw out an example to illustrate a different way of dealing with the state of components and to show off a few more things about containers.

A `Checkbox` sends item events when it's pushed, just like a `List`, a `Menu`, or a `Choice`. In our last example, we caught the action events from our choice and menu selections and worked with them when they happened. For something like a checkbox, we might want to be lazy and check on the state of the buttons only at some later time, such as when the user commits an action. It's like filling out a form; you can change your choices until you submit the form.

The following applet, `DriveThrough`, lets us check off selections on a fast food menu, as shown in Figure 11.4. `DriveThrough` prints the results when we press the **Place Order** button. Therefore, we can ignore all the item events generated by our checkboxes and listen only for the action events generated by the button.

Figure 11.7: The DriveThrough applet

[Graphic: Figure 11-7]

```
import java.awt.*;
import java.awt.event.*;
public class OrderForm extends java.applet.Applet implements ActionListener {
    Panel condimentsPanel = new Panel();
    CheckboxGroup entreeGroup = new CheckboxGroup();
    public void init() {
        condimentsPanel.add( new Checkbox("Ketchup"));
        condimentsPanel.add( new Checkbox("Mustard"));
        condimentsPanel.add( new Checkbox("Pickles"));
        Checkbox c;
        Panel entreePanel = new Panel();
        entreePanel.add( c = new Checkbox("Beef") );
        c.setCheckboxGroup( entreeGroup );
        entreePanel.add( c = new Checkbox("Chicken") );
        c.setCheckboxGroup( entreeGroup );
        entreePanel.add( c = new Checkbox("Veggie") );
        c.setCheckboxGroup( entreeGroup );
        entreeGroup.setCurrent( c );
        Panel orderPanel = new Panel();
        Button orderButton = new Button("Place Order");
        orderButton.addActionListener( this );
        orderPanel.add( orderButton );
        setLayout( new GridLayout(3, 1) );
        add( entreePanel );
        add( condimentsPanel );
        add( orderPanel );
    }
    public void actionPerformed(ActionEvent e) {
        takeOrder();
    }
    void takeOrder() {
        Checkbox c = entreeGroup.getCurrent();
            System.out.println( c.getLabel() + " sandwich" );
        Component [] components = condimentsPanel.getComponents();
        for (int i=0; i< components.length; i++)
            if ( (c = (Checkbox)components[i]).getState() )
                System.out.println( "With " + c.getLabel() );
        System.out.println("Thank you, drive through...");
    }
}
```

DriveThrough lays out two panels, each containing three checkboxes. The checkboxes in the entreePanel are tied together through a single CheckboxGroup object. We call their setCheckboxGroup() methods to put them in a single CheckboxGroup that makes the checkboxes mutually exclusive. The CheckboxGroup object is an odd animal. One expects it to be a container or a component, but it isn't; it's simply a helper object that coordinates the functionality of the Checkbox objects. Because a CheckboxGroup isn't a container, it doesn't have an add() method. To put a checkbox into a group, you call the setCheckboxGroup() method of the Checkbox class.

Once a set of checkboxes have been placed in a checkbox group, only one of the boxes may be checked at a time. In this applet, the checkbox group forces you to choose a beef, chicken, or veggie entree, but not more than one.

The condiment choices, however, aren't in a checkbox group, so you can request ketchup, mustard, and pickles on your chicken sandwich.

When the **Place Order** button is pushed, we receive an `ActionEvent` via our `actionPerformed()` method. At this point, we gather the information in the checkboxes and print it. `actionPerformed()` simply calls our `takeOrder()` method, which reads the checkboxes. We could have saved references to the checkboxes in a number of ways; this example demonstrates two. First, we find out which entree was selected. To do so, we call the `CheckboxGroup`'s `getCurrent()` method. `getCurrent()` returns the selected `Checkbox`; we use `getLabel()` to extract the entree's name.

To find out which condiments were selected, we use a more complicated procedure. The problem is that condiments aren't mutually exclusive, so we don't have the convenience of a `CheckboxGroup`. Instead, we ask the condiments `Panel` for a list of its components. The `getComponent()` method returns an array of references to the container's child components. We'll use this to loop over the components and print the results. We cast each element of the array back to `Checkbox` and call its `getState()` method to see if the checkbox is on or off. Remember that if we were dealing with different types of components, we could determine what kind of component we had with the `instanceof` operator.

11.7 ScrollPane and Scrollbars

One of the big advantages of Java 1.1 is the addition of a ScrollPane container. Previously, unless you were working with a text component, you had to manage scrolling yourself. It wasn't terribly difficult, but it was a pain: you had to create Scrollbar objects, attach them to whatever you were scrolling, and redisplay everything with new positions whenever the user made an adjustment. ScrollPane does it all for you. About the only time you absolutely need a Scrollbar is when you want to create a "volume control-type" object. For example, you might want to create a paint mixer that blended different amounts of red, blue, and green, depending on some scrollbar settings.

The unifying theme behind both ScrollPane and Scrollbar is the Adjustable interface, which defines the responsibilities of scrollable objects. An object that implements Adjustable lets you modify an integer value through some fixed range. When a user changes the value, the object generates an AdjustmentEvent; as you might expect, to get an AdjustmentEvent, you must implement AdjustmentListener and register by calling addAdjustmentListener(). Scrollbars implement Adjustable, and a ScrollPane can return Adjustable objects for each of its scrollbars.[2]

In this section, we'll demonstrate both the ScrollPane and Scrollbar classes. We'll start with a ScrollPane.

Working With A ScrollPane

Technically, a ScrollPane is a Container, but it's a funny one. It has its own layout manager, which can't be changed. It can only accomodate one component at a time. This seems like a big limitation, but it isn't. If you want to put a lot of stuff in a ScrollPane, just put your components into a Panel, with whatever layout manager you like, and put that panel into the ScrollPane.

When you create a ScrollPane, you can specify the conditions under which its srollbars will be displayed. This is called the scrollbar display policy; you can specify the policy by using one of the three constants below as an argument to the ScrollPane constructor.

SCROLLBARS_AS_NEEDED

> Only displays scrollbars if the object in the ScrollPane doesn't fit.

> [2] There may be a bug in the Adjustable objects you get from a ScrollPane. Although you can read their settings, you can't change them; methods like setMinimum() and setMaximum() (which should set the object's minimum and maximum values) throw an AWTError.

SCROLLBARS_ALWAYS

Always displays scrollbars, regardless of the object's size.

SCROLLBARS_NEVER

Never displays scrollbars, even if the object is too big. If you use this policy, you should provide some other way for the user to manipulate the ScrollPane.

By default, the policy is SCROLLBARS_AS_NEEDED.

Here's an applet that uses a ScrollPane to display a large image. As you'll see, the applet itself is very simple; all we do is get the image, set the applet's layout manager, create a ScrollPane, put the image in our pane, and add the ScrollPane to the applet. To make the program slightly cleaner, we create an ImageComponent component to hold the image, rather than placing the image directly into the ScrollPane. Here's the applet itself:

```
import java.awt.*;
public class ScrollPaneApplet extends java.applet.Applet {
    public void init() {
        Image image = getImage( getClass().getResource(getParameter("image")) );
        setLayout(  new BorderLayout() );
        ScrollPane scrollPane = new ScrollPane();
        scrollPane.add( new ImageComponent(image) );
        add( "Center", scrollPane );
    }
}
```

And here's the ImageComponent. It waits for the image to load, using a MediaTracker, and sets its size to the size of the image. It also provides a paint() method to draw the image. This takes a single call to drawImage(). The first argument is the image itself; the next two are the coordinates of the image relative to the ImageComponent; and the last is a reference to the ImageComponent itself (this), which serves as an image observer. (We'll discuss image observers in Chapter 14, *Working With Images*; for the time being, take this on faith.)

We also supply an update() method that calls paint(). As we'll see later, the default version of update() automatically clears the screen, which wastes time if we already know that our image will cover the entire screen. Therefore, we override update() so that it doesn't bother clearing the screen first.

Finally, ImageComponent provides a getPreferredSize() method, overriding the method it inherits from Component. This method simply returns the image's size, which is a Dimension object. When you're using a ScrollPane, it's important for the object you're scrolling to provide a reliable indication of its size, particularly if the object is a lightweight component.

```
import java.awt.*;
class ImageComponent extends Component {
    Image image;
    Dimension size;
    ImageComponent ( Image image ) {
        this.image = image;
        MediaTracker mt = new MediaTracker(this);
        mt.addImage( image, 0 );
        try { mt.waitForAll(); } catch (InterruptedException e) { /* error */ };
        size = new Dimension ( image.getWidth(null), image.getHeight(null) );
        setSize( size );
    }
    public void update( Graphics g ) {
        paint(g);
    }
    public void paint( Graphics g ) {
        g.drawImage( image, 0, 0, this );
    }
    public Dimension getPreferredSize() {
        return size;
    }
}
```

Using Scrollbars

Our next example is basically the same as the previous, except that it doesn't use the `ScrollPane`; it implements its own scroller using scrollbars. With Java 1.1, you'd never write code like this, but it does show how much work the `ScrollPane` saves, and also demonstrates how to use scrollbars in other situations.

Figure 11.8: The ComponentScrollerApplet

[Graphic: Figure 11-8]

Our applet is called `ComponentScrollerApplet`; it uses a homegrown scroll pane called `ComponentScroller`. The component that we scroll is the `ImageComponent` we developed in the previous example.

Now let's dive into the code for `ComponentScrollerApplet`:

```
import java.awt.*;
import java.awt.event.*;
public class ComponentScrollerApplet extends java.applet.Applet {
    public void init() {
        Image image = getImage( getClass().getResource(getParameter("image")) );
        ImageComponent canvas = new ImageComponent( image );
        setLayout(  new BorderLayout() );
        add( "Center", new ComponentScroller(canvas) );
    }
}
class ComponentScroller extends Container {
    Scrollbar horizontal, vertical;
    Panel scrollarea = new Panel();
    Component component;
    int orgX, orgY;
    ComponentScroller( Component comp ) {
        scrollarea.setLayout( null );  // We'll handle the layout
        scrollarea.add( component = comp );
        horizontal = new Scrollbar( Scrollbar.HORIZONTAL );
        horizontal.setMaximum( component.getSize().width );
        horizontal.addAdjustmentListener( new AdjustmentListener() {
            public void adjustmentValueChanged(AdjustmentEvent e) {
                component.setLocation( orgX = -e.getValue(), orgY );
            }
        } );
        vertical = new Scrollbar( Scrollbar.VERTICAL );
        vertical.setMaximum( component.getSize().height);
        vertical.addAdjustmentListener( new AdjustmentListener() {
            public void adjustmentValueChanged(AdjustmentEvent e) {
                component.setLocation( orgX, orgY = -e.getValue() );
            }
        } );
        setLayout( new BorderLayout() );
        add( "Center", scrollarea );
        add( "South", horizontal );
        add( "East", vertical );
    }
    public void doLayout() {
        super.doLayout();
        horizontal.setVisibleAmount( scrollarea.getSize().width );
        vertical.setVisibleAmount( scrollarea.getSize().height );
    }
}
```

So, what do our new components do? Let's start at the top and work our way down. The applet itself is very simple; it does all of its work in `init()`. First it sets its layout manager to `BorderLayout`. Then it acquires an `Image` object with a call to `getImage()`. Finally, the applet creates an `ImageComponent` to hold our image, creates a

`ComponentScroller` to hold the `ImageComponent`, and adds the scroller to the "Center" region of the layout. I chose `BorderLayout` because it resizes its central component to fill the entire area available.

Next comes the `ComponentScroller` itself. `ComponentScroller` takes a reference to our `ImageComponent` in its constructor. It adds the component it will be scrolling to a `Panel` with no layout manager. It then creates horizontal and vertical `Scrollbar` objects (`HORIZONTAL` and `VERTICAL` are constants of the `Scrollbar` class, used to specify a scrollbar's direction), sets their maximum values using the height and width of the `Panel`, and registers an `AdjustmentListener` for each scrollbar. The `AdjustmentListener` is an anonymous inner class that implements the `adjustmentValueChanged()` method. This method is called whenever the user moves the scrollbar. It extracts the new scrollbar setting from an `AdjustmentEvent` and uses this to move the component we're scrolling to its new location. We have a separate listener for each scrollbar, so we don't have to figure out which scrollbar generated the event. The listener for the horizontal scrollbar adjusts the component's x coordinate (`orgX`) and leaves its y coordinate unchanged; likewise, the listener for the vertical scrollbar adjusts the component's y coordinate. By adjusting the location of the `ImageComponent`, we control how much of the image is displayed; anything that falls outside of the scroller's `Panel` (`scrollarea`) isn't displayed.

The `ComponentScroller` overrides the `doLayout()` method of the `Container` class. This gives us an opportunity to change the size of the scrollbar "handles" whenever the scroller is resized. To do so, we call `super.doLayout()` first, to make sure that the container gets arranged properly; although we're overriding this method, we need to make sure that it does its work. Then we call the `setVisibleAmount()` method of each scrollbar with the new width and height of the scrolling area.

So in review: we call `setMaximum()` to set the vertical scrollbar's maximum value to the image's height; we call `setVisibleAmount()` to tell the vertical scrollbar how much area we have available; and it sets the size of its "handle" accordingly. For example, if the image is 200 pixels high, and the visible amount is 100 pixels, the scrollbar sets its handle to be roughly half its length. We do similar things to the horizontal scrollbar. As a result, the handles grow or shrink as we change the size of the viewing area and indicate how much of the image is visible.

The `setMaximum()` and `setVisibleAmount()` are both part of the `Adjustable` interface, which scrollbars implement. Other methods of this interface are:

`getOrientation()`	Tells you whether the scrollbar is HORIZONTAL or VERTICAL. There is no `setOrientation()` method in the interface, but the `Scrollbar` class does support `setOrientation()`.
`getVisibleAmount()` and `setVisibleAmount()`	Lets you control the size of the scrollbar's handle (slider). As we saw above, this is a convenient way to give the user a feel for the size of the object you're scrolling.
`getValue()` and `setValue()`	Lets you retrieve or change the scrollbar's current setting.
`getMinimum()` and `setMinimum()`	Lets you control the scrollbar's minimum value.
`getMaximum()` and `setMaximum()`	Lets you control the scrollbar's maximum value.

`getUnitIncrement()` and `setUnitIncrement()`	Lets you control the amount the scrollbar will move if the user clicks on the scrollbar's arrows; in many environments, this means the user wants to move up or down one line.
`getBlockIncrement()` and `setBlockIncrement()`	Lets you control the amount the scrollbar will move if the user clicks between an arrow and the slider; in many environments, this means the user wants to move up or down one page.
`addAdjustmentListener()` and `removeAdjustmentListener()`	Adds or removes listeners for the scrollbar's events.

It's worth asking why we put our image in a `Canvas`, which we then put into a `Panel`, which we put into another `Panel`, which we put into the applet. Surely there's a more efficient way. Yes there is, but we wanted to make as many reusable components as possible. With this design, you can use `ImageComponent` wherever you need to display an image and check that it is loaded first; you can use `ComponentScroller` wherever you need to scroll any kind of component, not just an image or a `Canvas`. Making resuable components is one of the big advantages of object oriented design; it's something you should always keep in mind.

11.8 Dialogs

A `Dialog` is another standard feature of user interfaces. In Java, a `Dialog` is a kind of `Window`, which means that it's really a container into which you put other components. A `Dialog` can be either *modal* or *nonmodal*. A modal dialog seizes the attention of the user by staying in the foreground and grabbing all input until it is satisfied. A non-modal dialog isn't quite so insistent; you're allowed to do other things before dealing with the dialog. `Dialog` objects are useful for pop-up messages and queries or important user-driven decisions.

Most of the components we've seen so far have some special kinds of events associated with them. A `Dialog` doesn't have any special events. Of course, this doesn't mean that a dialog doesn't generate events. Since a dialog is a `Window`, it can generate any event that a `Window` can. However, there aren't any special events, like action events or item events, to worry about. When you're dealing with a `Dialog`, your primary concern will be events generated by the components that you put into the `Dialog`.

We'll do a quick example of a `Dialog` window and then take a look at `FileDialog`, a subclass of `Dialog` that provides an easy-to-use file-selector component. Our example will be a modal dialog that asks a simple question:

A Simple Query Dialog

```
import java.awt.*;
import java.awt.event.*;
class ModalYesNoDialog extends Dialog implements ActionListener {
    private boolean isYes = false;
    ModalYesNoDialog( Frame frame, String question ) {
        super(frame, true /* modal */);
        Label label = new Label(question);
        label.setFont( new Font("Dialog",Font.PLAIN,20) );
        add( "Center", label );
        Panel yn = new Panel();
        Button button = new Button("Yes");
        button.addActionListener( this );
        yn.add( button );
        button = new Button("No");
        button.addActionListener( this );
        yn.add( button );
        add("South", yn);
        pack();
    }
    synchronized public boolean answer() {
        return isYes;
    }
    synchronized public void actionPerformed ( ActionEvent e ) {
        isYes = e.getActionCommand().equals("Yes");
        dispose();
    }
    public static void main(String[] s) {
        Frame f = new Frame();
        f.add( "Center", new Label("I'm the application") );
        f.add( "South", new Button("Can you press me?") );
        f.pack();
        f.show();
        ModalYesNoDialog query = new ModalYesNoDialog( f, "Do you love me?");
        query.show();
        if ( query.answer() == true )
            System.out.println("She loves me...");
        else
            System.out.println("She loves me not...");
    }
}
```

The heart of this example is a class called `ModalYesNoDialog` that implements a simple form with a question and two buttons. To create the `Dialog`, our class's constructor calls its superclass's constructor (`super()`), which is `Dialog()`. When we create the dialog, we must supply a parent `Frame`; we also specify that the `Dialog` is modal.

The rest of the constructor--for that matter, the rest of the class--doesn't have any surprises. We use a `Label` to display the question; we add a pair of buttons, labeled "Yes" and "No," for the user to give his answer. We provide an `answer()` method so we can ask the dialog which button the user pushed; and we provide an `actionPerformed()` method to receive the button events.

The rest of our program is an application that uses the `ModalYesNoDialog`. It creates a `Frame`, creates the `ModalYesNoDialog`, displays the dialog by calling its `show()` method, and reads the answer.

We used an application rather than an applet to demonstrate the `Dialog` because dialogs are somewhat unweildy in applets. You need to have a `Frame` to serve as the dialog's parent, and most applets don't need Frames. However, there's a simple workaround. There's no reason an applet can't use an invisible frame: just create a `Frame`, call its `pack()` method, but never call its `show()` method. The `Frame` won't be displayed, but will be able to serve as the parent to a dialog box.

Now let's talk briefly about nonmodal dialogs. The most obvious change is in the constructor: now you call

```
new Dialog(myFrame, false);
```

But there are a few other issues to think about. Using a nonmodal dialog is slightly more complex because it's asynchronous: the program doesn't wait until the user responds. Therefore, you might want to modify the `answer()` method so that it calls `wait()` to wait until the user replies. The code would look like this:

```
// add a new boolean for the answer() method
private boolean isAnswered = false;
// add a wait() in the answer() method
synchronized public boolean answer() {
    while ( !isAnswered )
        try { wait(); } catch (InterruptedException e) { /* error */ }
    return isYes;
}
```

If you do this, you also need to modify `actionPerformed()` to call `notifyAll()` and terminate the `wait()`:

```
    // add a notify() in the actionPeformed() method
    synchronized public void actionPerformed ( ActionEvent e ) {
        isYes = e.getActionCommand().equals("Yes");
        isAnswered = true;
        notifyAll();
        dispose();
    }
}
```

File Selection Dialog

A `FileDialog` is a standard file-selection box. As with other AWT components, most of `FileDialog` is implemented in the native part of the AWT toolkit, so it looks and acts like a standard file selector on your platform.

Now selecting files all day can be pretty boring without a greater purpose, so we'll exercise the `FileDialog` in a mini-editor application. `Editor` provides a text area in which we can load and work with files. We'll stop just shy of the capability to save and let you fill in the blanks (with a few caveats). The `FileDialog` created by `Editor` is shown in Figure 11.6.

Figure 11.9: A FileDialog

[Graphic: Figure 11-9]

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
class Editor extends Frame implements ActionListener {
    TextArea textArea = new TextArea();
    Editor() {
        super("Editor");
        setLayout( new BorderLayout() );
        add("Center", textArea);
        Menu menu = new Menu ("File");
        menu.add ( makeMenuItem ("Load") );
        menu.add ( makeMenuItem ("Save") );
        menu.add ( makeMenuItem ("Quit") );
        MenuBar menuBar = new MenuBar();
        menuBar.add ( menu );
        setMenuBar( menuBar );
        pack();
    }
    public void actionPerformed( ActionEvent e ) {
        String command = e.getActionCommand();
        if ( command.equals("Quit") )
            dispose();
        else if ( command.equals("Load") )
            loadFile();
        else if ( command.equals("Save") )
            saveFile();
    }
    private void loadFile () {
        FileDialog fd = new FileDialog( this, "Load File", FileDialog.LOAD );
        fd.show();
        String file = fd.getFile();
        if ( file == null ) // Cancel
            return;
        try {
            FileInputStream fis = new FileInputStream ( fd.getFile() );
            byte [] data = new byte [ fis.available() ];
            fis.read( data );
            textArea.setText( new String( data ) );
        } catch ( IOException e ) {
            textArea.setText( "Could not load file..." );
        }
    }
    private void saveFile() {
        FileDialog fd = new FileDialog( this, "Save File", FileDialog.SAVE );
        fd.show();
        // Save file data...
    }
    private MenuItem makeMenuItem( String name ) {
        MenuItem m = new MenuItem( name );
        m.addActionListener( this );
        return m;
    }
    public static void main(String[] s) {
        new Editor().show();
    }
}
```

`Editor` is a `Frame` that lays itself out with a `TextArea` and a pull-down menu. From the pull-down **File** menu, we can opt to **Load**, **Save**, or **Quit**. The `action()` method catches the events associated with these menu selections and takes the appropriate action.

The interesting parts of `Editor` are the `private` methods `loadFile()` and `saveFile()`. `loadFile()` creates a new `FileDialog` with three parameters: a parent frame (just as in the previous `Dialog` example), a title, and a directive. This parameter should be one of the `FileDialog` class's `static` identifiers `LOAD` or `SAVE`, which tell the dialog whether to load or save a file.

A `FileDialog` does its work when the `show()` method is called. Unlike most components, its `show()` method blocks the caller until it completes its job; the file selector then disappears. After that, we can retrieve the designated filename with the `FileDialog`'s `getFile()` method. In `loadFile()`, we use a fragment of code from Chapter 8, *Input/Output Facilities* to get the contents of the named file. We then add the contents to the `TextArea` with `setText()`. You can use `loadFile()` as a roadmap for the unfinished `saveFile()`

method, but it would be prudent to add the standard safety precautions. For example, you could use the previous `YesNo` example to prompt the user before overwriting an existing file.

PREVIOUS HOME NEXT
ScrollPane and Scrollbars BOOK INDEX Creating custom components

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

11.9 Creating custom components

In the previous sections, we've worked with many different user interface objects, and made a lot of new classes that are sort of like components. Our new classes do one particular thing well; a number of them can be added to applets or other containers just like the standard AWT components; and several of them are lightweight components that use system resources efficiently because they don't rely on a peer.

But these new classes still aren't really components. If you think about it, all our classes have been fairly self-contained; they know everything about what to do, and don't rely on other parts of the program to do much processing. Therefore, they are overly specialized. Our menu example created a `DinnerFrame` class that had a menu of dinner options, but it included all the processing needed to handle the user's selections. If we wanted to process the selections differently, we'd have to modify the class. That's not what we want; we'd like to separate registering the user's choices from processing those choices. In contrast, true components don't do any processing. They let the user take some action, and then inform some other part of the program, which processes the action.

So we need a way for our new classes to communicate with other parts of the program. Since we want our new classes to be components, they should communicate the way components communicate: that is, by generating events and sending those events to listeners. So far, we've written a lot of code that listened for events, but haven't seen any examples that generated events.

Generating events sounds like it ought to be difficult, but it isn't. You can either create new kinds of events, by subclassing `AWTEvent`, or use one of the standard event types. In either case, you need to register listeners for your events, and provide a means to deliver events to your listeners. If you are using the standard events, AWT provides an `AWTEventMulticaster` class that handles most of the machinery. We'll focus on that option in this section; at the end, we'll make some comments on how you might manage events on your own.

The `AWTEventMulticaster` is one of those things that looks a lot more complicated than it is. It is confusing, but most of the confusion occurs because it's hard to believe it's so simple. Its job is to maintain a linked list of event listeners and to propagate events to all the listeners on that linked list. So we can use a multicaster to register (and unregister) event listeners and to send any events we generate to all registered listeners.

The best way to show you how to use the multicaster is through an example. The following example creates a new component called `PictureButton`. `PictureButton` looks at least somewhat button-like and responds to mouse clicks (`MOUSE_RELEASED` events) by generating action events. (Figure 11.7 shows a `PictureButton` in both depressed and raised modes.) The `PictureButtonApplet` is passed the events in its `actionPerformed()` method, just as with any other button, and prints a message each time it's pressed.

Figure 11.10: The PictureButtonApplet

[Graphic: Figure 11-10]

```
import java.awt.*;
import java.awt.event.*;
public class PictureButtonApplet extends java.applet.Applet implements ActionListener
{
    Image image;
    public void init() {
        image = getImage( getClass().getResource(getParameter("image")) );
        PictureButton pictureButton = new PictureButton( image );
        add ( pictureButton );
        pictureButton.setActionCommand("Aaargh!");
        pictureButton.addActionListener( this );
    }

    public void actionPerformed( ActionEvent e ) {
        System.out.println( e );
    }
}
class PictureButton extends Component {
    private Image image;
    boolean pressed = false;
    ActionListener actionListener;
    String actionCommand;
    PictureButton(Image image) {
        this.image = image;
        MediaTracker mt = new MediaTracker(this);
        mt.addImage( image, 0 );
        try { mt.waitForAll(); } catch (InterruptedException e) { /* error */ };
        setSize( image.getWidth(null), image.getHeight(null) );
        enableEvents( AWTEvent.MOUSE_EVENT_MASK );
    }
    public void paint( Graphics g ) {
        g.setColor(Color.white);
        int width = getSize().width, height = getSize().height;
        int offset = pressed ? -2 : 0;  // fake depth
        g.drawImage( image, offset, offset, width, height, this );
        g.draw3DRect(0, 0, width-1, height-1, !pressed);
        g.draw3DRect(1, 1, width-3, height-3, !pressed);
    }
    public Dimension getPreferredSize() {
        return getSize();
    }
    public void processEvent( AWTEvent e ) {
        if ( e.getID() == MouseEvent.MOUSE_PRESSED ) {
            pressed = true;
            repaint();
        } else
        if ( e.getID() == MouseEvent.MOUSE_RELEASED ) {
            pressed = false;
            repaint();
            fireEvent();
        }
        super.processEvent(e);
    }
    public void setActionCommand( String actionCommand ) {
        this.actionCommand = actionCommand;
    }
    public void addActionListener(ActionListener l) {
        actionListener = AWTEventMulticaster.add(actionListener, l);
    }
    public void removeActionListener(ActionListener l) {
        actionListener = AWTEventMulticaster.remove(actionListener, l);
    }
    private void fireEvent() {
        if (actionListener != null) {
            ActionEvent event = new ActionEvent( this,
                                ActionEvent.ACTION_PERFORMED, actionCommand );
            actionListener.actionPerformed( event );
        }
    }
}
```

Before diving into the event multicaster, here are a few notes about the applet and the `PictureButton`. The applet is an `ActionListener` because it is looking for events coming from the button. Therefore it registers itself as a listener and contains an `actionPerformed()` method. The `PictureButton` doesn't have a label, so the applet explicitly sets the button's action command by calling `setActionCommand()`.

The button itself is concerned mostly with being pretty. It uses a media tracker to make sure that the image has loaded before displaying itself. The `paint()` method, which we won't discuss in detail, is devoted to making the button appear "pressed" (i.e., recessed) when the mouse is pressed. The `getPreferredSize()` method lets layout managers size the button appropriately.

Now we'll start with the button's machinery. The button needs to receive mouse events. It could register as a mouse listener, but in this case, it seems more appropriate to override `processEvent()`. `processEvent()` receives all incoming events. It first checks whether we have a `MOUSE_PRESSED` event; if so, it tells the button to repaint itself in its "pressed" mode. If the event is a `MOUSE_RELEASED` event, it tells the button to paint itself in its "unpressed" mode and calls the private `fireEvent()` method, which sends an action event to all listeners. Finally, `processEvent()` calls `super.processEvent()` to make sure normal event processing occurs; this is a good practice whenever you override a method that performs a significant task.

However, `processEvent()` doesn't receive events if they aren't generated; and normally, events aren't generated if there are no listeners. Therefore, the button's constructor calls `enableEvents()` with the argument `MOUSE_EVENT_MASK` to turn on mouse event processing.

Now we're ready to talk about how to generate events. The picture button has `addActionListener()` and `removeActionListener()` methods, for registering listeners. These just call the static methods `add()` and `remove()` of the `AWTEventMulticaster` class. Here's the `addActionListener()` method:

```
public void addActionListener(ActionListener l) {
    actionListener = AWTEventMulticaster.add(actionListener, l);
}
```

If you look back to see how the instance variable `actionListener` is declared, you'll see that it is an `ActionListener`. No big surprise--except that this code doesn't appear to make sense. It's saying "add an action listener to an action listener and store the result back in the original action listener."

There are a couple of tricks here. First, an `AWTEventMulticaster` implements all of the listener interfaces. Therefore, a multicaster can appear wherever an `ActionListener` (or any other listener) is required. In this case, the `actionListener` object will be a multicaster--perhaps not what you expected, and certainly not what's being passed in the argument `l`. Now the code is starting to make sense: earlier, I said that multicasters maintained linked lists of listeners. So this method really adds `l` to the linked list of action listeners that a multicaster is managing, and saved the new list.

But that begs the question: where does the multicaster come from? The linked list has to start somewhere. This is where the second trick comes in. `add()` is a static method, so we don't need a multicaster to call it. But we still need some way to start the linked list. The class's constructor is never called--in fact, it's protected, so you can't call it. The answer lies in the `add()` method, which creates an `AWTEventMulticaster` when you need it--that is, as soon as you add the second listener to the list. The arguments to `add()` may be `null`; one of them probably is `null` when you register your first action listener.

Removing action listeners works the same way. We use the `AWTEventMulticaster`'s `remove()` method. After the last listener is taken off the linked list, `remove()` returns `null`.

With this machinery in place, sending an event to all registered listeners is very simple. You just create an event by calling its constructor, and then call the appropriate method in the listener interface to deliver it. The `AWTEventMulticaster` makes sure that the event gets to all the listeners. In this example, we create an `ActionEvent` and deliver the event to the listeners' `actionPerformed()` methods by calling `actionListener.actionPerformed(event)`.

The code to generate other kinds of events is almost exactly the same. Remember the multicaster implements all the listener interfaces and has overloaded `add()` and `remove()` methods for every standard listener type. Therefore, it can be used for any kind of `AWTEvent`. It shouldn't be hard to adapt this example to other situations.

What if you want to generate your own event type by subclassing `AWTEvent`? To make things concrete, let's say you want to create an `ExplosionEvent` that you generate whenever your monitor catches fire. In this case, you should define your own `ExplosionListener` interface, and (possibly) your own `ExplosionAdapter` class. You can't use the `AWTEventMulticaster` unless your new event is a subclass of a standard event; extending the multicaster to support new event types probably isn't worth the effort. It's easier to write an `addExplosionListener()` method that maintains a `Vector` of listeners and to deliver events by calling the appropriate method of each listener in the `Vector`. We'll demonstrate this approach in the next section, where we implement another new component: a `Dial`.

A Dial Component

```
    Things to mention in widgets Dial event example:
        synchronization issues in add/remove/fire.
        You should sync add/remove... but be wary of syncing fire,
        deadlocks
```

The standard AWT classes don't have a component that's similar to an old fashioned dial--for example, the volume control on your radio. Such a component is something of a rarity; I don't remember seeing one in any application I've used. But there's no reason we can't build one. In this section, we implement a `Dial` class. We also define a new event type, `DialEvent`, and a new listener interface, `DialListener`. The dial can be used just like any other Java component. It is built entirely in Java and, therefore, is a lightweight component; it extends `Component` directly and doesn't have a peer.

By defining a new event type, I'm stretching the point slightly. There's no reason our dial couldn't use the standard `AdjustmentEvent`. However, this gives us a chance to show how to handle event listeners without using the event multicaster. There are many situations in which defining a new event type will be the only appropriate solution.

Figure 11.11 shows what the dial looks like; it is followed by the code.

Figure 11.11: The Dial Applet

[Graphic: Figure 11-11]

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
public interface DialListener {
    void dialAdjusted( DialEvent e );
}
public class DialEvent extends AWTEvent {
    int value;
    public static final int DIAL_ADJUSTED = RESERVED_ID_MAX + 1;
    DialEvent( Dial source, int id, int value ) {
        super( source, id );
        this.value = value;
    }
    public int getValue() {
        return value;
    }
}
public class Dial extends Component {
    int minValue = 0, value, maxValue = 100, radius;
    Vector dialListeners;
    Dial( int maxValue ) {
        this.maxValue = maxValue;
        enableEvents( AWTEvent.MOUSE_MOTION_EVENT_MASK );
    }
    public void paint( Graphics g ) {
        int tick = 10;
        radius = getSize().width/2 - tick;
        g.drawLine(radius*2+tick/2, radius, radius*2+tick, radius); // the tick
        draw3DCircle( g, 0, 0, radius, true );
        draw3DCircle( g, 1, 1, radius-1, true );
        int knobRadius = radius/7;
        double th = value*(2*Math.PI)/(maxValue-minValue);
        int x = (int)(Math.cos(th)*(radius-knobRadius*3)),
            y = (int)(Math.sin(th)*(radius-knobRadius*3));
        draw3DCircle( g, x+radius-knobRadius, y+radius-knobRadius, knobRadius, false
);
    }
    private void draw3DCircle( Graphics g, int x, int y, int radius, boolean raised )
{
        g.setColor( raised ? Color.white : Color.black );
        g.drawArc( x, y, radius*2, radius*2, 45, 180);
        g.setColor( raised ? Color.black : Color.white);
        g.drawArc( x, y, radius*2, radius*2, 225, 180);
    }
    public void processEvent( AWTEvent e ) {
        if ( e.getID() == MouseEvent.MOUSE_DRAGGED ) {
            int y=((MouseEvent)e).getY();
            int x=((MouseEvent)e).getX();
            double th = Math.atan( (1.0*y-radius)/(x-radius) );
            int value = ( (int)(th/(2*Math.PI) * (maxValue-minValue)) );
            if ( x < radius )
                setValue( value + maxValue/2 );
            else if ( y < radius )
                setValue( value + maxValue );
            else
                setValue( value );
            fireEvent();
        }
        super.processEvent( e );
    }
    public void setValue(int value) {
        this.value = value;
        repaint();
    }
    public int getValue()  { return value; }
    public void setMinimum(int minValue )  { this.minValue = this.minValue; }
    public int getMinimum()  { return minValue; }
    public void setMaximum(int maxValue )  { this.maxValue = maxValue; }
    public int getMaximum()  { return maxValue; }
    public void addDialListener(DialListener listener) {
        if ( dialListeners == null )
            dialListeners = new Vector();
        dialListeners.addElement( listener );
    }
    public void removeDialListener(DialListener listener) {
        if ( dialListeners != null )
            dialListeners.removeElement( listener );
    }
    private void fireEvent() {
        if ( dialListeners == null )
            return;
        DialEvent event = new DialEvent(this, DialEvent.DIAL_ADJUSTED, value);
        for (Enumeration e = dialListeners.elements(); e.hasMoreElements(); )
            ((DialListener)e.nextElement()).dialAdjusted( event );
    }
}
public class DialApplet extends java.applet.Applet
                                implements DialListener, AdjustmentListener {
    final int max = 100;
    Scrollbar scrollbar = new Scrollbar( Scrollbar.HORIZONTAL, 0, 1, 0, max );
    Dial dial = new Dial( max );
    public void init() {
        setLayout( new BorderLayout() );
        dial.addDialListener( this );
        add( "Center", dial );
        scrollbar.addAdjustmentListener( this );
        add( "South", scrollbar );
    }
    public void dialAdjusted( DialEvent e ) {
        scrollbar.setValue( e.getValue() );
    }
    public void adjustmentValueChanged( AdjustmentEvent e ) {
        dial.setValue( e.getValue() );
    }
}
```

Let's start from the top. We'll focus on the event handling and leave you to figure out the trigonometry on your own. The DialListener interface contains a single method, dialAdjusted(), which is called when a DialEvent occurs. The DialEvent itself is simple. It defines a new event ID, DIAL_ADJUSTED, that identifies dial events. This constant is defined so that it won't conflict with the ID numbers reserved for standard AWT events. The event itself only carries one item of data: the dial's new value. It has a single method that returns this value.

The constructor for the Dial class stores the dial's maximum value; its minimum value is 0. It then enables mouse motion events, which the Dial needs to tell how it is being manipulated.

paint(), draw3DCircle(), and processEvent() do a lot of trigonometry to figure out how to display the dial. draw3DCircle() is a private helper method that draws a circle that appears either raised or depressed; we use this to make the dial look three dimensional. processEvent() is called whenever any event occurs within the component's area. We only expect to receive mouse motion events, because these are the only events we enabled. processEvent() first checks the event's ID; if it is MOUSE_DRAGGED, the user has changed the dial's setting. We respond by computing a new value for the dial, repainting the dial in its new position and firing a DialEvent. Any other events (in particular, MOUSE_MOVED) are ignored. However, we call the superclass's processEvent() method to make sure that any other processing needed for this event occurs.

The next group of methods provide ways to retrieve or change the dial's current setting, minimum, and maximum values. They are similar to the methods in the Adjustable interface; you could argue that Dial really ought to implement Adjustable.

Finally, we reach the methods that work with listeners. addDialListener() adds a new listener to a Vector of listeners by calling addElement(). If the vector doesn't already exist, addDialListener() creates it. removeDialListener() simply takes a listener off the list, so that it won't receive any further events. fireEvent() is a private method that creates a DialEvent and sends it to every listener. It does so by converting the Vector into an Enumeration and running through every element in the list by calling nextElement() until hasMoreElements()

returns false. To send the event to a listener, it calls the listener's `dialAdjusted()` method. Note that `nextElement()` returns an `Object`; we must cast this object to `DialListener` before we can deliver the event.

To show how the applet is used, I included a simple applet called `DialApplet`. This applet places a `Dial` and a `Scrollbar` in a border layout. Any change to either the dial or the scrollbar is reflected by the other. The applet implements both `DialListener` and `AdjustmentListener`, and therefore has both `dialAdjusted()` and `adjustmentValueChanged()` methods. Although this isn't necessarily a good argument for creating a new event type, it's worth noticing that the logic of the listener methods is much simpler than it would have been if the dial generated adjustment events.

12.2 GridLayout

`GridLayout` arranges components into regularly spaced rows and columns. The components are arbitrarily resized to fit in the resulting areas; their minimum and preferred sizes are consequently ignored. `GridLayout` is most useful for arranging very regular, identically sized objects and for allocating space for Panels to hold other layouts in each region of the container.

`GridLayout` takes the number of rows and columns in its constructor. If you subsequently give it too many objects to manage, it adds extra columns to make the objects fit. You can also set the number of rows or columns to zero, which means that you don't care how many elements the layout manager packs in that dimension. For example, `GridLayout(2,0)` requests a layout with two rows and an unlimited number of columns; if you put ten components into this layout, you'll get two rows of five columns each.

The following applet sets a `GridLayout` with three rows and two columns as its layout manager; the results are shown in Figure 12.3.

Figure 12.3: A grid layout

[Graphic: Figure 12-3]

```
import java.awt.*;

public class Grid extends java.applet.Applet {
    public void init() {
        setLayout( new GridLayout( 3, 2 ));
        add( new Button("One") );
        add( new Button("Two") );
        add( new Button("Three") );
        add( new Button("Four") );
        add( new Button("Five") );
    }
}
```

The five buttons are laid out, in order, from left to right, top to bottom, with one empty spot.

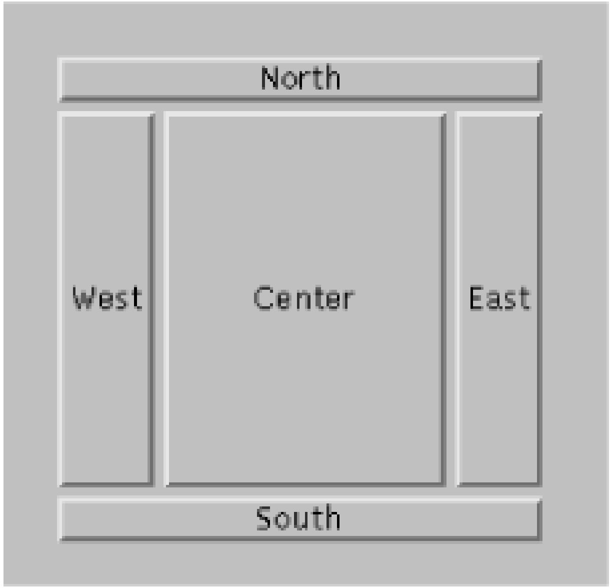
12.3 BorderLayout

`BorderLayout` is a little more interesting. It tries to arrange objects in one of five geographical locations: "North," "South," "East," "West," and "Center," possibly with some padding between. `BorderLayout` is the default layout for `Window` and `Frame` objects. Because each component is associated with a direction, `BorderLayout` can manage at most five components; it squashes or stretches those components to fit its constraints. As we'll see in the second example, this means that you often want to have `BorderLayout` manage sets of components in their own panels.

When we add a component to a border layout, we need to specify both the component and the position at which to add it. To do so, we use an overloaded version of the `add()` method that takes an additional argument as a constraint. This additional argument is passed to the layout manager when the new component is added. In this case it specifies the name of the position for the BorderLayout. Otherwise the `LayoutManager` is not consulted until it's asked to lay out the components.

The following applet sets a `BorderLayout` layout and adds our five buttons again, named for their locations; the result is shown in Figure 12.4.

Figure 12.4: A border layout



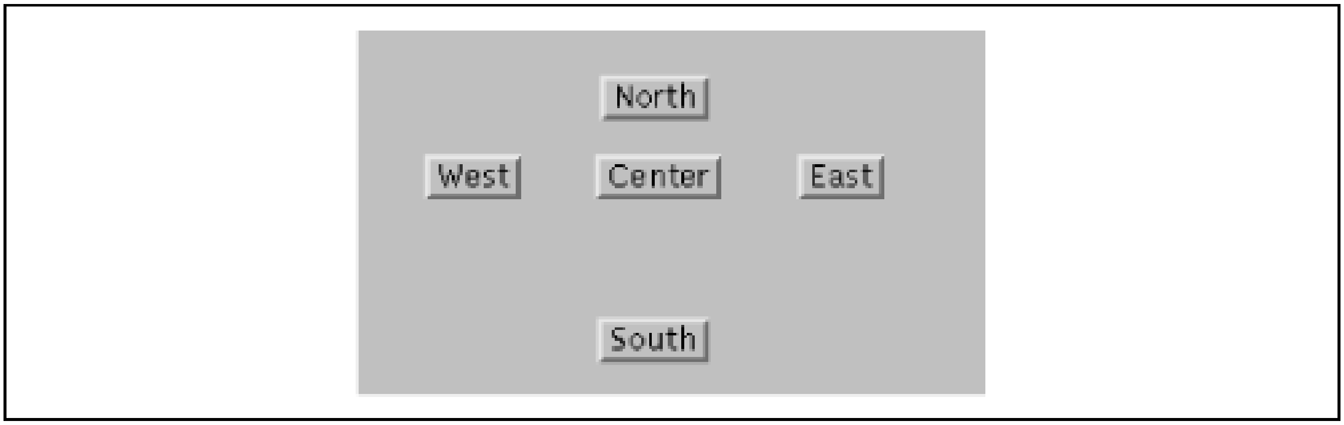
```
import java.awt.*;

public class Border extends java.applet.Applet {
    public void init() {
        setLayout( new java.awt.BorderLayout() );
        add( new Button("North"), "North" );
        add( new Button("East"), "East" );
        add( new Button("South"), "South" );
        add( new Button("West"), "West" );
        add( new Button("Center"), "Center" );
    }
}
```

So, how exactly is the area divided up? Well, the objects at "North" and "South" get their preferred height and are expanded to fill the full area horizontally. "East" and "West" components on the other hand, get their preferred width, and are expanded to fill the remaining area between "North" and "South" vertically. Finally, the "Center" object takes all of the rest of the space. As you can see in Figure 12.5, our buttons get distorted into interesting shapes.

What if we don't want `BorderLayout` messing with the sizes of our components? One option would be to put each button in its own `Panel`. The default layout for a `Panel` is `FlowLayout`, which respects the preferred size of components. The preferred sizes of the panels are effectively the preferred sizes of the buttons, but if the panels are stretched, they won't pull their buttons with them. `Border2` illustrates this approach as shown in Figure 12.5.

Figure 12.5: Another border layout



```
import java.awt.*;

public class Border2 extends java.applet.Applet {
    public void init() {
        setLayout( new BorderLayout() );
        Panel p = new Panel();
        p.add(new Button("East") );
        add( p, "East" );
        p = new Panel();
        p.add(new Button("West") );
        add( p, "West" ;
        p = new Panel();
        p.add(new Button("North") );
        add( p, "North" );
        p = new Panel();
        p.add(new Button("South") );
        add(p, "South" );
        p = new Panel();
        p.add(new Button("Center") );
        add( p, "Center" );
    }
}
```

In this example, we create a number of panels, put our buttons inside the panels, and put the panels into the applet, which has the `BorderLayout` manager. Now, the `Panel` for the "Center" button soaks up the extra space that comes from the `BorderLayout`. Each `Panel`'s `FlowLayout` centers the button in the panel and uses the button's preferred size. In this case, it's all a bit awkward. (This is one of the problems that `getMaximumSize()` will eventually solve.) We'll see how we could accomplish this more directly using `GridBagLayout` shortly.

Finally, this version of the applet has a lot of unused space. If we wanted, we could get rid of the extra space by resizing the applet:

```
setSize( getPreferredSize() );
```

PREVIOUS
GridLayout

HOME
BOOK INDEX

NEXT
CardLayout

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

12.4 CardLayout

`CardLayout` is a special layout manager for creating the effect of a stack of cards. Instead of arranging all of the container's components, it displays only one at a time. You would use this kind of layout to implement a hypercard stack or a Windows-style set of configuration screens. When you add a component to the layout, you use the two-argument version of `add()`; the extra argument is an arbitrary string that serves as the card's name:

```
add("netconfigscreen", myComponent);
```

To bring a particular card to the top of the stack, call the `CardLayout`'s `show()` method with two arguments: the parent `Container` and the name of the card you want to show. There are also methods like `first()`, `last()`, `next()`, and `previous()` for working with the stack of cards. These methods take a single argument: the parent `Container`. Here's a simple example:

```
import java.awt.*;

public class main extends java.applet.Applet {
    CardLayout cards = new CardLayout();

    public void init() {
        setLayout( cards );
        add( new Button("one"), "one" );
        add( new Button("two"), "two" );
        add( new Button("three"), "three" );
    }

    public boolean action( Event e, Object arg) {
        cards.next( this );
        return true;
    }
}
```

We add three buttons to the layout and cycle through them as they are pressed. In a more realistic example, we would build a group of panels, each of which might implement some part of a complex user interface, and add those panels to the layout. Each panel would have its own layout manager. The panels would be resized to fill the entire area available (i.e., the area of the `Container` they are in), and their individual layout managers would arrange their internal components.

PREVIOUS HOME NEXT
BorderLayout BOOK INDEX GridBagLayout

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

12.7 Absolute Positioning?

It's possible to set the layout manager to `null`: no layout control. You might do this to position an object on the display at some absolute coordinates. This is almost never the right approach. Components might have different minimum sizes on different platforms, and your interface would not be very portable.

The following applet doesn't use a layout manager and works with absolute coordinates instead:

```
import java.awt.*;

public class MoveButton extends java.applet.Applet {
    Button button = new Button("I Move");

    public void init() {
        setLayout( null );
        add( button );
        button.setSize( button.getPreferredSize() );
        button.move( 20, 20);
    }

    public boolean mouseDown( Event e, int x, int y ) {
        button.move( x, y );
        return ( true );
    }

}
```

Click in the applet area, outside of the button, to move the button to a new location. If you are running the example in an external viewer, try resizing the window and note that the button stays at a fixed position relative to the display origin.

PREVIOUS
Nonstandard Layout Managers

HOME
BOOK INDEX

NEXT
Drawing With the AWT

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

13.2 Colors

The `TestPattern` applet fills its shapes with a number of colors, using the `setColor()` method of the `Graphics` object. `setColor()` sets the current color in the graphics context, so we set it to a different color before each drawing operation. But where do these color values come from?

The `java.awt.Color` class handles color in Java. A `Color` object describes a single color. You can create an arbitrary `Color` by specifying the red, green, and blue values, either as integers between 0 and 255 or as floating-point values between 0.0 and 1.0. You can also use `getColor()` to look up a named color in the system properties table, as described in Chapter 7, *Basic Utility Classes*. `getColor()` takes a `String` color property name, retrieves the integer value from the `Properties` list, and returns the `Color` object that corresponds to that color.

The `Color` class also defines a number of `static final` color values; these are what we used in the `TestPattern` example. These constants, such as `Color.black` and `Color.red`, provide a convenient set of basic colors for your drawings.

Desktop Colors

The `Color` class I just described makes it easy to construct a particular color; however, that's not always what you want to do. Sometimes you want to match a preexisting color scheme. This is particularly important when you are designing a user interface; you might want your components to have the same colors as other components on that platform, and to change automatically if the user redefines his or her color scheme.

That's what the `SystemColor` class is for. A system color represents the color used by the local windowing system in a certain context. The SystemColor class holds lots of pre-defined SystemColors, just like the Color Class holds some pre-defined basic colors. For example, the field `activeCaption` represents the color used for the background to the title of an active window; `activeCaptionText` represents the color used for the title itself. `menu` represents the background color of menu selection; `menuText` represents the color of a menu item's text when it is not selected; `textHighlightText` is the color used when the item is selected; and so on. You could use the `window` value to set the color of a `Window` to match the other Windows on the user's screen--whether or not they're generated by Java programs.

```
myWindow.setBackground( SystemColor.window );
```

Because the `SystemColor` class is a subclass of `Color`, you can use it wherever you would use a `Color`. However, the `SystemColor` constants are tricky. They are constants as far as you, the programmer, are concerned; your code is not allowed to modify them. However, they can be modified at run-time by the Toolkit. If the user changes his color scheme, the system colors are automatically updated to follow suit; as a result, anything displayed with system colors will also change color the next time it is redrawn. For example, the window `myWindow` would automatically change its background color to the new background color.

The `SystemColor` class has one noticeable shortcoming. You can't compare a system color to a `Color` directly; the `Color.equals()` method doesn't return reliable results. For example, if you want to find out whether the window background color is red, you can't call:

```
Color.red.equals(SystemColor.window);
```

Instead, you should use `getRGB()` to find the color components of both objects, and compare them, rather than comparing the objects themselves.

13.3 Fonts

Text fonts in Java are represented by instances of the `java.awt.Font` class. A `Font` object is constructed from a font name, style identifier, and a point size. We can create a `Font` at any time, but it's meaningful only when applied to a particular component on a given display device. Here are a couple of fonts:

```
Font smallFont = new Font("Monospaced", Font.PLAIN, 10);
Font bigFont = new Font("Serif", Font.BOLD, 18);
```

The font name is a symbolic name for the font family. The following font names should be available on all platforms; Figure 13.4 shows what these fonts look like on a typical platform:[2]

> [2] The names `Helvetica`, `TimesRoman`, `Courier`, `Symbol`, and `ZapfDingbats` are supported in Java 1.1 for backwards compatibility, but shouldn't be used; they may be removed in a future version. `Symbols` and `ZapfDingbats`, which used to be available as `Font` names have now taken their proper place as ranges in the Unicode character table: 2200-22ff and 2700-27ff respectively.

Figure 13.4: Font examples

[Graphic: Figure 13-4]

- Serif (generic name for TimesRoman)
- SansSerif (generic name for Helvetica)
- Monospaced (generic name for Courier)
- Dialog

- DialogInput

The font you specify is mapped to an actual font on the local platform. Java's *fonts.properties* files map the font names to the available fonts, covering as much of the Unicode character set as possible. If you request a font that doesn't exist, you get the default font.

You can also use the `static` method `Font.getFont()` to look up a font name in the system properties list. `getFont()` takes a `String` font property name, retrieves the font name from the `Properties` table, and returns the `Font` object that corresponds to that font. You can use this mechanism, as with Colors, to define fonts with properties from outside your application.

The `Font` class defines three `static` style identifiers: `PLAIN`, `BOLD`, and `ITALIC`. You can use these values on all fonts. The point size determines the size of the font on a display. If a given point size isn't available, `Font` substitutes a default size.[3]

> [3] There is no straightforward way to determine if a given `Font` is available at a given point size in the current release of Java. Fonts are one of Java's weak points. Sun has promised better `Font` handling (and perhaps true, portable `Fonts`) in a future release.

You can retrieve information about an existing `Font` with a number of routines. The `getName()`, `getSize()` and `getStyle()` methods retrieve the symbolic name, point size and style, respectively. You can use the getFamily() method to find out the platform specific font family to which the font actually maps.

Finally, to actually use a `Font` object you can simply specify it as an argument to the `setFont()` method of a `Component` or `Graphics` object. Subsequent text-drawing commands like `drawString()` for that component or in that graphics context use the specified font.

Font Metrics

To get detailed size and spacing information for text rendered in a font, we can ask for a `java.awt.FontMetrics` object. Different systems will have different real fonts available; the available fonts may not match the font you request. Thus, a `FontMetrics` object presents information about a particular font on a particular system, not general information about a font. For example, if you ask for the metrics of a nine-point Monospaced font, what you get isn't some abstract truth about Monospaced fonts; you get the metrics of the font that the particular system uses for nine-point Monospaced--which may not be exactly nine point or even Monospaced.

Use the `getFontMetrics()` method for a `Component` to retrieve the `FontMetrics` for a `Font` as it would appear for that component:

```
public void init() {
    ...
    // Get the metrics for a particular font on this component
    FontMetrics smallFont = myLabel.getFontMetrics( smallFont );
    ...
}
```

The `Graphics` object also has a `getFontMetrics()` method that gets the `FontMetrics` information for the current font in the graphics context.

```
public void paint( Graphics g ) {
    // Get the metrics for the current font
    FontMetrics fm = g.getFontMetrics();
   ...
}
```

The following applet, `FontShow`, displays a word and draws reference lines showing certain characteristics of its font, as shown in Figure 13.5. Clicking in the applet toggles the point size between a small and a large value.

Figure 13.5: The FontShow applet

[Graphic: Figure 13-5]

```
import java.awt.*;
import java.awt.event.*;
public class FontShow extends java.applet.Applet {
    static final int LPAD=25;   // Frilly line padding
    boolean bigFont = true;
    public void init() {
        addMouseListener( new MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
                bigFont = !bigFont;
                repaint();
            }
        } );
    }
    public void paint( Graphics g ) {
        String message = getParameter( "word" );
        g.drawRect(0, 0, getSize().width-1, getSize().height-1);
        if ( bigFont )
            g.setFont( new Font("Dialog",Font.PLAIN,24) );
        else
            g.setFont( new Font("Dialog",Font.PLAIN,12) );
        FontMetrics metrics = g.getFontMetrics();
        int fontAscent = metrics.getMaxAscent ();
        int fontDescent = metrics.getMaxDescent();
        int messWidth = metrics.stringWidth ( message );
        // Center text
        int startX = getSize().width/2 - messWidth/2;
        int startY = getSize().height/2 - fontDescent/2 + fontAscent/2;
        g.drawString(message, startX, startY);
        g.setColor( Color.white );  // Base lines
        g.drawLine( startX-LPAD, startY, startX+messWidth+LPAD, startY );
        g.drawLine( startX, startY+ LPAD, startX, startY-fontAscent-LPAD );
        g.setColor( Color.green );  // Ascent line
        g.drawLine( startX-LPAD, startY-fontAscent, startX+messWidth+LPAD, startY-
fontAscent );
        g.setColor( Color.red );    // Descent line
        g.drawLine( startX-LPAD, startY+fontDescent, startX+messWidth+LPAD,
startY+fontDescent );
    }
}
```

Compile `FontShow` and run it with an applet tag like the following:

```
<applet height=200 width=250 code=FontShow>
    <param name="word" value="Lemming">
</applet>
```

The `word` parameter specifies the text to be displayed.

`FontShow` may look a bit complicated, but there's really not much to it. The bulk of the code is in `paint()`, which simply sets the font, draws our word, and adds a few lines to illustrate some of the font's characteristics (metrics). For fun we also catch mouse clicks (in the `mouseClicked()` method) and alternate the font size by setting the `bigFont` variable and repainting.

By default, text is rendered above and to the right of the coordinates specified in the `drawString()` method. If you think of that starting point as the origin of a coordinate system, we'll call the axes the "baselines" of the font. `FontShow` draws these lines in white. The greatest height the characters stretch above the baseline is called the *ascent* and is shown by a green line. Some fonts also have parts of letters that fall below the baseline. The farthest distance any character reaches below the baseline is called the *descent*. `FontShow` illustrates this with a red line.

We ask for the ascent and descent of our font with the `FontMetrics` `getMaxAscent()` and `getMaxDescent()` methods. We also ask for the width of our string (when rendered in this font) with the `stringWidth()` method. We use this information to center the word in the display area. To center the word vertically, we average the influence of the ascent and descent.

Table 13.2 provides a short list of methods that return useful font metrics.

Table 13.2: Font Metric Methods

Method	**Description**
`getFont()`	`Font` object these metrics describe
`getAscent()`	Height above baseline

`getDescent()`	Depth below baseline
`getLeading()`	Standard vertical spacing between lines
`getHeight()`	Total line height (ascent + descent + leading)
`charWidth(char ch)`	Width of a character
`stringWidth(String str)`	Width of a string
`getWidths()`	The widths of the first 256 characters in this font; returns `int[]`
`getMaxAdvance()`	Maximum character width of any character

Leading space is the padding between lines of text. The `getHeight()` method reports the total height of a line of text, including the leading space.

PREVIOUS | HOME | NEXT
Colors | BOOK INDEX | Images

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

13.5 Drawing Techniques

Having learned to walk, let's try a jog. In this section, we'll look at some techniques for doing fast and flicker-free drawing and painting. If you're interested in animation or smooth updating, you should read on.[4]

> [4] At this point, you still have to build your own animation software. JavaSoft will be releasing an animation package as part of the Java Media APIs.

Drawing operations take time, and time spent drawing leads to delays and imperfect results. Our goal is to minimize the amount of drawing work we do and, as much as possible, to do that work away from the eyes of the user. You'll remember that our `TestPattern` applet had a blinking problem. It blinked because `TestPattern` performs several, large, area-filling operations each time its `paint()` method is called. On a very slow system, you might even be able to see each shape being drawn in succession. `TestPattern` could be easily fixed by drawing into an off-screen buffer and then copying the completed buffer to the display. To see how to eliminate flicker and blinking problems, we'll look at an applet that needs even more help.

`TerribleFlicker` illustrates some of the problems of updating a display. Like many animations, it has two parts: a constant background and a changing object in the foreground. In this case, the background is a checkerboard pattern and the object is a small, scaled image we can drag around on top of it, as shown in Figure 13.6. Our first version of `TerribleFlicker` lives up to its name and does a very poor job of updating.

Figure 13.6: The TerribleFlicker applet

[Graphic: Figure 13-6]

```
import java.awt.*;
import java.awt.event.*;
public class TerribleFlicker extends java.applet.Applet
```

```
                                        implements MouseMotionListener {
    int grid = 10;
    int currentX, currentY;
    Image img;
    int imgWidth = 60, imgHeight = 60;
    public void init() {
        img = getImage( getClass().getResource(getParameter("img")) );
        addMouseMotionListener( this );
    }
    public void mouseDragged( MouseEvent e ) {
        currentX = e.getX(); currentY = e.getY();
        repaint();
    }
    public void mouseMoved( MouseEvent e ) { }; // complete MouseMotionListener
    public void paint( Graphics g ) {
        int w = getSize().width/grid;
        int h = getSize().height/grid;
        boolean black = false;
        for ( int y = 0; y <= grid; y++ )
            for ( int x = 0; x <= grid; x++ ) {
                g.setColor(  (black = !black) ? Color.black : Color.white );
                g.fillRect( x * w, y * h, w, h );
            }
        g.drawImage( img, currentX, currentY, imgWidth, imgHeight, this );
    }
}
```

Try dragging the image; you'll notice both the background and foreground flicker as they are repeatedly redrawn. What is `TerribleFlicker` doing, and what is it doing wrong?

As the mouse is dragged, `TerribleFlicker` keeps track of its position in two instance variables, `currentX` and `currentY`. On each call to `mouseDragged()`, the coordinates are updated, and `repaint()` is called to ask that the display be updated. When `paint()` is called, it looks at some parameters, draws the checkerboard pattern to fill the applet's area, and finally paints a small version of the image at the latest coordinates.

Our first, and biggest, problem is that we are updating, but we have neglected to implement the applet's `update()` method with a good strategy. Because we haven't overridden `update()`, we are getting the default implementation of the `Component` `update()` method, which looks something like this:

```
// Default implementation of applet update
public void update( Graphics g ) {
    setColor ( backgroundColor );
    fillRect( 0, 0, getSize().width, getSize().height );
    paint ( g );
}
```

This method simply clears the display to the background color and calls our `paint()` method. This is almost never the best strategy, but is the only appropriate default for `update()`, which doesn't know how much of the screen we're really going to paint.

Our applet paints its own background, in its entirety, so we can provide a simpler version of `update()` that doesn't bother to clear the display:

```
// add to TerribleFlicker
public void update( Graphics g ) {
    paint( g );
}
```

This applet works better because we have eliminated one large, unnecessary, and (in fact) annoying graphics operation. However, although we have eliminated a `fillRect()` call, we're still doing a lot of wasted drawing. Most of the background stays the same each time it's drawn. You might think of trying to make `paint()` smarter, so that it wouldn't redraw these areas, but remember that `paint()` has to be able to draw the entire scene because it might be called in situations when the display isn't intact. The solution is to have `update()` help out by restricting the area `paint()` can draw.

Clipping

The `setClip()` method of the `Graphics` class restricts the drawing area of a graphics context to a smaller region. A graphics context normally has an effective clipping region that limits drawing to the entire display area of the component. We can specify a smaller clipping region with `setClip()`.

How is the drawing area restricted? Well, foremost, drawing operations that fall outside of the clipping region are not displayed. If a drawing operation overlaps the clipping region, we see only the part that's inside. A second effect is that, in a good implementation, the graphics context can recognize drawing operations that fall completely outside the clipping region and ignore them altogether. Eliminating unnecessary operations can save time if we're doing something complex, like filling a bunch of polygons. This doesn't save the time our application spends calling the drawing methods, but the overhead of calling these kinds of drawing methods is usually negligible compared to the time it takes to execute them. (If we were generating an image pixel by pixel, this would not be the case, as the calculations would be the major time sink, not the drawing.)

So we can save time in our applet by having our update method set a clipping region that results in only the affected portion of the display being redrawn. We can pick the smallest rectangular area that includes both the old image position and the new image position, as shown in Figure 13.7. This is the only portion of the display that really needs to change; everything else stays the same.

Figure 13.7: Determining the clipping region

[Graphic: Figure 13-7]

An arbitrarily smart `update()` could save even more time by redrawing only those regions that have changed. However, the simple clipping strategy we've implemented here can be applied to many kinds of drawing, and gives quite good performance, particularly if the area being changed is small.

One important thing to note is that, in addition to looking at the new position, our updating operation now has to remember the last position at which the image was drawn. Let's fix our applet so it will use a clipping region. To keep this short and emphasize the changes, we'll take some liberties with design and make our next example a subclass of `TerribleFlicker`. Let's call it `ClippedFlicker`:

```
public class ClippedFlicker extends TerribleFlicker {
    int nextX, nextY;

    public void mouseDragged( MouseEvent e ) {
        nextX = e.getX(); nextY = e.getY();
        repaint();
    }
    void clipToAffectedArea( Graphics g, int oldx, int oldy, int newx,
                                    int newy, int width, int height) {
        int x = Math.min( oldx, newx );
        int y = Math.min( oldy, newy );
        int w = ( Math.max( oldx, newx ) + width ) - x;
        int h = ( Math.max( oldy, newy ) + height ) - y;
        g.setClip( x, y, w, h );
    }
    public void update( Graphics g ) {
        int lastX = currentX, lastY = currentY;
        currentX = nextX; currentY = nextY;
        clipToAffectedArea( g, lastX, lastY, currentX, currentY, imgWidth, imgHeight
);
        paint( g );
    }
}
```

You should find that `ClippedFlicker` is significantly faster, though it still flickers. We'll make one more change in the next section to eliminate that.

So, what have we changed? First, we've overridden `mouseDragged()` so that instead of setting the current coordinates of the image, it sets another pair of coordinates called `nextX` and `nextY`. These are the coordinates at which we'll display the image the next time we draw it.

`update()` now has the added responsibility of taking the next position and making it the current position, by setting the `currentX` and `currentY` variables. This effectively decouples `mouseDragged()` from our painting routines. We'll discuss why this is advantageous in a bit. `update()` then uses the current and next coordinates to set a clipping region on the `Graphics` object before handing it off to `paint()`.

We have created a new, `private` method to help it do this. `clipToAffectedArea()` takes as arguments the new and old coordinates and the width and height of the image. It determines the bounding rectangle as shown in Figure 13.6, then calls `setClip()` to set the clipping region. As a result, when `paint()` is called, it draws only the affected area of the screen.

So, what's the deal with `nextX` and `nextY`? By making `update()` keep track of the next, current, and last coordinates separately, we accomplish two things. First, we always have an accurate view of where the last image was drawn and second, we have decoupled where the next image will be drawn from `mouseDragged()`.

It's important to decouple painting from `mouseDragged()` because there isn't necessarily a one-to-one correspondence between calls to `repaint()` and subsequent calls by AWT to our `update()` method. This isn't a defect; it's a feature that allows AWT to schedule and consolidate painting requests. Our concern is that our `paint()` method may be called at arbitrary times while the mouse coordinates are changing. This is not necessarily bad. If we are trying to position our object, we probably don't want the display to be redrawn for every intermediate position of the mouse. It would slow down the dragging unnecessarily.

If we were concerned about getting every single change in the mouse's position, we would have two options. We could either do some work in the `mouseDragged()` method itself, or put our events into some kind of queue. We'll see an example of the first solution in our `DoodlePad` example a bit later. The latter solution would mean circumventing AWT's own event-scheduling capabilities and replacing them with our own, and we don't want to take on that responsibility.

Double Buffering

Now let's get to the most powerful technique in our toolbox: *double buffering*. Double buffering is a technique that fixes our flickering problems completely. It's easy to do and gives us almost flawless updates. We'll combine it with our clipping technique for better performance, but in general you can use double buffering with or without clipping.

Double buffering our display means drawing into an off-screen buffer and then copying our completed work to the display in a single painting operation, as shown in Figure 13.8. It takes the same amount of time to draw a frame, but double buffering instantaneously updates our display when it's ready.

Figure 13.8: Double buffering

[Graphic: Figure 13-8]

We can get this effect by changing just a few lines of our `ClippedFlicker` applet. Modify `update()` to look like the following and add the new `offScreenImage` instance variable as shown:

```
...
public class DoubleBufferedClipped extends ClippedFlicker {
    Image offScreenImage;
    Graphics offScreenGC;
    public void update( Graphics g ) {
        if ( offScreenImage == null ) {
            offScreenImage = createImage( getSize().width, getSize().height );
            offScreenGC = img.getGraphics();
        }
        int lastX = currentX, lastY = currentY;
        currentX = nextX; currentY = nextY;
        clipToAffectedArea( offScreenGC, lastX, lastY, currentX, currentY, imgWidth, 
imgHeight );
        clipToAffectedArea( g, lastX, lastY, currentX, currentY, imgWidth, imgHeight 
);
        paint( offScreenGC );
        g.drawImage(offScreenImage, 0, 0, this);
    }
}
...
```

Now, when you drag the image, you shouldn't see any flickering. The update rate should be about the same as in the previous example (or marginally slower), but the image should move from position to position without noticeable repainting.

So, what have we done this time? Well, the new instance variable, `offScreenImage`, is our off-screen buffer. It is a drawable `Image` object. We can get an off-screen `Image` for a component with the `createImage()` method. `createImage()` is similar to `getImage()`, except that it produces an empty image area of the specified size. We can then use the off-screen image like our standard display area by asking it for a graphics context with the `Image getGraphics()` method. After we've drawn into the off-screen image, we can copy that image back onto the screen with `drawImage()`.

The biggest change to the code is that we now pass `paint()` the graphics context of our off-screen buffer, rather than that of the on-screen display. `paint()` is now drawing on `offScreenImage`; it's our job to copy the image to the display when it's done. This might seem a little suspicious to you, as we are now using `paint()` in two capacities. AWT calls `paint()` whenever it's necessary to repaint our entire applet and passes it an on-screen graphics context. When we update ourselves, however, we call `paint()` to do its work on our off-screen area and then copy that image onto the screen from within `update()`.

Note that we're still clipping. In fact, we're clipping both the on-screen and off-screen buffers. Off-screen clipping has the same benefits we described earlier: AWT should be able to ignore wasted drawing operations. On-screen clipping minimizes the area of the image that gets drawn back to the display. If your display is fast, you might not even notice the savings, but it's an easy optimization, so we'll take advantage of it.

We create the off-screen buffer in `update()` because it's a convenient and safe place to do so. Also, note that our image observer probably won't be called, since `drawImage()` isn't doing anything nasty like scaling, and the image itself is always available.

The `dispose()` method of the `Graphics` class allows us to deallocate a graphics context explicitly when we are through with it. This is simply an optimization. If we were creating new graphics contexts frequently (say, in each paint()), we could give the system help in getting rid of them. This might provide some performance improvement when doing heavy drawing. We could allow garbage collection to reclaim the unused objects; however, the garbage collection process might be hampered if we are doing intense calculations or lots of repainting.

Off-Screen Drawing

In addition to serving as buffers for double buffering, off-screen images are useful for saving complex, hard-to-produce, background information. We'll look at a simple example: the "doodle pad." `DoodlePad` is a simple drawing tool that lets us scribble by dragging the mouse, as shown in Figure 13.9. It draws into an off-screen image; its `paint()` method simply copies the image to the display area.

Figure 13.9: The DoodlePad applet

[Graphic: Figure 13-9]

```
import java.awt.*;
import java.awt.event.*;
public class DoodlePad extends java.applet.Applet implements ActionListener {
    DrawPad dp;
    public void init() {
        setLayout( new BorderLayout() );
        add( "Center", dp = new DrawPad() );
        Panel p = new Panel();
        Button clearButton = new Button("Clear");
        clearButton.addActionListener( this );
        p.add( clearButton );
        add( "South", p );
    }
    public void actionPerformed( ActionEvent e ) {
        dp.clear();
    }
}
class DrawPad extends Canvas {
    Image drawImg;
    Graphics drawGr;
    int xpos, ypos, oxpos, oypos;
    DrawPad() {
        setBackground( Color.white );
        enableEvents( AWTEvent.MOUSE_EVENT_MASK
            | AWTEvent.MOUSE_MOTION_EVENT_MASK );
    }
    public void processEvent( AWTEvent e ) {
        int x = ((MouseEvent)e).getX(), y = ((MouseEvent)e).getY();
        if ( e.getID() == MouseEvent.MOUSE_DRAGGED ) {
            xpos = x; ypos = y;
            if ( drawGr != null )
                drawGr.drawLine( oxpos, oypos, oxpos=xpos, oypos=ypos );
            repaint();
        } else
        if ( e.getID() == MouseEvent.MOUSE_PRESSED ) {
            oxpos = x; oypos = y;
        }
        super.processEvent(e);
    }
    public void update( Graphics g ) {
        paint(g);
    }
    public void paint( Graphics g ) {
        if ( drawImg == null ) {
            drawImg = createImage( getSize().width, getSize().height );
            drawGr = drawImg.getGraphics();
        }
        g.drawImage(drawImg, 0, 0, null);
    }
    public void clear() {
        drawGr.clearRect(0, 0, getSize().width, getSize().height);
        repaint();
    }
}
```

Give it a try. Draw a nice moose, or a sunset. I just drew a lovely cartoon of Bill Gates. If you make a mistake, hit the **Clear** button and start over.

The parts should be familiar by now. We have made a type of `Canvas` called `DrawPad`. The new `DrawPad` component handles mouse events by enabling both simple mouse events (mouse clicks) and mouse motion events (mouse drags), and then overriding the `processEvent()` method to handle these events. By doing so, we are simulating the old (Java 1.0) event handling model; in this situation, it's a little more convenient than implementing all the methods of the `MouseListener` and `MouseMotionListener` interfaces. The `processEvent()` method handles `MOUSE_DRAGGED` movement events by drawing lines into an off-screen image and calling `repaint()` to update the display. `DrawPad`'s `paint()` method simply does a `drawImage()` to copy the off-screen drawing area to the display. In this way, `DrawPad` saves our sketch information.

What is unusual about `DrawPad` is that it does some drawing outside of `paint()` or `update()`. In our clipping example, we talked about decoupling `update()` and `mouseDragged()`; we were willing to discard some mouse movements in order to save some updates. In this case, we want to let the user scribble with the mouse, so we should respond to every mouse movement. Therefore, we do our work in `processEvent()` itself. As a rule, we should be careful about doing heavy work in event handling methods because we don't want to interfere with other tasks the AWT thread is performing. In this case, our line drawing operation should not be a burden, and our primary concern is getting as close a coupling as possible between the mouse movement events and the sketch on the screen.

In addition to drawing a line as the user drags the mouse, the part of `processEvent()` that handles `MOUSE_DRAGGED()` events maintains a set of old coordinates, to be used as a starting point for the next line segment. The part of `processEvent()` that handles `MOUSE_PRESSED` events resets the old coordinates to the current mouse position whenever the user picks up and moves to a new location. Finally, `DrawPad` provides a `clear()` method that clears the off-screen buffer and calls `repaint()` to update the display. The `DoodlePad` applet ties the `clear()` method to an appropriately labeled button through its `actionPerformed()` method.

What if we wanted to do something with the image after the user has finished scribbling on it? Well, as we'll see in the next section, we could get the pixel data for the image from its `ImageProducer` object and work with that. It wouldn't be hard to create a save facility that stores the pixel data and reproduces it later. Think about how you might go about creating a networked "bathroom wall" where people could scribble on your Web pages.

PREVIOUS HOME NEXT
Images BOOK INDEX Working With Images

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

14.2 Working with Audio

So you've read all the material on drawing and image processing, and you're wondering what in the world audio has to do with images. Well, not much actually, except that true multimedia presentations often combine image techniques such as animation with sound. So we're going to spend a few minutes here talking about audio, for lack of a better place to discuss it.

As we write this, the good people at Sun are hard at work developing the API that Java applets will use for playing audio. A future release of Java will undoubtedly have support for real-time and continuous audio streams, sound management, mixing, synchronization, and filtering. Unfortunately, at the moment, we can tell you only about the basics.

`java.applet.AudioClip` defines an interface for objects that can play sound. An object that implements `AudioClip` can be told to `play()` its sound data, `stop()` playing the sound, or `loop()` continually.

An applet can call its `getAudioClip()` method to retrieve sounds over the network. This method takes an absolute or relative URL to specify where the audio file is located. The viewer may take the sound from a cache or retrieve it over the network. The following applet, `NoisyButton`, gives a simple example:

```
import java.awt.*;
import java.awt.event.*;
public class NoisyButton extends java.applet.Applet implements ActionListener {
    java.applet.AudioClip sound;
    public void init() {
        sound = getAudioClip( getClass().getResource(getParameter("sound")) );
        Button button = new Button("Play Sound");
        button.addActionListener( this );
        add ( button );
    }
    public void actionPerformed( ActionEvent e ) {
        if ( sound != null )
            sound.play();
    }
}
```

```
    *** Update description...
```

NoisyButton retrieves an AudioClip from the server; we use getCodeBase() to find out where the applet lives and getParameter() to find the name of the audio file. (The applet tag that displays NoisyButton must include a parameter tag for sound.) Unfortunately, this is about the extent of what we can do with sound right now. If you want to experiment, there are a few additional methods in the sun.audio classes. Stay tuned for a bigger and better audio API as part of the upcoming Java Media package.

1.3 The java.io Package

The `java.io` package contains the classes that handle fundamental input and output operations in Java. Almost all fundamental I/O in Java is based on streams. A stream represents a flow of data, or a channel of communication, with a reading process at one end of the stream and a writing process at the other end, at least conceptually. As of Java 1.1, the `java.io` package is the largest of the fundamental packages. See Chapter 6, *I/O*, for a more in-depth description of the basic I/O capabilities provided by this package.

Java 1.0 supports only byte streams. The `InputStream` class is the superclass of all of the Java 1.0 byte input streams, while `OutputStream` is the superclass of all the byte output streams. A number of other byte stream classes extend the functionality of these basic streams. For example, the `FileInputStream` and `FileOutputStream` classes read from and write to files, respectively, while `DataInputStream` and `DataOutputStream` read and write binary representations of the primitive Java data types. The main problem with these byte streams is that they do not handle the conversion between the Unicode character set used internally by Java and other character sets used when reading or writing data.

As of Java 1.1, `java.io` contains classes that represent character streams. These character stream classes convert other character encodings that appear in I/O streams to and from Unicode characters. The `Reader` class is the superclass of all the Java 1.1 character input streams, while `Writer` is the superclass of all character output streams. Many of the reader and writer classes have analogous behavior to corresponding byte stream classes. For instance, `FileReader` and `FileWriter` are character streams that read from and write to files, respectively.

The `InputStreamReader` and `OutputStreamWriter` classes provide a bridge between byte streams and character streams. If you wrap an `InputStreamReader` around an `InputStream` object, the bytes in the byte stream are read and converted to characters using the character encoding scheme specified by the `InputStreamReader`. Likewise, you can wrap an `OutputStreamWriter` around any `OutputStream` object, which allows you to write characters and have them converted to bytes.

As of Java 1.1, `java.io` also contains classes to support *object serialization*. Object serialization is the ability to write the complete state of an object to an output stream, and then later recreate that object by reading in the serialized state from an input stream. The `ObjectOutputStream` and `ObjectInputStream` classes handle serializing and deserializing objects, respectively. These classes provide basic serialization capabilities for all objects that implement the `Serializable` interface. Chapter 7, *Object Serialization*, provides a more detailed explanation of the new object serialization functionality in Java 1.1.

The `RandomAccessFile` class is the only class in `java.io` that does not use a stream for reading or writing data. As its name implies, `RandomAccessFile` provides nonsequential access to a file, so you can use it to read from or write to specific locations in a file.

The `File` class represents a file on the local filesystem. The class provides methods to identify a file, both in terms of its path and its filename. There are also methods that retrieve information about a file, such as its status as a directory or a file, its length, and its last modification time.

See Chapter 11, *The java.io Package*, for complete reference material on all of the classes in the `java.io` package.

1.6 The java.text Package

The java.text package is new in Java 1.1. It contains classes that support the parsing and formatting of data. These classes also support the internationalization of Java programs. Internationalization refers to the process of making a program flexible enough to run correctly in any locale. An internationalized program must, however, be localized to enable it to run in a particular locale. The internationalization capabilities in Java are quite significant, especially in this age of the global Internet.

Many of the classes in java.text are meant to handle formatting string representations of dates, times, numbers, and messages based on the conventions of a locale. The Format class is the superclass of all of the classes that generate and parse string representations of various types of data.

The DateFormat class formats and parses dates and times according to the customs and language of a particular locale. By the same token, the NumberFormat class formats and parses numbers, including currency values, in a locale-dependent manner. The MessageFormat class creates a textual message from a pattern string, while ChoiceFormat maps numerical ranges to strings. By themselves, these classes do not provide different results for different locales. However, they can be used in conjunction with ResourceBundle objects from java.util that generate locale-specific pattern strings.

The Collator class handles collating strings according to the rules of a particular locale. Different languages have different characters and different rules for sorting those characters; Collator and its subclass, RuleBasedCollator, are designed to take those differences into account when collating strings. In addition, the CollationKey class optimizes the sorting of a large collection of strings.

The BreakIterator class finds various boundaries, such as word boundaries and line boundaries, in textual data. As you might expect, BreakIterator locates these boundaries according to the rules of a particular locale.

See Chapter 16, *The java.text Package*, for complete reference material on all of the classes in the java.text package.

PREVIOUS
The java.util Package

HOME
BOOK INDEX

NEXT
The java.math Package

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

2.3 String Concatenation

Java's string concatenation operator (`+`) provides special support for the `String` and `StringBuffer` classes. If either operand of the binary `+` operator is a reference to a `String` or `StringBuffer` object, the operator is the string concatenation operator instead of the arithmetic addition operator. The string concatenation operator produces a new `String` object that contains the concatenation of its operands; the characters of the left operand precede the characters of the right operand in the newly created string.

If one of the operands of the `+` operator is a reference to a string object and the other is not, the operator converts the nonstring operand to a string object using the following rules:

- A `null` operand is converted to the string literal `"null"`.
- If the operand is a non-`null` reference to an object that is not a string, the object's `toString()` method is called. The result of the conversion is the value returned by the object's `toString()` method, unless the return value is `null`, in which case the result of the conversion is the string literal `"null"`.
- A `char` operand is converted to a reference to a string object that has a length of one and contains that character.
- An integer operand (other than `char`) is converted to a string object that contains the base 10 string representation of its value. If the value is negative, the string starts with a minus sign; if it is positive there is no sign character. If the value is zero, the result of the conversion is `"0"`. Otherwise, the string representation of the integer does not have any leading zeros.
- If the operand is a floating-point value, the exact string representation depends on the value being converted. If its absolute value is greater than or equal to 10^-3 or less than or equal to 10^7, it is converted to a string with an optional minus sign (if the value is negative) followed by up to eight digits before the decimal point, a decimal point, and the necessary number of digits after the

decimal point (but no trailing zero if there is more than one significant digit). There is always a minimum of one digit after the decimal point.

- Otherwise, the value is converted to a string with an optional minus sign (if the value is negative), followed by a single digit, a decimal point, the necessary number of digits after the decimal point (but no trailing zero if there is more than one significant digit), and the letter `E` followed by a plus or a minus sign and a base 10 exponent of at least one digit. Again, there is always a minimum of one digit after the decimal point.
- The values `NaN`, `NEGATIVE_INFINITY`, `POSITIVE_INFINITY`, `-0.0`, and `+0.0` are represented by the strings `"NaN"`, `"--Infinity"`, `"Infinity"`, `"--0.0"`, and `"0.0"`, respectively.
- A `boolean` operand is converted to either the string literal `"true"` or the string literal `"false"`.

The following is a code example that uses the string concatenation operator:

```
// format seconds into hours, minutes, and seconds
String formatTime(int t) {
    int minutes, seconds;
    seconds = t%60;
    t /= 60;
    minutes = t%60;
    return t/60 + ":" + minutes + ":" + seconds;
}
```

Java uses `StringBuffer` objects to implement string concatenation. Consider the following code:

```
String s, s1, s2;
s = s1 + s2
```

To compute the string concatenation, Java's compiler generates this code:

```
s = new StringBuffer().append(s1).append(s2).toString()
```

3.2 Synchronizing Multiple Threads

The correct behavior of a multithreaded program generally depends on multiple threads cooperating with each other. This often involves threads not doing certain things at the same time or waiting for each other to perform certain tasks. This type of cooperation is called *synchronization*. This section discusses some common strategies for synchronization and how they can be implemented in Java.

The simplest strategy for ensuring that threads are correctly synchronized is to write code that works correctly when executed concurrently by any number of threads. However, this is more easily said than done. Most useful computations involve doing some activity, such as updating an instance variable or updating a display, that must be synchronized in order to happen correctly.

If a method only updates its local variables and calls other methods that only modify their local variables, the method can be invoked by multiple threads without any need for synchronization. `Math.sqrt()` and the `length()` method of the `String` class are examples of such methods.

A method that creates objects and meets the above criterion may not require synchronization. If the constructors invoked by the method do not modify anything but their own local variables and instance variables of the object they are constructing, and they only call methods that do not need to be synchronized, the method itself does not need to be synchronized. An example of such a method is the `substring()` in the `String` class.

Beyond these two simple cases, it is impossible to give an exhaustive list of rules that can tell you whether or not a method needs to be synchronized. You need to consider what the method is doing and think about any ill effects of concurrent execution in order to decide if synchronization is necessary.

Single-Threaded Execution

When more than one thread is trying to update the same data at the same time, the result may be wrong or inconsistent. Consider the following example:

```
class CountIt {
    int i = 0;
    void count() {
        i = i + 1;
    }
}
```

The method `count()` is supposed to increment the variable `i` by one. However, suppose that there are two threads, `A` and `B`, that call `count()` at the same time. In this case, it is possible that `i` could be incremented only once, instead of twice. Say the value of `i` is 7. Thread `A` calls the `count()` method and computes `i+1` as 8. Then thread `B` calls the `count()` method and computes `i+1` as 8 because thread `A` has not yet assigned the new value to `i`. Next, thread `A` assigns the value 8 to the variable `i`. Finally, thread `B` assigns the value 8 to the variable `i`. Thus, even though the `count()` method is called twice, the variable has only been incremented once when the sequence is finished.

Clearly, this code can fail to produce its intended result when it is executed concurrently by more than one thread. A piece of code that can fail to produce its intended result when executed concurrently is called a *critical section*. However, a critical section does behave correctly when it is executed by only one thread at a time. The strategy of single-threaded execution is to allow only one thread to execute a critical section of code at a time. If a thread wants to execute a critical section that another thread is already executing, the thread has to wait until the first thread is done and no other thread is executing that code before it can proceed.

Java provides the `synchronized` statement and the `synchronized` method modifier for implementing single-threaded execution. Before executing the block in a `synchronized` statement, the current thread must obtain a lock for the object referenced by the expression. If a method is declared with the `synchronized` modifer, the current thread must obtain a lock before it can invoke the method. If the method is not declared `static`, the thread must obtain a lock associated with the object used to access the method. If the method is declared `static`, the thread must obtain a lock associated with the class in which the method is declared. Because a thread must obtain a lock before executing a `synchronized` method, Java guarantees that `synchronized` methods are executed one thread at a time.

Modifying the `count()` method to make it a `synchronized` method ensures that it works as intended.

```
class CountIt {
    int i = 0;
    synchronized void count() {
        i = i + 1;
    }
}
```

The strategy of single-threaded execution can also be used when multiple methods update the same data. Consider the following example:

```
class CountIt2 {
    int i = 0;
    void count() {
        i = i + 1;
    }
    void count2() {
        i = i + 2;
    }
}
```

By the same logic used above, if the `count()` and `count2()` methods are executed concurrently, the result could be to increment `i` by 1, 2, or 3. Both the `count()` and `count2()` methods can be declared as `synchronized` to ensure that they are not executed concurrently with themselves or each other:

```
class CountIt2 {
    int i = 0;
    synchronized void count() {
        i = i + 1;
    }
    synchronized void count2() {
        i = i + 2;
    }
}
```

Sometimes it's necessary for a thread to make multiple method calls to manipulate an object without another thread calling that object's methods at the same time.

Consider the following example:

```
System.out.print(new Date());
System.out.print(" : ");
System.out.println(foo());
```

If the code in the example is executed concurrently by multiple threads, the output from the two threads will be interleaved. The `synchronized` keyword provides a way to ensure that only one thread at a time can execute a block of code. Before executing the block in a `synchronized` statement, the current thread must obtain a lock for the object referenced by the expression. The above code can be modified to give a thread exclusive access to the `OutputStream` object referenced by `System.out`:

```
synchronized (System.out) {
    System.out.print(new Date());
    System.out.print(" : ");
    System.out.println(foo());
}
```

Note that this approach only works if other code that wants to call methods in the same object also uses similar `synchronized` statements, or if the methods in question are all `synchronized` methods. In this case, the `print()` and `println()` methods are `synchronized`, so other pieces of code that need to use these methods do not need to use a `synchronized` statement.

Another situation in which simply making a method `synchronized` does not provide the needed single-threaded execution occurs when an inner class is updating fields in its enclosing instance. Consider the following code:

```
public class Z extends Frame {
    int pressCount = 0;
    ...
    private class CountButton extends Button
                              implements ActionListener {
        public void actionPerformed(ActionEvent evt) {
            pressCount ++;
        }
    }
    ...
}
```

If a `Z` object instantiates more than one instance of `CountButton`, you need to use single-threaded execution to ensure that updates to `pressCount` are done correctly. Unfortunately, declaring the `actionPerformed()` method of `CountButton` to be `synchronized` does not accomplish that goal because it only forces the method to acquire a lock on the instance of `CountButton` it is associated with before it executes. The object you need to acquire a lock for is the enclosing instance of `Z`.

One way to have a `CountButton` object capture a lock on its enclosing instance of `Z` is to update `pressCount` inside of a `synchronized` statement:

```
synchronized (Z.this) {
    pressCount ++;
}
```

The drawback to this approach is that every piece of code that accesses `pressCount` in any inner class of Z must be in a similar synchronized statement. Otherwise, it is possible for pressCount to be updated incorrectly. The more pieces of code that need to be inside of synchronized statements, the more places there are to introduce bugs in your program.

A more robust approach is to have the inner class update a field in its enclosing instance by calling a synchronized method in the enclosing instance:

```
public class Z extends Frame {
    int pressCount = 0;
    synchronized incrementPressCount() {
        pressCount++;
    }
    ...
    private class CountButton extends Button
                              implements ActionListener {
        public void actionPerformed(ActionEvent evt) {
            incrementPressCount();
        }
    }
    ...
}
```

Optimistic Single-Threaded Execution

When multiple threads are updating a data structure, single-threaded execution is the obvious strategy to use to ensure correctness of the operations on the data structure. However, single-threaded execution can cause some problems of its own. Consider the following example:

```
public class Queue extends java.util.Vector {
    synchronized public void put(Object obj) {
        addElement(obj);
    }
    synchronized public Object get() throws EmptyQueueException {
        if (size() == 0)
            throw new EmptyQueueException();
        Object obj = elementAt(0);
        removeElementAt(0);
        return obj;
    }
}
```

This example implements a first-in, first-out (FIFO) queue. If the get() method of a Queue object is called when the queue is empty, the method throws an exception. Now suppose that you want to write the `get()` method so that when the queue is empty, the method waits for an item to be put in the queue, rather than throwing an exception. In order for an item to be put in the queue, the `put()` method of the queue must be invoked. But using the single-threaded execution strategy, the `put()` method will never be able to run while the `get()` method is waiting for the queue to receive an item. A good way to solve this dilemma is to use a strategy called *optimistic single-threaded execution*.

The optimistic single-threaded execution strategy is similar to the single-threaded execution strategy. They both begin by getting a lock on an object to ensure that the currently executing thread is the only thread that can execute a piece of code, and they both end by releasing that lock. The difference is what happens in between. Using the optimistic single-threaded execution strategy, if a piece of code discovers that conditions are not right to proceed, the code releases the lock it has on the object that enforces single-threaded execution and waits. When another piece of code changes things in such a way that might allow the first piece of code to proceed, it notifies the first piece of code that it should try to regain the lock and proceed.

To implement this strategy, the `Object` class provides methods called `wait()`, `notify()`, and `notifyAll()`. These methods are inherited by every other class in Java. The following example shows how to implement a queue that uses the optimistic single-threaded execution strategy, so that when the queue is empty, its `get()` method waits for the queue to have an item put in it:

```
public class Queue extends java.util.Vector {
    synchronized public void put(Object obj) {
        addElement(obj);
        notify();
    }
    synchronized public Object get() throws EmptyQueueException {
        while (size() == 0)
            wait();
        Object obj = elementAt(0);
        removeElementAt(0);
        return obj;
    }
}
```

In the above implementation of the `Queue` class, the `get()` method calls `wait()` when the queue is empty. The `wait()` method releases the lock that excludes other threads from executing methods in the `Queue` object, and then waits until another thread calls the `put()` method. When `put()` is called, it adds an item to the queue and calls `notify()`. The `notify()` method tells a thread that is waiting to return from a `wait()` method that it should attempt to regain its lock and proceed. If there is more than one thread waiting to regain the lock on the object, `notify()` chooses one of the threads arbitrarily. The `notifyAll()` method is similar to `notify()`, but instead of choosing one thread to notify, it notifies all of the threads that are waiting to regain the lock on the object.

Notice that the `get()` method calls `wait()` inside a `while` loop. Between the time that `wait()` is notified that it should try to regain its lock and the time it actually does regain the lock, another thread may have called the `get()` method and emptied the queue. The `while` loop guards against this situation.

Rendezvous

Sometimes it is necessary to have a thread wait to continue until another thread has completed its work and died. This type of synchronization uses the rendezvous strategy. The `Thread` class provides the `join()` method for implementing this strategy. When the `join()` method is called on a `Thread` object, the method returns immediately if the thread is dead. Otherwise, the method waits until the thread dies and then returns.

Balking

Some methods should not be executed concurrently and have a time-sensitive nature that makes postponing calls to them a bad idea. This is a common situation when software is controlling real-world devices. Suppose you have a Java program that is embedded in an electronic control for a toilet. There is a method called `flush()` that is responsible for flushing a toilet, and `flush()` can be called from more than one thread. If a thread calls `flush()` while another thread is already executing `flush()`, the second call should do nothing. A toilet is capable of only one flush at a time, and having a concurrent call to the `flush()` method result in a second flush would only waste water.

This scenario suggests the use of the balking strategy. The balking strategy allows no more than one thread to execute a method at a time. If another thread attempts to execute the method, the method simply returns without doing anything. Here is an example that shows what such a `flush()` method might look like:

```
boolean busy;
void flush() {
    synchronized (this) {
        if (busy)
            return;
        busy = true;
    }
    // code to make flush happen goes here
    busy = false;
}
```

Explicit Synchronization

When the synchronization needs of a thread are not known in advance, you can use a strategy called explicit synchronization. The explicit synchronization strategy allows you to explicitly tell a thread when it can and cannot run. For example, you may want an animation to start and stop in response to external events that happen at unpredictable times, so you need to be able to tell the animation when it can run.

To implement this strategy, the `Thread` class provides methods called `suspend()` and `resume()`. You can suspend the execution of a thread by calling the `suspend()` method of the `Thread` object that controls the thread. You can later resume execution of the thread by calling the `resume()` method on the `Thread` object.

PREVIOUS | HOME | NEXT
Using Thread Objects | BOOK INDEX | Exception Handling

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

5.4 Hashtables

The `Dictionary` class is an `abstract` class that defines methods for associating key objects with value objects. Given a key, an instance of `Dictionary` is able to return its associated value. The `Hashtable` class is a concrete subclass of `Dictionary` that uses a data structure called a *hashtable* and a technique called *chained hashing* to allow values associated with keys to be fetched with minimal searching. You might use a `Hashtable` object to associate weather reports with the names of cities and towns, for example.

Before explaining hashtables or chained hashing, consider the problem of finding a key/value pair in an array that contains references to key/value pairs in no particular order. The array might look something like what is shown in Figure 5.1.

Figure 5.1: An array of key/value pairs

[Graphic: Figure 5-1]

Since we cannot make any assumptions about where in the array a key is to be found, the most reasonable search strategy is a linear search that starts at one end of the array and looks at each array element until it finds what it is looking for or reaches the other end of the array. For an array with just a few elements, a linear search is a reasonable strategy, but for an array with hundreds of elements it is not. If we know where in the array to look for a key, however, we can eliminate most of the searching effort. Knowing where to look for a key is the idea behind a hashtable.

With a hashtable, each key object has a relatively unique integer value that is called a *hashcode*. The `Object` class defines a `hashCode()` method, so every object in Java has such a method. The hashcode returned by this method computes an array index for a key object as follows:

```
array.length % hashCode()
```

This array index, or hash index, stores the key/value pair in a hashtable array. If there is nothing stored at that index, the key/value pair is placed at that position in the array. However, if there is already a key/value pair at that hash index, the `Hashtable` stores the key/value pair in a linked list at that position in the array. This strategy for managing multiple keys with the same hash index is called *chained hashing*. The array for hashtable that uses this strategy might look like Figure 5.2.

Figure 5.2: An array of key/value pairs that uses chained hashing

[Graphic: Figure 5-2]

Now, when we want to fetch a key/value pair, all we have to do is recalculate the hash index for the key object and look at that position in the hashtable array. If the key stored at that hash index is the right key, then we have found what we are looking for by examining only one array element instead of searching. However, if the key is not the right key, all we have to do is search the items in the linked list at that position to find our key/value pair.

You can create a Hashtable object using the constructor that takes no arguments:

```
Hashtable h = new Hashtable()
```

This constructor creates an empty Hashtable. There are other constructors that take parameters to allow you to tune the performance of a Hashtable object. The first parameter you can specify is the capacity of the hash table, which is the length of the array used to implement it. The longer the array, the less likely it is that multiple keys will share the same hash index. The default array length is 101. To create a Hashtable object with an array length of 1009, use the following constructor:

```
Hashtable h = new Hashtable(1009);
```

The number that you choose for the array length should be a prime number. If it is not, the key/value pairs stored in the array will tend to be less evenly distributed.

The load factor of a hashtable is the ratio of the number of key/value pairs in the hashtable to the array length. A load factor of 0 means that the Hashtable is empty. As the load factor increases, so does the likelihood that multiple key/value pairs will share the same hash index. When the load factor becomes greater than 1, it means that the number of key/value pairs in a hashtable is greater than the array length, so that at least one hash index is being shared by multiple key/value pairs. Clearly, a low load factor is better than a high load factor in terms of performance. You can specify the maximum permissible load factor for a Hashtable object when you create it. For example:

```
Hashtable h = new Hashtable(1009, .62);
```

If not specified, the maximum load factor for a Hashtable object is .75. When a key/value pair is added to a Hashtable that would otherwise cause the load factor to exceed the maximum value, the Hashtable performs a rehash. This means that the Hashtable creates a new array with a length one greater than double the length of the old array. It then recomputes the hash index for each key/value pair in the old array and stores each key/value pair in the new array at the new hash index. Obviously, this is an undesirable performance hit, so if you know approximately how many items you will add to a Hashtable, you should create one with an appropriate initial capacity.

After you have created a Hashtable object, you can add new key/value pairs to it, or modify the value in an existing key/value pair, by calling the put() method. The put() method takes two arguments: a reference to a key object and a reference to a value object. It first looks for a key/value pair in the hashtable with the key equal to the specified key. If there is such a key/value pair, the put() method replaces the previous value with the specified value and returns a reference to the previous value object. If, however, there is no such key/value pair, the put() method creates a new key/value pair, adds it to the hashtable and returns null. Here is a fragment of a class that uses a Hashtable to store weather forecasts.

```
import java.util.Hashtable;
class WeatherForecastDictionary {
    private Hashtable ht = new Hashtable(13291);
    public void putForecast(String locale, WeatherForecast forecast) {
        ht.put(locale, forecast);
    }
...
```

The `get()` method returns the value associated with a given key in a `Hashtable` object. It takes one argument that is a reference to the key it should search for. If the `get()` method does not find a key/value pair with a key equal to the specified key, it returns `null`. Here is a method that uses the `get()` method to retrieve a weather forecast:

```
public WeatherForecast getForecast(String locale) {
    return (WeatherForecast)ht.get(locale);
}
```

The various equality tests done by a `Hashtable` use a given key object's `equals()` method. Because of the way that an object's `hashCode()` and `equals()` methods are used by the `Hashtable` class, it is important that if you override the definition of either of these methods, you do so in a consistent way. In other words, if two objects are considered equal by the `equals()` method for the class, then the `hashCode()` method for each object must return the same hashcode value. If that is not the case, when those objects are used as keys in a `Hashtable` object, the `Hashtable` will produce inconsistent results.

Once you have added key/value pairs to a `Hashtable`, you can use the `keys()` and `elements()` methods to get `Enumeration` objects that iterate through the key and value objects, respectively. The `containsKey()` method allows you to search the `Hashtable` for a particular key object, while `contains()` searches for a particular value object. The `Hashtable` class also defines a `remove()` method for removing key/value pairs from a `Hashtable`.

6.2 Output Streams and Writers

The `OutputStream` class is an `abstract` class that defines methods to write a stream of bytes sequentially. Java provides subclasses of the `OutputStream` class for writing to files and byte arrays, among other things. Other subclasses of `OutputStream` can be chained together to provide additional logic, such as writing multibyte data types or converting data to a string representation. It is also easy to define a subclass of `OutputStream` that writes to another kind of destination.

In Java 1.1, the `Writer` class is an `abstract` class that defines methods to write to a stream of characters sequentially. Many of the byte-oriented subclasses of `OutputStream` have counterparts in the character-oriented world of `Writer` objects. Thus, there are subclasses of `Writer` for writing to files and character arrays.

OutputStream

The `OutputStream` class is the `abstract` superclass of all other byte output stream classes. It defines three `write()` methods for writing to a raw stream of bytes:

```
write(int b)
write(byte[] b)
write(byte[] b, int off, int len)
```

Some `OutputStream` subclasses may implement buffering to increase efficiency. `OutputStream` provides a method, `flush()`, that tells the `OutputStream` to write any buffered output to the underlying device, which may be a disk drive or a network.

Because the `OutputStream` class is `abstract`, you cannot create a "pure" `OutputStream`. However, the various subclasses of `OutputStream` can be used interchangeably. For example, methods often take `OutputStream` parameters. This means that such a method accepts any subclass of `OutputStream` as an argument.

`OutputStream` is designed so that `write(byte[])` and `write(byte[], int, int)` call `write(int)`. Thus, when you subclass `OutputStream`, you only need to define the `write()` method. However, for efficiency's sake, you should also override `write(byte[], int, int)` with a method that can write a block of data more efficiently than writing each byte separately.

Writer

The `Writer` class is the `abstract` parent class of all other character output stream classes. It defines nearly the same methods as `OutputStream`, except that the `write()` methods deal with characters instead of bytes:

```
write(int c)
write(char[] cbuf)
write(char[] cbuf, int off, int len)
write(String str)
write(String str, int off, int len)
```

`Writer` also includes a `flush()` method that forces any buffered data to be written to the stream.

`Writer` is designed so that `write(int)` and `write(char[])` both call `write(char[], int, int)`. Thus, when you subclass `Writer`, you only need to define the `write(char[], int, int)` method. Note that this design is different from, and more efficient than, that of `OutputStream`.

OutputStreamWriter

The `OutputStreamWriter` class serves as a bridge between `Writer` objects and `OutputStream` objects. Although an `OutputStreamWriter` acts like a character stream, it converts its characters to bytes using a character encoding scheme and writes them to an underlying `OutputStream`. This class is the output counterpart of `InputStreamReader`. When you create an `OutputStreamWriter`, specify the underlying `OutputStream` and, optionally, the name of an encoding scheme. The following example shows how to construct an `OutputStreamWriter` that writes characters to a file, encoded using the ISO 8859-5 encoding:

```
String fileName = "encodedfile.txt";
String encodingName = "8859_5";
OutputStreamWriter out;
try {
    FileOutputStream fileOut = new FileOutputStream (fileName);
    out = new OutputStreamWriter (fileOut, encodingName);
} catch (UnsupportedEncodingException e1) {
    System.out.println(encodingName + " is not a supported encoding scheme.");
} catch (IOException e2) {
    System.out.println("The file " + fileName + " could not be opened.");
}
```

FileWriter and FileOutputStream

The `FileOutputStream` class is a subclass of `OutputStream` that writes a stream of bytes to a file. The `FileOutputStream` class has no explicit open method. Instead, the file is implicitly opened, if appropriate, when you create the `FileOutputStream` object. There are several ways to create a `FileOutputStream`:

- You can create a `FileOutputStream` by passing the name of a file to be written:

  ```
  FileOutputStream f1 = new FileOutputStream("foo.txt");
  ```

- Another constructor is available in Java 1.1 that allows you to specify whether you want to append to the file or overwrite it. The following example constructs a `FileOutputStream` that appends the given file:

  ```
  FileOutputStream f1 = new FileOutputStream("foo.txt", true);
  ```

- You can create a `FileOutputStream` with a `File` object:

  ```
  File f = new File("foo.txt");
  FileOutputStream f2 = new FileOutputStream(f);
  ```

- You can create a `FileOutputStream` with a `FileDescriptor` object. A `FileDescriptor` encapsulates the native operating system's representation of an open file. You can get a `FileDescriptor` from a `RandomAccessFile` by calling its `getFD()` method. You create a `FileOutputStream` that writes to the open file associated with a `RandomAccessFile` as follows:

  ```
  RandomAccessFile raf;
  raf = new RandomAccessFile("z.txt","rw");
  FileInputStream f3 = new FileOutputStream(raf.getFD());
  ```

  The `FileWriter` class is a subclass of `Writer` that writes a stream of characters to a file. The characters to be written are converted to bytes using the default character encoding scheme. If you do not want to use the default encoding scheme, you need to wrap an `OutputStreamWriter` around a `FileOutputStream` as shown above. You can create a `FileWriter` from a filename, a `File` object, or a `FileDescriptor` object, as described above for `FileOutputStream`.

StringWriter

The `StringWriter` class is a subclass of `Writer` that stores its data in a `String` object. Internally, it uses a `StringBuffer`, which can be examined using `getBuffer()`. A `String` containing the data that has been written can be obtained with `toString()`. The following example creates a `StringWriter` and writes data into it:

```
StringWriter out = new StringWriter();
char[] buffer = {'b', 'o', 'o', '!', 'h', 'a'};
out.write('B');
out.write("uga");
out.write(buffer, 0, 4);
System.out.println(out.toString());
```

This example produces the following output:

```
Bugaboo!
```

CharArrayWriter and ByteArrayOutputStream

The `CharArrayWriter` class is a subclass of `Writer` that writes characters to an internal array. There are three ways to retrieve the data that has been written to the `CharArrayWriter`:

- The `toCharArray()` method returns a reference to a copy of the internal array.
- The `toString()` method returns a `String` constructed from the internal array.
- The `writeTo()` method writes the internal array to another `Writer`.

This example demonstrates how to create a `CharArrayWriter`, write data into it, and retrieve the data:

```
CharArrayWriter out = new CharArrayWriter();
try {
    out.write("Daphne");
}catch (IOException e) {}
char[] buffer = out.toCharArray();
System.out.println(buffer);
String result = out.toString();
System.out.println(result);
```

This example produces the following output:

```
Daphne
Daphne
```

The internal buffer of the `CharArrayWriter` is expanded as needed when data is written. If you know how many characters you will be writing, you can make your `CharArrayWriter` a little more efficient by passing an initial size to its constructor.

`ByteArrayOutputStream` is the byte-oriented equivalent of `CharArrayWriter`. It works in much the same way, with the following exceptions:

- The `write()` methods deal with bytes, not characters. Additionally, `ByteArrayOutputStream` does not have the `write(String)` methods that `CharArrayWriter` defines.
- Instead of `toCharArray()`, `ByteArrayOutputStream` has a `toByteArray()` method.
- Three `toString()` methods are provided. The one with no arguments converts the bytes in the internal array to characters using the default encoding scheme.[1] In Java 1.1, the `toString(int)` method is deprecated, since it does not convert bytes to characters appropriately. Instead, pass an encoding name to `toString(String)`; this method correctly converts the internal byte array to a character string.

[1] In Java 1.1, the default encoding scheme is used for the conversion. In earlier versions, characters are simply created using the eight bits of each byte as the low eight bits of the character.

PipedOutputStream and PipedWriter

The `PipedOuputStream` class is a subclass of `OutputStream` that facilitates communication between threads. A `PipedOutputStream` must be connected to a `PipedInputStream` to be useful, as it writes bytes that can be read by a connected `PipedInputStream`. There are a few ways to connect a `PipedOutputStream` to a `PipedInputStream`. You can first create the `PipedInputStream` and pass it to the `PipedOutputStream` constructor like this:

```
PipedInputStream pi = new PipedInputStream();
PipedOutputStream po = new PipedOutputStream(pi);
```

You can also create the `PipedOutputStream` first and pass it to the `PipedInputStream` constructor like this:

```
PipedOutputStream po = new PipedOutputStream();
PipedInputStream pi = new PipedInputStream(po);
```

The `PipedOutputStream` and `PipedInputStream` classes each have a `connect()` method you can use to explicitly connect a `PipedOutputStream` and a `PipedInputStream` as follows:

```
PipedOutputStream po = new PipedOutputStream();
PipedInputStream pi = new PipedInputStream();
po.connect(pi);
```

Or you can use `connect()` as follows:

```
PipedOutputStream po = new PipedOutputStream();
PipedInputStream pi = new PipedInputStream();
pi.connect(po);
```

Only one `PipedInputStream` can be connected to a `PipedOutputStream` at a time. If you use a `connect()` method to connect a `PipedOutputStream` to an already connected `PipedInputStream`, any unread bytes from the previously connected `PipedOutputStream` are lost.

`PipedWriter` is the character-based equivalent of `PipedOutputStream`. It works in the same way, except that a `PipedWriter` is connected to a `PipedReader` to complete the pipe, using either the appropriate constructor or the `connect()` method.

FilterOutputStream and FilterWriter

The `FilterOutputStream` class is a wrapper class for `OutputStream` objects. Conceptually, objects that belong to a subclass of `FilterOutputStream` are wrapped around another `OutputStream` object. The constructor for this class requires an `OutputStream`. The constructor sets the object's `out` instance variable to reference the specified `OutputStream`, so from that point on, the `FilterOutputStream` is associated with the given `OutputStream`. All of the methods of `FilterOutputStream` work by calling the corresponding methods in the underlying `OutputStream`. Because the `close()` method of a `FilterOutputStream` calls the `close()` method of the `OutputStream` that it wraps, you do not need to explicitly close the underlying `OutputStream`.

A FilterOutputStream does not add any functionality to the object that it wraps, so by itself it is not very useful. However, subclasses of the FilterOutputStream class do add functionality to the objects that they wrap in two ways:

- Some subclasses add logic to the methods of OutputStream. For example, the BufferedOutputStream class adds logic that buffers write operations.

- Other subclasses add new methods. An example of this is DataOutputStream, which provides methods for writing primitive Java data types to the stream.

The FilterWriter class is the character-based equivalent of FilterOutputStream. A FilterWriter is wrapped around an underlying Writer object; the methods of FilterWriter call the corresponding methods of the underlying Writer. However, unlike FilterOutputStream, FilterWriter is an abstract class, so you cannot instantiate it directly.

DataOutputStream

The DataOutputStream class is a subclass of the FilterOutputStream class that provides methods for writing a variety of data types to an OutputStream. The DataOutputStream class implements the DataOutput interface, so it defines methods for writing all of the primitive Java data types.

You create a DataOutputStream by passing a reference to an underlying OutputStream to the constructor. Here is an example that creates a DataOutputStream and uses it to write the length of an array as an int and then to write the values in array as long values:

```
void writeLongArray(OutputStream out, long[] a) throws IOException {
    DataOutputStream dout = new DataOutputStream(out);
    dout.writeInt(a.length);
    for (int i = 0; i < a.length; i++) {
        dout.writeLong(a[i]);
    }
}
```

BufferedWriter and BufferedOutputStream

The BufferedWriter class is a subclass of Writer that stores output destined for an underlying Writer in an internal buffer. When the buffer fills up, the entire buffer is written, or flushed, to the underlying Writer. Using a BufferedWriter is usually faster than using a regular Writer because it reduces the number of calls that must be made to the underlying device, be it a disk or a network. You can use the flush() method to force a BufferedWriter to write the contents of the buffer to the underlying Writer.

The following example shows how to create a BufferedWriter around a network socket's output stream:

```
public Writer getBufferedWriter(Socket s) throws IOException {
    OutputStreamWriter converter = new OutputStreamWriter(s.getOutputStream());
    return new BufferedWriter(converter);
}
```

First, create an OutputStreamWriter that converts characters to bytes using the default encoding scheme. After they are converted, the bytes are written to the socket. Then simply wrap a BufferedWriter around the OutputStreamWriter to buffer the output.

The BufferedOutputStream class is the byte-based equivalent of BufferedWriter. It works in the same way as BufferedWriter, except that it buffers output for an underlying OutputStream. Here's how you would rewrite the previous example to create a BufferedOutputStream around a socket:

```
public OutputStream getBufferedOutputStream(Socket s) throws IOException {
    return new BufferedOutputStream(s.getOutputStream());
}
```

PrintWriter and PrintStream

The PrintWriter class is a subclass of Writer that provides a set of methods for printing string representations of every Java data type. A PrintWriter can be wrapped around an underlying Writer object or an underlying OutputStream object. In the case of wrapping an OutputStream, any characters written to the PrintWriter are converted to bytes using the default encoding scheme.[2] Additional constructors allow you to specify if the underlying stream should be flushed after every line-separator character is written.

> [2] You can achieve the same effect using an OutputStreamWriter, but it is easier to use the PrintWriter(OutputStream) constructor. However, if you want to use an encoding scheme other than the default one, you need to create your own OutputStreamWriter.

The PrintWriter class provides a print() and a println() method for every primitive Java data type. As their names imply, the println() methods do the same thing as their print() counterparts, but also append a line separator character.

The following example demonstrates how to wrap a PrintWriter around an OutputStream:

```
boolean b = true;
char c = '%'
double d = 8.31451
int i = 42;
String s = "R = ";
PrintWriter out = new PrintWriter(System.out, true);
out.print(s);
out.print(d);
out.println();
out.println(b);
out.println(c);
out.println(i);
```

This example produces the following output:

```
R = 8.31451
```

```
true
%
42
```

PrintWriter objects are often used to report errors. For this reason, the methods of this class do not throw exceptions. Instead, the methods catch any exceptions thrown by any downstream OutputStream or Writer objects and set an internal flag, so that the object can remember that a problem occurred. You can query the internal flag by calling the checkError() method.

Although you can create a PrintWriter that flushes the underlying stream every time a line-separator character is written, this may not always be exactly what you want. Suppose that you are writing a program that has a character-based user interface, and that you want the program to output a prompt and then allow the user to input a response on the same line. In order to make this work with a PrintWriter, you need to get the PrintWriter to write the characters in its buffer without writing a line separator. You can do this by calling the flush() method.

PrintWriter is new as of Java 1.1; it is more capable than the PrintStream class. You should use PrintWriter instead of PrintStream because it uses the default encoding scheme to convert characters to bytes for an underlying OutputStream. The constructors for PrintStream are deprecated in Java 1.1. In fact, the whole class probably would have been deprecated, except that it would have generated a lot of compilation warnings for code that uses System.out and System.err.

6.3 File Manipulation

While streams are used to handle most types of I/O in Java, there are some nonstream-oriented classes in `java.io` that are provided for file manipulation. Namely, the `File` class represents a file on the local filesystem, while the `RandomAccessFile` class provides nonsequential access to data in a file. In addition, the `FilenameFilter` interface can be used to filter a list of filenames.

File

The `File` class represents a file on the local filesystem. You can use an instance of the `File` class to identify a file, obtain information about the file, and even change information about the file. The easiest way to create a `File` is to pass a filename to the `File` constructor, like this:

```
new File("readme.txt")
```

Although the methods that the `File` class provides for manipulating file information are relatively platform independent, filenames must follow the rules of the local filesystem. The `File` class does provide some information that can be helpful in interpreting filenames and path specifications. The variable `separatorChar` specifies the system-specific character used to separate the name of a directory from what follows.[3] In a Windows environment, this is a backslash (`\`), while in a UNIX or Macintosh environment it is a forward slash (`/`). You can create a `File` object that refers to a file called `readme.txt` in a directory called `myDir` as follows:

```
new File("myDir" + File.separatorChar + "readme.txt")
```

The `File` class also provides some constructors that make this task easier. For example, there is a `File` constructor that takes two strings as arguments: the first string is the name of a directory and the second

[3] This information is also available as `System.getProperty('file.separator')`, which is how the `File` class gets it.

string is the name of a file. The following example does the exact same thing as the previous example:

```
new File("myDir", "readme.txt")
```

The `File` class has another constructor that allows you to specify the directory of a file using a `File` object instead of a `String`:

```
File dir = new File("myDir");
File f = new File(dir, "readme.txt");
```

Sometimes a program needs to process a list of files that have been passed to it in a string. For example, such a list of files is passed to the Java environment by the `CLASSPATH` environment variable and can be accessed by the expression:

```
System.getProperty("java.class.path")
```

This list contains one or more filenames separated by separator characters. In a Windows or Macintosh environment, the separator character is a semicolon (`;`), while in a UNIX environment, the separator character is a colon (`:`). The system-specific separator character is specified by the `pathSeparatorChar` variable. Thus, to turn the value of `CLASSPATH` into a collection of `File` objects, we can write:

```
StringTokenizer s;
Vector v = new Vector();
s = new StringTokenizer(System.getProperty("java.class.path"),
                        File.pathSeparator);
while (s.hasMoreTokens())
    v.addElement(new File(s.nextToken()));
```

You can retrieve the pathname of the file represented by a `File` object with `getPath()`, the filename without any path information with `getName()`, and the directory name with `getParent()`.

The `File` class also defines methods that return information about the actual file represented by a `File` object. Use `exists()` to check whether or not the file exists. `isDirectory()` and `isFile()` tell whether the file is a file or a directory. If the file is a directory, you can use `list()` to get an array of filenames for the files in that directory. The `canRead()` and `canWrite()` methods indicate whether or not a program is allowed to read from or write to a file. You can also retrieve the length of a file with `length()` and its last modified date with `lastModified()`.

A few `File` methods allow you to change the information about a file. For example, you can rename a file with `rename()` and delete it with `delete()`. The `mkdir()` and `mkdirs()` methods provide a way to create directories within the filesystem.

Many of these methods can throw a SecurityException if a program does not have permission to access the filesystem, or particular files within it. If a SecurityManager has been installed, the checkRead() and checkWrite() methods of the SecurityManager verify whether or not the program has permission to access the filesystem.

FilenameFilter

The purpose of the FilenameFilter interface is to provide a way for an object to decide which filenames should be included in a list of filenames. A class that implements the FilenameFilter interface must define a method called accept(). This method is passed a File object that identifies a directory and a String that names a file. The accept() method is expected to return true if the specified file should be included in the list, or false if the file should not be included. Here is an example of a simple FilenameFilter class that only allows files with a specified suffix to be in a list:

```
import java.io.File;
import java.io.FilenameFilter;
public class SuffixFilter implements FilenameFilter {
    private String suffix;
    public SuffixFilter(String suffix) {
        this.suffix = "." + suffix;
    }
    public boolean accept(File dir, String name) {
        return name.endsWith(suffix);
    }
}
```

A FilenameFilter object can be passed as a parameter to the list() method of File to filter the list that it creates. You can also use a FilenameFilter to limit the choices shown in a FileDialog.

RandomAccessFile

The RandomAccessFile class provides a way to read from and write to a file in a nonsequential manner. The RandomAccessFile class has two constructors that both take two arguments. The first argument specifies the file to open, either as a String or a File object. The second argument is a String that must be either "r" or "rw". If the second argument is "r", the file is opened for reading only. If the argument is "rw", however, the file is opened for both reading and writing. The close() method closes the file. Both constructors and all the methods of the RandomAccessFile class can throw an IOException if they encounter an error.

The `RandomAccessFile` class defines three different `read()` methods for reading bytes from a file. The `RandomAccessFile` class also implements the `DataInput` interface, so it provides additional methods for reading from a file. Most of these additional methods are related to reading Java primitive types in a machine-independent way. Multibyte quantities are read assuming the most significant byte is first and the least significant byte is last. All of these methods handle an attempt to read past the end of file by throwing an `EOFException`.

The `RandomAccessFile` class also defines three different `write()` methods for writing bytes of output. The `RandomAccessFile` class also implements the `DataOutput` interface, so it provides additional methods for writing to a file. Most of these additional methods are related to writing Java primitive types in a machine-independent way. Again, multibyte quantities are written with the most significant byte first and the least significant byte last.

The `RandomAccessFile` class would not live up to its name if it did not provide a way to access a file in a nonsequential manner. The `getFilePointer()` method returns the current position in the file, while the `seek()` method provides a way to set the position. Finally, the `length()` method returns the length of the file in bytes.

PREVIOUS	HOME	NEXT
Output Streams and Writers	BOOK INDEX	Object Serialization

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

PREVIOUS

NEXT

7.3 Versioning of Classes

One you have written a class that works with serialization, the next concern is that serialized instances of that class can be deserialized by programs that use a different version of the same class.

After a class is written, it is often necessary to modify its definition as requirements change or new features are needed. Deserialization may fail if the definition of a class in use when an instance was serialized is different than the definition in use when the instance is deserialized. If you do not take any measures to assure the serialization mechanism that the two classes are different versions of the same class, deserialization fails by throwing an `InvalidClassException`. And even if the serialization mechanism is satisfied that the two class definitions represent different versions of the same class, it may find incompatible differences between the definitions.

The following changes to the definition of a class are noticed by the serialization mechanism:

- Adding or deleting instance variables.
- Moving a class up or down the inheritance hierarchy.
- Making a non-`static`, non-`transient` variable either `static` or `transient` has the same effect as deleting the variable. Similarly, changing a variable that is `static` or `transient` to be non-`static` or non-`transient` has the same effect as adding the variable.
- Changing the data type of a `transient` variable from a primitive data type to an object reference type or from an object reference type to a primitive data type.
- Changing the `readObject()` or `writeObject()` method of a class so that it calls `defaultReadObject()` or `defaultWriteObject()` when it did not previously, or so that it does not call one of these methods when it did previously. The removal or addition of a `readObject()` or `writeObject()` method that does not call `defaultReadObject()` or `defaultWriteObject()` has a similar effect.

- Changing a class from `Serializable` to `Externalizable` or from `Externalizable` to `Serializable`.

It's possible to code around some of these problems if you can first convince the serialization mechanism that the two class definitions are different versions of the same class. In order to convince the serialization mechanism of such a thing, the class definition used for deserialization of an object must define a `static final long` variable named `serialVersionUID`. If the class used for serialization also defined that variable with the same value, the two class definitions are assumed to define different versions of the same class.

If the class used for serialization does not define `serialVersionUID`, the serialization mechanism performs the comparison using a value that is computed by calling the `ObjectStreamClass.getSerialVersionUID()` method. That computation is based on the fields defined by the class. To take advantage of this automatic computation when you define `serialVersionUID`, you should use the *serialver* program that comes with the JDK to determine the appropriate value for `serialVersionUID`. The *serialver* program computes a value for `serialVersionUID` by calling the `ObjectStreamClass.getSerialVersionUID()` method.

Assuming you've convinced the serialization mechanism that the two class definitions represent different versions of the same class, here is some advice on how to deal with the differences that can be worked around:

Missing variables

> If the class used to deserialize an object defines variables the class used to serialize the object did not define, the serialized object does not contain any values for those variables. This situation can also arise if the class used to serialize the object defined a variable as `static` or `transient`, while the class used to deserialize the object defines it as non-`static` or non-`transient`.
>
> When an object is deserialized and there are variables missing in its serialized form, the variables in the deserialized object are set to default values. In other words, the value of such a variable is `true` if it has an arithmetic data type, `false` if it has a `boolean` data type, or `null` if it has an object reference type. Deserialization ignores intializers in variable declarations.
>
> When you add variables to a `Serializable` class, consider the possibility that the new version of the class will deserialize an object serialized with an older version of the class. If that happens and it is unacceptable for the new variables to have default values after deserialization, you can define a `validateObject()` method for the class to check for the default values and provide acceptable values or throw an `InvalidObjectException`.

Extra variables

If the serialized form of an object contains values for variables that are not defined by the class used to deserialize that object, the values are read and then ignored. If the value of such a variable is an object, the object is created and immediately becomes a candidate for garbage collection.

Missing classes

If the class used to deserialize an object inherits from an ancestor class that the class used to serialize the object did not inherit from, the serialized object does not contain any values for the variables of the additional ancestor class. Just as with missing variables, those variables are deserialized with their default values.

When you add an ancestor class to a `Serializable` class, consider the possibility that the new version of the class will deserialize an object serialized with an older version of the class. If that happens and it is unacceptable for instance variables in the new ancestor class to have default values after deserialization, you can define a `validateObject()` method for the class to check for the default values and provide acceptable values or throw an `InvalidObjectException`.

Extra classes

If the class used to serialize an object inherits from an ancestor class that the class used to deserialize the object does not inherit from, the values for the variables defined by that extra ancestor class are read but not used.

Adding writeObject() and readObject() methods

You can add `writeObject()` and `readObject()` methods to a class that did not have them. In order to deserialize objects that were serialized using the older class definition, the new methods must begin by calling `defaultWriteObject()` and `defaultReadObject()`. That ensures that information written out using default logic is still processed using default logic.

If the `writeObject()` and `readObject()` methods write and read additional information to and from the byte stream, you should also add an additional variable to the class to serve as a version indicator. For example, you might declare an `int` variable and initialize it to one. If, after `defaultReadObject()` returns, the value of that variable is `0`, you know the object was serialized using the old class definition and that any additional information that would have been written by the `writeObject()` method will not be there.

Removing writeObject() and readObject() methods

If you remove `writeObject()` and `readObject()` methods from a class and deserialize an object using the new class definition, the information written by a call to `writeObject()` is simply read by the default logic and any additional information is ignored.

Changing a class so that it implements Serializable

If a superclass of an object did not implement `Serializable` when the object was serialized, and that superclass does implement `Serializable` when the object is deserialized, the result is similar to the missing class situation. There is no information about the variables of the newly `Serializable` superclass in the byte stream, so its instance variables are initialized to default values.

Changing a class so that it does not implement Serializable

If a superclass of an object implemented `Serializable` when the object was serialized, and that superclass does not implement `Serializable` when the object is deserialized, the result is similar to the extra class situation. The information in the byte stream for that class is read and discarded.

PREVIOUS | HOME | NEXT

Writing Classes to Work with Serialization | BOOK INDEX | Networking

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

8.2 URL Objects

The URL class provides higher-level access to data than sockets do. A URL object encapsulates a Uniform Resource Locator (URL) specification. Once you have created a URL object, you can use it to access the data in the location specified by the URL. A URL allows you to access the data without needing to be aware of the details of the protocol being used, such as HTTP or FTP. For some types of data, a URL object provides a way to get the data already encapsulated in an appropriate kind of object. For example, a URL can provide JPEG data encapsulated in an ImageProducer object or text data encapsulated in a String object.

You can create a URL object as follows:

```
try {
    URL js = new URL("../../../../www.javasoft.com/index.html");
}catch (MalformedURLException e) {
    return;
}
```

This type of URL specification is called an absolute URL specification because it completely specifies where to find the data. It is also possible to create a URL object with a relative URL specification that is combined with an absolute specification:

```
try {
   URL jdk = new URL(js,"java.sun.com/products/JDK/index.html");
}catch (MalformedURLException e) {
    return;
}
```

In this example, the URL created in the previous example is combined with a relative URL specification that doesn't specify a network address or a root directory. The constructor can only combine the specifications if the protocol for both specifications is the same. If no protocol is specified, HTTP is assumed. The rules for combining the specifications depend on the protocol. In fact, the syntax rules for the portion of the URL after the protocol and up to an optional # depend on the protocol. If there's a # in the URL specification, the portion of the spec after the # is considered reference information that specifies a location within a file.

Once you have created a `URL` object, you can use the following access methods to get the information that the `URL` object encapsulates:

- `getProtocol()`
- `getHost()`
- `getFile()`
- `getPort()`
- `getRef()`

If you want to determine if two `URL` objects refer to the same file, you can use the `sameFile(URL)` method, which compares all the information in two `URL` objects except the reference information.

The highest level of functionality available from a `URL` object is provided by the `getContent()` method. The `getContent()` method tries to determine the type of data in the file specified by the `URL`, and then it returns the contents of the file encapsulated in an appropriate object for that type of data. For example, if the file contains GIF data, `getContent()` returns an ImageProducer object. If the type of data is not explicitly specified, `getContent()` tries to guess the type from the filename extension and possibly also from the contents of the file. The data type names that Java uses conform to the naming scheme for MIME data types, as do the filename extensions that are recognized. The data types that correspond to various file extensions are shown in Table 8.2.

Table 8.2: File Extensions and Data Types

Suffix	**Data Type**	**Suffix**	**Data Type**
.a [1]	`application/octet-stream`	*.ms*	`application/x-troff-ms`
.ai	`application/postscript`	*.mv*	`video/x-sgi-movie`
.aif	`audio/x-aiff`	*.nc*	`application/x-netcdf`
.aifc	`audio/x-aiff`	*.o* [1]	`application/octet-stream`
.aiff	`audio/x-aiff`	*.obj* [2]	`application/octet-stream`
.arc	`application/octet-stream`	*.oda*	`application/oda`

.au	audio/basic	*.pbm*	image/x-portable-bitmap
.avi	application/x-troff-msvideo	*.pdf*	application/pdf
.bcpio	application/x-bcpio	*.pgm*	image/x-portable-graymap
.bin	application/octet-stream	*.pl*	text/plain
.c	text/plain	*.pnm*	image/x-portable-anymap
.c++	text/plain	*.ppm*	image/x-portable-pixmap
.cc	text/plain	*.ps*	application/postscript
.cdf	application/x-netcdf	*.qt*	video/quicktime
.cpio	application/x-cpio	*.ras*	image/x-cmu-rast
.dump	application/octet-stream	*.rgb*	image/x-rgb
.dvi	application/x-dvi	*.roff*	application/x-troff
.el	text/plain	*.rtf* [2]	application/rtf
.eps	application/postscript	*.rtx*	application/rtf
.etx	text/x-setext	*.saveme*	application/octet-stream
.exe	application/octet-stream	*.sh*	application/x-shar
.gif	image/gif	*.shar*	application/x-shar
.gtar	application/x-gtar	*.snd*	audio/basic
.gz	application/octet-stream	*.src*	application/x-wais-source
.h	text/plain	*.sv4cpio*	application/x-sv4cpio
.hdf	application/x-hdf	*.sv4crc*	application/x-sv4crc
.hqx	application/octet-stream	*.t*	application/x-troff
.htm	text/html	*.tar*	application/x-tar
.html	text/html	*.tex*	application/x-tex
.ief	image/ief	*.texi*	application/x-texinfo
.java	text/plain	*.texinfo*	application/x-texinfo
.jfif	image/jpeg	*.text*	text/plain
.jfif-tbnl	image/jpeg	*.tif*	image/tiff
.jpe	image/jpeg	*.tiff*	image/tiff
.jpeg	image/jpeg	*.tr*	application/x-troff
.jpg	image/jpeg	*.tsv*	text/tab-separated-values
.latex	application/x-latex	*.txt*	text/plain
.lib [2]	application/octet-stream	*.ustar*	application/x-ustar

.man	application/x-troff-man	*.uu*	application/octet-stream
.me	application/x-troff-me	*.wav*	audio/x-wav
.mime	message/rfc822	*.wsrc*	application/x-wais-source
.mov	video/quicktime	*.xbm*	image/x-xbitmap
.movie	video/x-sgi-movie	*.xpm*	image/x-xpixmap
.mpe	video/mpeg	*.xwd*	image/x-xwindowdump
.mpeg	video/mpeg	*.z* [2]	application/octet-stream
.mpg	video/mpeg	*.zip* [2]	application/zip

Footnotes:

[1] UNIX only.

[2] Windows only.

If the filename does not end with a recognized extension, the first few bytes of the file are examined. If the first few bytes match the signature of a known type, the file is assumed to be of that type. Table 8.3 shows the byte combinations that are recognized.

Table 8.3: File Contents and Corresponding File Type

File Begins with	**Inferred Data Type**
"GIF8"	image/gif
"#def"	image/x-bitmap
"! XPM2"	image/x-pixmap
"<html>"	text/html
"<head>"	text/html
"<body>"	text/html

If you want to access the raw contents of a file instead of getting it encapsulated in an object, you can call the `openStream()` method of a `URL`. The `openStream()` method returns a reference to an `InputStream` object that you can use to read the file.

URLConnection Objects

After a URL object has parsed its specification, it actually creates a URLConnection object that is responsible for the protocol that it uses. The URLConnection is also responsible for determining the type of data in the file. The object is an instance of a subclass of URLConnection that is specific to the protocol specified by the URL object. As of Java 1.1, the java.net package includes the HttpURLConnection class for the HTTP protocol.

The URLConnection object for a URL provides complete control over the downloading of data from that URL. Unfortunately, the functionality of URLConnection is quite complex and goes beyond the scope of this book. For a detailed explanation of URLConnection, see *Java Network Programming* by Eliotte Rusty Harold, published by O'Reilly & Associates.

PREVIOUS | HOME | NEXT
Sockets | BOOK INDEX | Security

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

9.2 ClassLoader

Java supports dynamically loaded classes, so the class loading mechanism plays an important role in the Java security model. The default class loading mechanism in Java loads classes from local files found relative to directories specified by the `CLASSPATH` environment variable. The `CLASSPATH` environment variable should have a value made up of one or more directory paths separated by a colon. The path implied by the package of a class is relative to the directories specified in the `CLASSPATH` environment variable.

In contrast, an instance of the `java.lang.ClassLoader` class defines how classes are loaded over the network. You can specify a security policy for loading classes by defining a subclass of `ClassLoader` that implements the policy. The `loadClass()` method of a `ClassLoader` loads a top-level class, such as a subclass of `Applet`. That `ClassLoader` object then becomes associated with the loaded class. You can retrieve the `ClassLoader` object that loads the class by calling the `getClassLoader()` of an instance of the loaded class; every class in Java inherits this method from the `Object` class.

An object of a class loaded using a `ClassLoader` can attempt to load additional classes without explicitly using a `ClassLoader` object. The object does this by calling the `forName()` method of the `Class` class. However, if a `ClassLoader` object is associated with any pending method invocation in the current thread, the `forName()` method uses that `ClassLoader` to load the additional classes. In essence, this means that the object can only load classes through its associated `ClassLoader`.

If Java security is implemented correctly, an untrusted applet cannot escape the security policy implemented by the `ClassLoader` object used to load it because it cannot access any other `ClassLoader` objects. An applet should not be able to create its own `ClassLoader` objects. It is the responsibility of the `checkCreateClassLoader()` method of `SecurityManager` to enforce this restriction.

Because a `SecurityManager` can determine the `ClassLoader`, if any, used to load a class, it can use the `ClassLoader` to help determine the trustworthiness ofthe class. Classes loaded by different

ClassLoader objects cannot accidentally be mixed up because a class is identified by the combination of its fully qualified name and its ClassLoader.

BufferedOutputStream

Name

BufferedOutputStream

Synopsis

Class Name:

`java.io.BufferedOutputStream`

Superclass:

`java.io.FilterOutputStream`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

A `BufferedOutputStream` object provides a more efficient way to write just a few bytes at a time to an `OutputStream`. `BufferedOutputStream` objects use a buffer to store output for an associated `OutputStream`. In other words, a large number of bytes are stored in an internal buffer and only written when the buffer fills up or is explicitly flushed. A `BufferedOutputStream` is more efficient than a regular `OutputStream` because the data is written to memory, rather than a disk or a network. Minimizing the number of write operations to a disk or the network minimizes the cumulative overhead for these operations.

You should wrap a `BufferedOutputStream` around any `OutputStream` whose `write()` operations may be time consuming or costly, such as a `FileOutputStream`.

Class Summary

```
public class java.io.BufferedOutputStream
             extends java.io.FilterOutputStream {
  // Variables
  protected byte[] buf;
  protected int count;
  // Constructors
  public BufferedOutputStream(OutputStream out);
  public BufferedOutputStream(OutputStream out, int size);
  // Instance Methods
  public synchronized void flush();
  public synchronized void write(int b);
  public synchronized void write(byte[] b, int off, int len);
}
```

Variables

buf

protected byte[] buf

Description

> The buffer that stores the data for the output stream.

count

protected int count

Description

> The current position in the buffer.

Constructors

BufferedOutputStream

public BufferedOutputStream(OutputStream out)

Parameters

`out`

The output stream to buffer.

Description

This constructor creates a `BufferedOutputStream` that acts on the specified `OutputStream`, using a buffer with the default size of 512 bytes.

public BufferedOutputStream(OutputStream out, int size)

Parameters

`out`

The output stream to buffer.

`size`

The size of buffer to use.

Description

This constructor creates a `BufferedOutputStream` that acts on the specified `OutputStream`, using a buffer that is `size` bytes long.

Instance Methods

flush

public synchronized void flush() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterOutputStream.flush()`

Description

This method writes the contents of the buffer to the underlying output stream. It is called automatically when the buffer fills up. You can also call it before the buffer is full. This is known as "flushing" the buffer. This method blocks until the underlying `write()` is complete.

write

public synchronized void write(int b) throws IOException

Parameters

`b`

The value to write.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterOutputStream.write(int)`

Description

This method places a byte containing the low-order eight bits of the given integer into the buffer. If the buffer is full, it is flushed, and the value `b` is placed in the newly empty buffer. If the buffer is flushed, this method blocks until `flush()` returns; otherwise this method returns immediately.

`public synchronized void write(byte b[], int off, int len) throws IOException`

Parameters

`b`

An array of bytes to write to the stream.

`off`

An offset into the byte array.

`len`

The number of bytes to write.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterOutputStream.write(byte[], int, int)`

Description

This method copies `len` bytes from `b`, starting at `off`, into the buffer. If there is enough space left in the buffer for the new data, it is copied into the buffer and the method returns immediately. Otherwise, the buffer is flushed, and the new data is written directly to the underlying stream. This is subtly different from the behavior of `write(int)`, which places new data in the buffer after a `flush()`.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`close()`	`FilterOutputStream`
`equals(Object)`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`toString()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`
`write(byte[])`	`FilterOutputStream`		

See Also

`FilterOutputStream`, `IOException`, `OutputStream`

BufferedReader

Name

BufferedReader

Synopsis

Class Name:

`java.io.BufferedReader`

Superclass:

`java.io.Reader`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

A BufferedReader object provides a more efficient way to read just a few characters at a time from a Reader. BufferedReader objects use a buffer to store input from an associated Reader. In other words, a large number of characters are read from the underlying reader and stored in an internal buffer. A BufferedReader is more efficient than a regular Reader because reading data from memory is faster than reading it from a disk or a network. All reading is done directly from the buffer; the disk or network needs to be accessed only occasionally to fill up the buffer.

You should wrap a BufferedReader around any Reader whose read() operations may be time consuming or costly, such as a FileReader or InputStreamReader.

BufferedReader provides a way to mark a position in the stream and subsequently reset the stream to that position, using mark() and reset().

A BufferedReader is similar to a BufferedInputStream, but it operates on a stream of Java characters instead of a byte stream, which makes it easier to support internationalization.

Class Summary

```
public class java.io.BufferedReader extends java.io.Reader {
  // Constructors
  public BufferedReader(Reader in);
  public BufferedReader(Reader in, int sz);
  // Instance Methods
  public void close();
  public void mark(int readAheadLimit);
  public boolean markSupported();
  public int read();
  public int read(char[] cbuf, int off, int len);
  public String readLine();
  public boolean ready();
  public void reset();
  public long skip(long n);
}
```

Constructors

BufferedReader

public BufferedReader(Reader in)

Parameters

`in`

The reader to buffer.

Description

This constructor creates a `BufferedReader` that buffers input from the given `Reader` using a buffer with the default size of 8192 characters.

public BufferedReader(Reader in, int sz)

Parameters

`in`

The reader to buffer.

`sz`

The size of buffer to use.

Throws

`IllegalArgumentException`

If the specified size is less than 0.

Description

This constructor creates a `BufferedReader` that buffers input from the given `Reader`, using a buffer of the given size.

Instance Methods

close

public void close() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`Reader.close()`

Description

This method closes this `BufferedReader` and its underlying `Reader`.

mark

public void mark(int readAheadLimit) throws IOException

Parameters

`readlimit`

The maximum number of bytes that can be read before the saved position becomes invalid.

Throws

`IOException`

If the stream is closed.

Overrides

`Reader.mark(int)`

Description

This method causes the `BufferedReader` to remember its current position. A subsequent call to `reset()` causes the object to return to that saved position, and thus reread a portion of the buffer.

markSupported

public boolean markSupported()

Returns

The `boolean` value `true`.

Overrides

`Reader.markSupported()`

Description

This method returns `true` to indicate that this class supports `mark()` and `reset()`.

read

public int read() throws IOException

Returns

The next character of data, or `-1` if the end of the stream is encountered.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`Reader.read()`

Description

This method returns the next character from the buffer. If all the characters in the buffer have been read, the buffer is filled from the underlying `Reader`, and the next character is returned. If the buffer does not need to be filled, this method returns immediately. If the buffer needs to be filled, this method blocks until data is available from the underlying `Reader`, the end of the stream is reached, or an exception is thrown.

public int read(char[] cbuf, int off, int len) throws IOException

Parameters

`cbuf`

An array of characters to be filled from the stream.

`off`

Offset into the character array.

`len`

Number of characters to read.

Returns

The actual number of characters read or `-1` if the end of the stream is encountered immediately.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`Reader.read(char[], int, int)`

Description

This method reads characters from the internal buffer into the given array `cbuf`, starting at index `off` and continuing for up to `len` bytes. If there are any characters in the buffer, this method returns immediately. Otherwise the buffer needs to be filled; this method blocks until the data is available from the underlying `InputStream`, the end of the stream is reached, or an exception is thrown.

readLine

public String readLine() throws IOException

Returns

A `String` containing the line just read, or `null` if the end of the stream has been reached.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method reads a line of text. Lines are terminated by `"\n"`, `"\r"`, or `"\r\n"`. The line terminators are not returned with the line string.

ready

public boolean ready() throws IOException

Returns

A `boolean` value that indicates whether the stream is ready to be read.

Throws

`IOException`

If the stream is closed.

Overrides

`Reader.ready()`

Description

If there is data in the buffer, or if the underlying stream is ready, this method returns `true`. The underlying stream is ready if the next `read()` is guaranteed to not block. Note that a return value of `false` does not guarantee that the next read operation will block.

reset

public void reset() throws IOException

Throws

`IOException`

If the reader is closed, `mark()` has not been called, or the saved position has been invalidated.

Overrides

`Reader.reset()`

Description

This method sets the position of the `BufferedReader` to a position that was saved by a previous call to `mark()`. Subsequent characters read from this `BufferedReader` will begin from the saved position and continue normally.

skip

public long skip(long n) throws IOException

Parameters

`n`

The number of characters to skip.

Returns

The actual number of characters skipped.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`Reader.skip()`

Description

This method skips `n` characters of input. If the new position of the stream is still within the data contained in the buffer, the method returns immediately. Otherwise the buffer is repeatedly filled until the requested position is available.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`read(char[])`	`Reader`
`toString()`	`Object`	`void wait()`	`Object`
`void wait(long)`	`Object`	`void wait(long, int)`	`Object`

See Also

`IllegalArgumentException`, `IOException`, `Reader`, `String`

BufferedWriter

Name

BufferedWriter

Synopsis

Class Name:

`java.io.BufferedWriter`

Superclass:

`java.io.Writer`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

A `BufferedWriter` object provides a more efficient way to write just a few characters at a time to a `Writer`. `BufferedWriter` objects use a buffer to store output for an associated `Writer`. In other words, a large number of characters are stored in an internal buffer and only written when the buffer fills up or is explicitly flushed. A `BufferedWriter` is more efficient than a regular `Writer` because the data is written to memory, rather than a disk or a network. Minimizing the number of write operations to a disk or the network minimizes the cumulative overhead for these operations.

You should wrap a `BufferedWriter` around any `Writer` whose `write()` operations may be time consuming or costly, such as a `FileWriter` or a `OutputStreamWriter`.

This class is very similar to `BufferedOutputStream`, but it operates on a stream of Java characters instead of a byte stream; this makes it easier to support internationalization.

Class Summary

```
public class java.io.BufferedWriter extends java.io.Writer {
  // Constructors
  public BufferedWriter(Writer out);
  public BufferedWriter(Writer out, int size);
  // Instance Methods
  public void close();
  public void flush();
  public void newLine();
  public void write(int c);
  public void write(char[] cbuf, int off, int len);
  public void write(String s, int off, int len);
}
```

Constructors

BufferedWriter

public BufferedWriter (Writer out)

Parameters

`out`

The output stream to buffer.

Description

This constructor creates a `BufferedWriter` that acts on the specified `Writer`, using a buffer with the default size of 8192 characters.

public BufferedWriter (Writer out, int size)

Parameters

`out`

The output stream to buffer.

`size`

The size of buffer to use.

Throws

`IllegalArgumentException`

If the specified size is less than 0.

Description

This constructor creates a `BufferedWriter` that acts on the specified `Writer`, using a buffer that is `size` bytes long.

Instance Methods

close

public void close() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`Writer.close()`

Description

This method closes this `BufferedWriter` and its underlying `Writer`.

flush

public void flush() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`Writer.flush()`

Description

This method writes the contents of the buffer to the underlying `Writer` and calls `flush()` on the underlying `Writer`. It is called automatically when the buffer fills up. You can also call it before the buffer is full. This is known as "flushing" the buffer. This method blocks until the underlying `write()` is complete.

newLine

public void newLine() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Description

> This method writes the newline character or characters to the stream. It uses `System.getProperty('line.separator')` to choose the newline appropriate for the run-time system. Calling this method is preferable to explicitly writing a newline character.

write

public void write(int c) throws IOException

Parameters

> `c`
>
> > The value to write.

Throws

> `IOException`
>
> > If any kind of I/O error occurs.

Overrides

> `Writer.write(int)`

Description

> This method places the low-order 16 bits of the specified value into the buffer. If the buffer is full, it is flushed, and the value `c` is placed in the newly empty buffer. If the buffer is flushed, this method blocks while the data is written; otherwise this method returns immediately.

`public void write(char[] cbuf, int off, int len) throws IOException`

Parameters

> `cbuf`
>
> > An array of characters to write.

`off`

An offset into the character array.

`len`

The number of characters to write.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`Writer.write(char[], int, int)`

Description

This method copies `len` characters from `cbuf`, starting at `off`, into the buffer. If there is enough space left in the buffer for the new data, it is copied into the buffer, and the method returns immediately. Otherwise, the buffer is filled and flushed repeatedly until all the new data has been copied into the buffer.

public void write(String s, int off, int len) throws IOException

Parameters

`s`

The string to be written.

`off`

An offset into the string.

`len`

The number of characters to write.

Throws

`IOException`

If an I/O error occurs.

Overrides

`Writer.write(String, int, int)`

Description

This method copies `len` characters from `s`, starting at `off`, into the buffer. If there is enough space left in the buffer for the new data, it is copied into the buffer and the method returns immediately. Otherwise, the buffer is filled and flushed repeatedly until all the new data has been copied into the buffer.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`	`write(char[])`	`Writer`
`write(String)`	`Writer`		

See Also

`IllegalArgumentException`, `IOException`, `String`, `Writer`

PREVIOUS
BufferedReader
HOME
BOOK INDEX
NEXT
ByteArrayInputStream

ByteArrayInputStream

Name

ByteArrayInputStream

Synopsis

Class Name:

    java.io.ByteArrayInputStream

Superclass:

    java.io.InputStream

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

A `ByteArrayInputStream` is a stream whose data comes from a byte array. None of the methods of this class throw an `IOException` because the data comes from an array instead of an actual I/O device. This class does not support the ability to mark a position in the stream. A call to `reset()`, however, does position the stream at the beginning of the byte array.

The position of the end of the stream depends on the constructor used. If the `ByteArrayInputStream(byte[] buf)` constructor is used, the end of the stream is the end of the byte array. If the `ByteArrayInputStream(byte[] buf, int offset, int length)` constructor is used, the end of the stream is reached at the index given by `offset+length`.

Class Summary

```
public class java.io.ByteArrayInputStream extends java.io.InputStream {
  // Variables
  protected byte[] buf;
  protected int count;
  protected int pos;
  // Constructors
  public ByteArrayInputStream(byte[] buf);
  public ByteArrayInputStream(byte[] buf, int offset, int length);
  // Instance Methods
  public synchronized int available();
  public synchronized int read();
  public synchronized int read(byte[] b, int off, int len);
  public synchronized void reset();
  public synchronized long skip(long n);
}
```

Variables

buf

protected byte[] buf

Description

> The buffer represented by this stream.

count

protected int count

Description

A placeholder that marks the end of the data this `ByteArrayInputStream` represents.

pos

protected int pos

Description

The current position in the buffer.

Constructors

ByteArrayInputStream

public ByteArrayInputStream(byte[] buf)

Parameters

`buf`

The stream source.

Description

This constructor creates a `ByteArrayInputStream` object that uses the given array of bytes as its data source. The data is not copied, so changes made to the array affect the data the `ByteArrayInputStream` returns.

public ByteArrayInputStream(byte[] buf, int offset, int length)

Parameters

`buf`

The stream source.

`offset`

An index into the buffer where the stream should begin.

`length`

The number of bytes to read.

Description

This constructor creates a `ByteArrayInputStream` that uses, as its data source, `length` bytes in a given array of bytes, starting at `offset` bytes from the beginning of the array. The data is not copied, so changes made to the array affect the data the `ByteArrayInputStream` returns.

Instance Methods

available

public synchronized int available()

Returns

The number of bytes remaining to be read in the array.

Overrides

`InputStream.available()`

Description

This method returns the number of bytes remaining to be read in the byte array.

read

public synchronized int read()

Returns

The next byte or `-1` if the end of the stream is encountered.

Overrides

`InputStream.read()`

Description

This method returns the next byte in the array.

public synchronized int read(byte[] b, int off, int len)

Parameters

`b`

An array to read bytes into.

`off`

An offset into `b`.

`len`

The number of bytes to read.

Returns

The number of bytes read or `-1` if the end of the stream is encountered.

Overrides

`InputStream.read(byte[], int, int)`

Description

This method copies up to `len` bytes from its internal byte array into the given array `b`, starting at index `off`.

reset

public synchronized void reset()

Overrides

`InputStream.reset()`

Description

This method resets the position of the input stream to the beginning of the byte array. If you specified an offset into the array, you might expect this method to reset the position to where you first started reading from the stream, but that is not the case.

skip

public synchronized long skip(long n)

Parameters

`n`

The number of bytes to skip.

Returns

The number of bytes skipped.

Overrides

`InputStream.skip()`

Description

This method skips `n` bytes of input. If you try to skip past the end of the array, the stream is positioned at the end of the array.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`close()`	`InputStream`
`equals (Object)`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`hashCode()`	`Object`
`mark(int)`	`InputStream`	`markSupported ()`	`InputStream`
`notify()`	`Object`	`notifyAll()`	`Object`
`read(byte[])`	`InputStream`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`InputStream`, `String`

PREVIOUS HOME NEXT
BufferedWriter BOOK INDEX ByteArrayOutputStream

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

ByteArrayOutputStream

Name

ByteArrayOutputStream

Synopsis

Class Name:

`java.io.ByteArrayOutputStream`

Superclass:

`java.io.FilterOutputStream`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

A `ByteArrayOutputStream` is a stream whose data is written to an internal byte array. None of the methods of this class throws an `IOException` because the data is written to an array instead of an actual I/O device.

The data for a `ByteArrayOutputStream` can be sent to another `OutputStream` using the `writeTo()` method. A copy of the array can be obtained using the `toCharArray()` method.

Class Summary

```
public class java.io.ByteArrayOutputStream extends java.io.OutputStream {
  // Variables
  protected byte[] buf;
  protected int count;

  // Constructors
  public ByteArrayOutputStream();
  public ByteArrayOutputStream(int size);
  // Instance Methods
  public synchronized void reset();
  public int size( );
  public synchronized byte[] toByteArray();
  public String toString();
  public String toString(int hibyte);            // Deprecated in 1.1
  public String toString(String enc);            // New in 1.1
  public synchronized void write(int b);
  public synchronized void write(byte[] b, int off, int len);
  public synchronized void writeTo(OutputStream out);
}
```

Variables

buf

protected byte[] buf

Description

The buffer that holds data for this stream.

count

protected int count

Description

A placeholder that marks the end of the data in the buffer.

Constructors

ByteArrayOutputStream

public ByteArrayOutputStream()

Description

This constructor creates a `ByteArrayOutputStream` with an internal buffer that has a default size of 32 bytes. The buffer grows automatically as data is written to the stream.

public ByteArrayOutputStream(int size)

Parameters

`size`

The initial buffer size.

Description

This constructor creates a `ByteArrayOutputStream` with an internal buffer that has a size of `size` bytes. The buffer grows automatically as data is written to the stream.

Instance Methods

reset

public synchronized void reset()

Description

This method discards the current contents of the buffer and resets the position of the stream to zero. Subsequent data is written starting at the beginning of the array.

size

public int size()

Description

> This method returns the number of bytes currently stored in this object's internal buffer. It is a count of the number of bytes that have been written to the stream.

toByteArray

public synchronized byte[] toByteArray()

Returns

> A copy of the data that has been written to this `ByteArrayOutputStream`.

Description

> This method copies the data in the internal array and returns a reference to the copy. The returned array is as long as the data that has been written to the stream, i.e., the same as `size()`.

toString

public String toString()

Returns

> A copy of the data that has been written to this `ByteArrayOutputStream`.

Overrides

> `Object.toString()`

Description

> This method returns a reference to a `String` object that contains a copy of the bytes currently stored in this object's internal buffer. The bytes are assumed to represent characters in the encoding that is customary for the native platform, so the bytes are converted to Unicode characters based on that assumption.

public String toString(int hibyte)

Availability

Deprecated as of JDK 1.1

Parameters

`hibyte`

A value to use as the high byte of each character.

Returns

A copy of the data that has been written to this `ByteArrayOutputStream`, where each character in the string has a high byte of `hibyte` and a low byte taken from the corresponding byte in the array.

Description

This method provides a way to convert from bytes to characters. As of 1.1, it is deprecated and replaced with `toString(String)`.

`public String toString(String enc) throws UnsupportedEncodingException`

Availability

New as of JDK 1.1

Parameters

`enc`

The encoding scheme to use.

Returns

A copy of the data that has been written to this `ByteArrayOutputStream`, converted from bytes to characters via the named encoding scheme `enc`.

Throws

`UnsupportedEncodingException`

The specified encoding is not supported.

Description

This method returns a Java `String` created from the byte array of this stream. The conversion is performed according to the encoding scheme `enc`.

write

public synchronized void write(int b)

Parameters

`b`

The value to write.

Overrides

`OutputStream.write(int)`

Description

This method writes the low-order 8 bits of the given value into the internal array. If the array is full, a larger array is allocated.

public synchronized void write(byte b[], int off, int len)

Parameters

`b`

The array to copy from.

`off`

Offset into the byte array.

`len`

Number of bytes to write.

Overrides

`OutputStream.write(byte[], int, int)`

Description

This method copies `len` bytes to this object's internal array from `b`, starting `oset` elements from the beginning of the supplied array `b`. If the internal array is full, a larger array is allocated.

writeTo

```
public synchronized void writeTo(OutputStream out) throws IOException
```

Parameters

`out`

The destination stream.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method writes the contents of this object's internal buffer to the given `OutputStream`. All the data that has been written to this `ByteArrayOutputStream` is written to `out`.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`close()`	`OutputStream`
`equals(Object)`	`Object`	`finalize()`	`Object`
`flush()`	`OutputStream`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`

`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`
`write(byte[])`	`OutputStream`		

See Also

`IOException`, `OutputStream`, `String`, `UnsupportedEncodingException`

PREVIOUS HOME NEXT
ByteArrayInputStream BOOK INDEX CharArrayReader

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

PREVIOUS

NEXT

CharArrayReader

Name

CharArrayReader

Synopsis

Class Name:

```
java.io.CharArrayReader
```

Superclass:

```
java.io.Reader
```

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `CharArrayReader` class represents a stream whose data comes from a character array. This class is similar to `ByteArrayInputStream`, but it deals with a Java character stream rather than a byte stream. Furthermore, this class supports marking a position in the stream, which `ByteArrayInputStream` does not.

The position of the end of the stream depends on the constructor used. If the `CharArrayReader(char[] buf)` constructor is used, the end of the stream is the end of the character array. If the `CharArrayReader(char[] buf, int offset, int length)` constructor is used, the end of the stream is reached at the index given by `offset+length`.

Class Summary

```
public class java.io.CharArrayReader extends java.io.Reader {
  // Variables
  protected char[] buf;
  protected int count;
  protected int markedPos;
  protected int pos;
  // Constructors
  public CharArrayReader(char[] buf);
  public CharArrayReader(char[] buf, int offset, int length);
  // Instance Methods
  public void close();
  public void mark(int readAheadLimit);
  public boolean markSupported();
  public int read();
  public int read(char[] b, int off, int len);
  public boolean ready();
  public void reset();
  public long skip(long n);
}
```

Variables

buf

protected char[] buf

Description

The buffer represented by this reader.

count

protected int count

Description

The size of the buffer, or in other words, the length of the array.

markedPos

protected int markedPos

Description

The buffer position when `mark()` was called. If `mark()` has not been called, this variable is 0.

pos

protected int pos

Description

The current position in the buffer.

Constructors

CharArrayReader

public CharArrayReader(char[] buf)

Parameters

`buf`

The reader source.

Description

This constructor creates a `CharArrayReader` object that uses the given array of characters as its data source. The data is not copied, so changes made to the array affect the data that the `CharArrayReader` returns.

public CharArrayReader(char[] buf, int offset, int length)

Parameters

`buf`

The reader source.

`offset`

An offset into the array.

`length`

The number of bytes to read.

Description

This constructor creates a `CharArrayReader` that uses, as its data source, `length` characters in a given array of bytes, starting at `offset` characters from the beginning of the array. The data is not copied, so changes made to the array affect the data that the `CharArrayReader` returns.

Instance Methods

close

public void close()

Overrides

`Reader.close()`

Description

This method closes the reader by removing the link between this `CharArrayReader` and the array it was created with.

mark

public void mark(int readAheadLimit) throws IOException

Parameters

`readAheadLimit`

The maximum number of characters that can be read before the saved position becomes invalid.

Throws

`IOException`

If the stream is closed or any other kind of I/O error occurs.

Overrides

`Reader.mark(int)`

Description

This method causes the `CharArrayReader` to remember its current position. A subsequent call to `reset()` causes the object to return to that saved position, and thus reread a portion of the buffer. Because the data for this stream comes from a `char` array, there is no limit on reading ahead, so `readAheadLimit` is ignored.

markSupported

public boolean markSupported()

Returns

The `boolean` value `true`.

Overrides

`Reader.markSupported()`

Description

This method returns `true` to indicate that this class supports `mark()` and `reset()`.

read

public int read() throws IOException

Returns

The next character or `-1` if the end of the stream is encountered.

Throws

`IOException`

If the stream is closed or any other kind of I/O error occurs.

Overrides

`Reader.read()`

Description

This method returns the next character in the array.

public int read(char[] b, int off, int len) throws IOException

Parameters

`b`

An array of characters to be filled from the stream.

`off`

An offset into the character array.

`len`

The number of characters to read.

Returns

The actual number of characters read or `-1` if the end of the stream is encountered immediately.

Throws

`IOException`

If the stream is closed or any other kind of I/O error occurs.

Overrides

`Reader.read(char[], int, int)`

Description

This method copies up to `len` characters from its internal array into the given array `b`, starting at index `off`.

ready

public boolean ready() throws IOException

Returns

A `boolean` value that indicates whether the stream is ready to be read.

Throws

`IOException`

If the stream is closed or any other kind of I/O error occurs.

Overrides

`Reader.ready()`

Description

If there is any data left to be read from the character array, this method returns `true`.

reset

public void reset() throws IOException

Throws

`IOException`

If the stream is closed or any other kind of I/O error occurs.

Overrides

`Reader.reset()`

Description

This method resets the position of the `CharArrayReader` to the position that was saved by calling the `mark()` method. If `mark()` has not been called, the `CharArrayReader` is reset to read from the beginning of the array.

skip

public long skip(long n) throws IOException

Parameters

`n`

The number of characters to skip.

Returns

The actual number of characters skipped.

Throws

`IOException`

If the stream is closed or any other kind of I/O error occurs.

Overrides

`Reader.skip()`

Description

This method skips n characters of input. If you try to skip past the end of the array, the stream is positioned at the end of the array.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals (Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`read(char[])`	`Reader`
`toString()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`IOException`, `Reader`, `String`

PREVIOUS ByteArrayOutputStream
HOME
BOOK INDEX
NEXT CharArrayWriter

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

CharArrayWriter

Name

CharArrayWriter

Synopsis

Class Name:

`java.io.CharArrayWriter`

Superclass:

`java.io.Writer`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `CharArrayWriter` class represents a stream whose data is written to an internal character array. This class is similar to `ByteArrayOutputStream`, but it operates on an array of Java characters instead of a byte array.

The data from a `CharArrayWriter` can be sent to another `Writer` using the `writeTo()` method. A copy of the array can be obtained using the `toCharArray()` method.

Class Summary

```
public class java.io.CharArrayWriter extends java.io.Writer {
  // Variables
  protected char[] buf;
  protected int count;
  // Constructors
  public CharArrayWriter();
  public CharArrayWriter(int initialSize);
  // Instance Methods
  public void close();
  public void flush();
  public void reset();
  public int size();
  public char[] toCharArray();
  public String toString();
  public void write(int c);
  public void write(char[] c, int off, int len);
  public void write(String str, int off, int len);
  public void writeTo(Writer out);
}
```

Variables

buf

protected char[] buf

Description

The buffer that holds data for this stream.

count

protected int count

Description

A placeholder that marks the end of the data in the buffer.

Constructors

CharArrayWriter

public CharArrayWriter()

Description

This constructor creates a `CharArrayWriter` with an internal buffer that has a default size of 32 characters. The buffer grows automatically as data is written to the stream.

public CharArrayWriter(int initialSize)

Parameters

`initialSize`

The initial buffer size.

Description

This constructor creates a `CharArrayWriter` with an internal buffer that has a size of `initialSize` characters. The buffer grows automatically as data is written to the stream.

Instance Methods

close

public void close()

Overrides

> `Writer.close()`

Description

> This method does nothing. For most subclasses of `Writer`, this method releases any system resources that are associated with the `Writer` object. However, the `CharArrayWriter`'s internal array may be needed for subsequent calls to `toCharArray()` or `writeTo()`. For this reason, `close()` does nothing, and the internal array is not released until the `CharArrayWriter` is garbage collected.

flush

public void flush()

Overrides

> `Writer.flush()`

Description

> This method does nothing. The `CharArrayWriter` writes data directly into its internal array; thus it is never necessary to flush the stream.

reset

public void reset()

Description

> This method discards the current contents of the buffer and resets the position of the stream to zero. Subsequent data is written starting at the beginning of the array.

size

public int size()

Description

> This method returns the number of characters currently stored in this object's internal buffer. It is a count of the number of characters that have been written to the stream.

toCharArray

public char[] toCharArray()

Returns

> A copy of the data that has been written to this `CharArrayWriter` in the form of a `char` array.

Description

> This method copies the data in the internal array and returns a reference to the copy. The returned array is as long as the data that has been written to the stream, i.e., the same as `size()`.

toString

public String toString()

Returns

> A copy of the data that has been written to this `CharArrayWriter` in the form of a `String`.

Overrides

> `Object.toString()`

Description

> This method returns a reference to a `String` object created from the characters stored in this object's internal buffer.

write

public void write(int c)

Parameters

`c`

The value to write.

Overrides

`Writer.write(int)`

Description

This method writes the low-order 16 bits of the given value into the internal array. If the array is full, a larger array is allocated.

public void write(char[] c, int off, int len)

Parameters

`c`

An array of characters to write to the stream.

`off`

An offset into the character array.

`len`

The number of characters to write.

Overrides

`Writer.write(char[], int, int)`

Description

This method copies `len` characters to this object's internal array from `c`, starting `off` elements from the beginning of the array. If the internal array is full, a larger array is allocated.

public void write(String str, int off, int len)

Parameters

`str`

A `String` to write to the stream.

`off`

An offset into the string.

`len`

The number of characters to write.

Overrides

`Writer.write(String, int, int)`

Description

This method copies `len` characters to this object's internal array from `str`, starting `off` characters from the beginning of the given string. If the internal array is full, a larger array is allocated.

writeTo

public void writeTo(Writer out) throws IOException

Parameters

`out`

The destination stream.

Throws

`IOException`

If any kind of I/O error occurs.

Description

> This method writes the contents of this object's internal buffer to the given `Writer`. All the data that has been written to this `CharArrayWriter` is written to `out`.

Inherited Methods

Method	**Inherited From**	**Method**	**Inherited From**
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`
`write(char[])`	`Writer`	`write(String)`	`Writer`

See Also

`IOException`, `String`, `Writer`

CharConversionException

Name

CharConversionException

Synopsis

Class Name:

```
java.io.CharConversionException
```

Superclass:

```
java.io.IOException
```

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

A `CharConversionException` object is thrown when a problem occurs in converting a character to a byte.

Class Summary

```
public class java.io.CharConversionException extends java.io.IOException {
  // Constructors
  public CharConversionException();
  public CharConversionException(String s);
}
```

Constructors

CharConverionException

public CharConversionException()

Description

This constructor creates a `CharConversionException` with no detail message.

public CharConversionException(String s)

Parameters

`s`

The detail message.

Description

This constructor creates a `CharConversionException` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`

`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Exception`, `IOException`, `Throwable`

PREVIOUS | HOME | NEXT
CharArrayWriter | BOOK INDEX | DataInput

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

File

Name

File

Synopsis

Class Name:

> `java.io.File`

Superclass:

> `java.lang.Object`

Immediate Subclasses:

> None

Interfaces Implemented:

> None

Availability:

> JDK 1.0 or later

Description

The `File` class provides methods to obtain information about files and directories. A `File` object encapsulates the name of a file or a directory. A `File` object can list the files in a directory, check the existence and type of a file, create new directories, and rename and delete files, among other things. However, the `File` class does not handle I/O to files. Actual reading and writing is accomplished using `RandomAccessFile`, `FileReader`, `FileWriter`, `FileInputStream`, and `FileOutputStream` objects.

The `File` class also defines some constants for the platform-specific directory and path separator characters. If you want to avoid putting system-dependent path information in your program, you may want to reference all files relative to the directory in which your program is running (i.e., the current directory). Alternatively, you can use `java.awt.FileDialog` to prompt the user for system-dependent paths.

Many of the methods in `File` throw a `SecurityException` if the application does not have sufficient privileges for the requested operation. This happens in two steps. First, `System.getSecurityManager()` is called. If a `SecurityManager` has been installed, it is queried for the appropriate permission. For example, `File.canRead()` calls `SecurityManager.canRead()`. If the application does not have permission to read the specified file, the `SecurityManager` throws a `SecurityException`, which in turn is thrown by `File.canRead()`.

Class Summary

```
public class java.io.File extends java.lang.Object
                          implements java.io.Serializable {
  // Constants
  public final static String pathSeparator;
  public final static char pathSeparatorChar;
  public final static String separator;
  public final static char separatorChar;
  // Constructors
  public File(String path);
  public File(String path, String name);
  public File(File dir, String name);
  // Instance Methods
  public boolean canRead();
  public boolean canWrite();
  public boolean delete();
  public boolean equals(Object obj);
  public boolean exists();
  public String getAbsolutePath();
  public String getCanonicalPath();              // New in 1.1
  public String getName();
  public String getParent();
  public String getPath();
  public int hashCode();
  public native boolean isAbsolute();
  public boolean isDirectory();
  public boolean isFile();
  public long lastModified();
  public long length();
  public String[] list();
  public String[] list(FilenameFilter filter);
  public boolean mkdir();
  public boolean mkdirs();
  public boolean renameTo(File dest);
  public String toString();
}
```

Constants

pathSeparator

public final static String pathSeparator

Description

> This string holds the value of `System.getProperty('path.separator')`. It contains the character that separates paths in a path list. Usually it is "`:`" or "`;`".

pathSeparatorChar

public final static char pathSeparatorChar

Description

> This variable holds the first (and only) character in `pathSeparator`.

separator

public final static String separator

Description

This string holds the value of `System.getProperty('file.separator')`. It contains the character that separates directory and filenames in a path string. Usually it is "`/`" or "`\`".

separatorChar

public final static char separatorChar

Description

This variable holds the first (and only) character in `separator`.

Constructors

File

public File(String path)

Parameters

`path`

A full pathname (i.e., a directory and filename).

Description

This constructor creates a `File` object that represents the file specified by `path`.

public File(String path, String name)

Parameters

`path`

A directory path.

name

A filename.

Description

This constructor creates a `File` object that represents the file with the specified `name` in the directory described by `path`. In other words, the full pathname is the directory, followed by the separator character, followed by the filename.

If `path` is `null`, the constructor creates a `File` that represents the file with the specified `name` in the current directory. The current directory is the directory in which the program is running.

public File(File dir, String name)

Parameters

dir

A `File` object that represents a directory.

name

A filename.

Description

This constructor creates a `File` object that represents the file with the specified `name` in the directory described by the `File` object `dir`. In other words, the full pathname is the directory represented by `dir`, followed by the separator character, followed by the filename.

If `dir` is `null`, the constructor creates a `File` that represents the file with the specified `name` in the current directory. The current directory is the directory in which the program is running.

Instance Methods

canRead

public boolean canRead()

Returns

A `boolean` value that indicates if the file is readable.

Throws

`SecurityException`

If the application does not have permission to read the `File`.

Description

This method returns `true` if `File` corresponds to an existing, readable file or directory. Otherwise it returns `false`.

canWrite

public boolean canWrite()

Returns

A `boolean` value that indicates if the file is writable.

Throws

`SecurityException`

If the application does not have permission to write to the `File`.

Description

This method returns `true` if `File` corresponds to an existing, writable file or directory. Otherwise it returns `false`.

delete

public boolean delete()

Returns

`true` if the file is deleted; otherwise `false`.

Throws

`SecurityException`

If the application does not have permission to delete the file.

Description

This method attempts to delete the file or directory associated with this `File` object. A directory is only deleted if it is empty.

equals

public boolean equals(Object obj)

Parameters

`obj`

The `Object` to be compared.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `obj` is an instance of `File` that encapsulates the same pathname as this object.

exists

public boolean exists()

Returns

`true` if the file or directory exists; `false` otherwise.

Throws

`SecurityException`

If the application does not have permission to read the `File`.

Description

This method returns `true` if this `File` corresponds to an existing file or directory.

getAbsolutePath

public String getAbsolutePath()

Returns

A `String` that contains the absolute pathname.

Description

This method returns the absolute pathname of the file or directory associated with this `File`.

getCanonicalPath

public String getCanonicalPath() throws IOException

Availability

New as of JDK 1.1

Returns

A `String` that contains the canonical, or exact, pathname.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method returns the canonical pathname of the file or directory associated with this `File`.

getName

public String getName()

Returns

A `String` that contains the filename.

Description

This method returns the filename associated with this `File`. The string returned does not include the name of the directory.

getParent

public String getParent()

Returns

A `String` that contains the parent directory of the file, or `null` if it does not exist.

Description

This method returns the name of the parent directory of the file or directory associated with this `File`. The algorithm used returns everything in the pathname before the last separator character.

getPath

public String getPath()

Returns

A `String` that contains the pathname of the file.

Description

This method returns the full pathname associated with this `File`.

hashCode

public int hashCode()

Returns

A hashcode value for this file.

Overrides

`Object.hashCode()`

Description

This method returns a hashcode based on the pathname associated with this `File`.

isAbsolute

public native boolean isAbsolute()

Returns

`true` if the `File` represents an absolute path; `false` otherwise.

Description

This method indicates if the `File` represents an absolute path; what constitutes an absolute path is system-dependent.

isDirectory

public boolean isDirectory()

Returns

true if the File represents a directory; false otherwise.

Throws

SecurityException

If the application does not have permission to read the File.

Description

This method returns true if this File corresponds to a directory.

isFile

public boolean isFile()

Returns

true if the File represents a normal file; false otherwise.

Throws

SecurityException

If the application does not have permission to read the File.

Description

This method returns true if this File corresponds to a normal file, as opposed to an alternative, such as a directory, a named pipe, or a device.

lastModified

public long lastModified()

Returns

The time the file was last modified, or 0L if the file does not exist.

Throws

`SecurityException`

If the application does not have permission to read the `File`.

Description

This method returns the modification time of the file or directory that corresponds to this `File`. The format of the time returned is useful for comparing modification times; it's not meant to be used for other purposes.

length

public long length()

Returns

The file length, in bytes, or `0L` if the file does not exist.

Throws

`SecurityException`

If the application does not have permission to read the `File`.

Description

This method returns the length of the file or directory that corresponds to this `File`.

list

public String[] list()

Returns

An array of the names of the files and directories contained by this `File`, or `null` if this `File` is not a directory.

Throws

`SecurityException`

If the application does not have permission to read the `File`.

Description

This method returns the contents of a directory. The current directory and the parent directory are not included in the list.

public String[] list(FilenameFilter filter)

Parameters

`filter`

A filter to use.

Returns

An array of the names of the files and directories contained by this `File` and filtered by `filter`, or `null` if this `File` is not a directory.

Throws

`SecurityException`

If the application does not have permission to read the `File`.

Description

This method returns of the contents of a directory as selected by the given `FilenameFilter` object. Specifically, a name is included if the `FilenameFilter` object's `accept()` method returns `true` for that name.

If `filter` is `null`, this method is equivalent to, but slower than, `list()`.

mkdir

public boolean mkdir()

Returns

`true` if the directory is created; `false` otherwise.

Throws

`SecurityException`

If the application does not have permission to write to the `File`.

Description

This method creates a directory with the pathname specified by this `File`.

mkdirs

public boolean mkdirs()

Returns

`true` if the directory is created; `false` otherwise.

Throws

`SecurityException`

If the application does not have permission to write to the `File`.

Description

This method creates a directory with the pathname specified by this `File`. The method also creates all the parent directories if necessary.

renameTo

public boolean renameTo(File dest)

Parameters

`dest`

A `File` that specifies the new name.

Returns

`true` if the name is changed; `false` otherwise.

Throws

`SecurityException`

If the application does not have permission to write to this `File` or the file represented by `dest`.

Description

This method changes the pathname of this `File` to the pathname specified by `dest`.

toString

public String toString()

Returns

A `String` that contains the pathname of this `File`.

Overrides

`Object.toString()`

Description

This method returns a string representation of this `File` object.

Inherited Methods

Method	**Inherited From**	**Method**	**Inherited From**
`clone()`	`Object`	`finalize()`	`Object`

`getClass()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`FileInputStream`, `FilenameFilter`, `FileOutputStream`, `FileReader`, `FileWriter`, `IOException`, `SecurityException`

PREVIOUS | HOME | NEXT
Externalizable | BOOK INDEX | FileDescriptor

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

FileInputStream

Name

FileInputStream

Synopsis

Class Name:

```
java.io.FileInputStream
```

Superclass:

```
java.io.InputStream
```

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The FileInputStream class represents a byte stream that reads data from a file. The file can be specified using a FileDescriptor, a File object, or a String that represents a pathname. All of the constructors can throw a SecurityException if the application does not have permission to read from the specified file.

FileInputStream provides a low-level interface for reading data from a file. You should wrap a FileInputStream with a DataInputStream if you need a higher-level interface that can handle reading strings and binary data. You should also think about wrapping a FileInputStream with a BufferedInputStream to increase reading efficiency.

Data must be read sequentially from a FileInputStream; you can skip forward, but you cannot move back. If you need random access to file data, use the RandomAccessFile class instead.

Class Summary

```
public class java.io.FileInputStream extends java.io.InputStream {
  // Constructors
  public FileInputStream(String name);
  public FileInputStream(File file);
  public FileInputStream(FileDescriptor fdObj);
  // Public Instance Methods
  public native int available();
  public native void close();
  public final FileDescriptor getFD();
  public native int read();
  public int read(byte[] b);
  public int read(byte[] b, int off, int len);
  public native long skip(long n);
  // Protected Instance Methods
  protected void finalize();
}
```

Constructors

FileInputStream

public FileInputStream(String name) throws FileNotFoundException

Parameters

`name`

A `String` that contains the pathname of the file to be accessed. The path must conform to the requirements of the native operating system.

Throws

`FileNotFoundException`

If the named file cannot be found.

`SecurityException`

If the application does not have permission to read the named file.

Description

This constructor creates a `FileInputStream` that gets its input from the file named by the specified `String`.

public FileInputStream(File file) throws FileNotFoundException

Parameters

`file`

The `File` to use as input.

Throws

`FileNotFoundException`

If the named file cannot be found.

`SecurityException`

If the application does not have permission to read the named file.

Description

This constructor creates a `FileInputStream` that gets its input from the file represented by the specified `File`.

public FileInputStream(FileDescriptor fdObj)

Parameters

`fdObj`

The `FileDescriptor` of the file to use as input.

Throws

`SecurityException`

If the application does not have permission to read the specified file.

`NullPointerException`

If `FileDescriptor` is `null`.

Description

This constructor creates a `FileInputStream` that gets its input from the file identified by the given `FileDescriptor`.

Public Instance Methods

available

public native int available() throws IOException

Returns

The number of bytes that can be read from the file without blocking.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`InputStream.available()`

Description

This method returns the number of available bytes of data. Initially, this is the length of the file.

close

public native void close() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`InputStream.close()`

Description

This method closes this file input stream and releases any resources used by it.

getFD

public final FileDescriptor getFD() throws IOException

Returns

The file descriptor for the file that supplies data for this stream.

Throws

`IOException`

If there is no `FileDescriptor` associated with this object.

Description

This method returns the file descriptor associated with the data source of this `FileInputStream`.

read

public native int read() throws IOException

Returns

The next byte of data or `-1` if the end of the stream is encountered.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`InputStream.read()`

Description

This method reads the next byte of data from the file. The method blocks if no input is available.

public int read(byte[] b) throws IOException

Parameters

`b`

An array of bytes to be filled from the stream.

Returns

The actual number of bytes read or `-1` if the end of the stream is encountered immediately.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`InputStream.read(byte[])`

Description

This method reads data into the given array. The method fills the array if enough bytes are available. The method blocks until some input is available.

public int read(byte[] b, int off, int len) throws IOException

Parameters

`b`

An array of bytes to be filled from the stream.

`off`

An offset into the byte array.

`len`

The number of bytes to read.

Returns

The actual number of bytes read or `-1` if the end of the stream is encountered immediately.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`InputStream.read(byte[], int, int)`

Description

This method reads `len` bytes of data into the given array, starting at element `off`. The method blocks until some input is available.

skip

public native long skip(long n) throws IOException

Parameters

`n`

The number of bytes to skip.

Returns

The actual number of bytes skipped.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterInputStream.skip()`

Description

This method skips `n` bytes of input in the stream.

Protected Instance Methods

finalize

protected void finalize() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`Object.finalize()`

Description

This method is called when the `FileInputStream` is garbage collected to ensure that `close()` is called. If the stream has a valid file descriptor, the `close()` method is called to free the system resources used by this stream.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`getClass()`	`Object`	`hashCode()`	`Object`
`mark(int)`	`InputStream`	`markSupported()`	`InputStream`
`notify()`	`Object`	`notifyAll()`	`Object`
`reset()`	`InputStream`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`BufferedInputStream`, `DataInputStream`, `File`, `FileDescriptor`, `FileNotFoundException`, `InputStream`, `IOException`, `NullPointerException`, `RandomAccessFile`, `SecurityException`

PREVIOUS

NEXT

FilenameFilter

Name

FilenameFilter

Synopsis

Interface Name:

`java.io.FilenameFilter`

Super-interface:

None

Immediate Sub-interfaces:

None

Implemented by:

None

Availability:

JDK 1.0 or later

Description

The FilenameFilter interface is implemented by a class that wants to filter the filenames that should be included in a list of filenames. For example, the list() method of the File class can take a FilenameFilter object to filter the filenames that are listed. The java.awt.FileDialog class also uses a FilenameFilter to limit the choices that are presented to the user.

Interface Declaration

```
public abstract interface java.io.FilenameFilter {
  // Methods
  public abstract boolean accept(File dir, String name);
}
```

Methods

accept

public abstract boolean accept(File dir, String name)

Parameters

dir

The directory that contains the file.

name

The name of the file.

Returns

true if the file should be shown; false otherwise.

Description

This method returns a boolean value that indicates whether or not a file should be included in a list of filenames. The method should return true if a file should be included; otherwise it should return false. A simple filter might return true for filenames with a certain extension, like

.java. A more complex filter could check the directory name, the file's readability, and last modification time, for example.

See Also

File

FileNotFoundException

Name

FileNotFoundException

Synopsis

Class Name:

`java.io.FileNotFoundException`

Superclass:

`java.io.IOException`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

A `FileNotFoundException` is thrown when a specified file cannot be located.

Class Summary

```
public class java.io.FileNotFoundException extends java.io.IOException {
  // Constructors
  public FileNotFoundException();
  public FileNotFoundException(String s);
}
```

Constructors

FileNotFoundException

public FileNotFoundException()

Description

> This constructor creates a `FileNotFoundException` with no detail message.

public FileNotFoundException(String s)

Parameters

> `s`
>
> > The detail message.

Description

> This constructor creates a `FileNotFoundException` with the specified detail message.

Inherited Methods

Method	**Inherited From**	**Method**	**Inherited From**
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`

`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Exception`, `IOException`, `Throwable`

PREVIOUS HOME NEXT

FilenameFilter BOOK INDEX FileOutputStream

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

FileOutputStream

Name

FileOutputStream

Synopsis

Class Name:

```
java.io.FileOutputStream
```

Superclass:

```
java.io.OutputStream
```

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The FileOutputStream class represents a byte stream that writes data to a file. The file can be specified using a FileDescriptor, a File object, or a String that represents a pathname. All of the constructors can throw a SecurityException if the application does not have permission to write to the specified file.

FileOutputStream provides a low-level interface for writing data to a file. Wrap a FileOutputStream with a DataOutputStream or a PrintStream if you need a higher-level interface that can handle writing strings and binary data. You should also think about wrapping a FileOutputStream with a BufferedOutputStream to increase writing efficiency.

Data must be written sequentially to a FileOutputStream; you can either overwrite existing data or append data to the end of the file. If you need random access to file data, use the RandomAccessFile class instead.

Class Summary

```
public class java.io.FileOutputStream extends java.io.OutputStream {
  // Constructors
  public FileOutputStream(String name);
  public FileOutputStream(String name, boolean append);     // New in 1.1
  public FileOutputStream(File file);
  public FileOutputStream(FileDescriptor fdObj);
  // Public Instance Methods
  public native void close();
  public final FileDescriptor getFD();
  public native void write(int b);
  public void write(byte[] b);
  public void write(byte[] b, int off, int len);
  // Protected Instance Methods
  protected void finalize();
}
```

Constructors

FileOutputStream

public FileOutputStream(String name) throws IOException

Parameters

name

A `String` that contains the pathname of the file to be used for output. The path must conform to the requirements of the native operating system.

Throws

`FileNotFoundException`

If the named file cannot be found.

`SecurityException`

If the application does not have permission to write to the named file.

Description

This constructor creates a `FileOutputStream` that sends its output to the file named by the specified `String`.

`public FileOutputStream(String name, boolean append) throws IOException`

Availability

New as of JDK 1.1

Parameters

`name`

A `String` that contains the pathname of the file to be used for output. The path must conform to the requirements of the native operating system.

`append`

Specifies whether or not data is appended to the output stream.

Throws

`FileNotFoundException`

If the named file cannot be found.

`SecurityException`

If the application does not have permission to write to the named file.

Description

This constructor creates a `FileOutputStream` that sends its output to the named file. If `append` is `true`, the stream is positioned at the end of the file, and data is appended to the end of the file. Otherwise, if `append` is `false`, the stream is positioned at the beginning of the file, and any previous data is overwritten.

public FileOutputStream(File file) throws IOException

Parameters

`file`

The `File` to use as output.

Throws

`FileNotFoundException`

If the named file cannot be found.

`SecurityException`

If the application does not have permission to write to the named file.

Description

This constructor creates a `FileOutputStream` that sends its output to the file represented by the specified `File`.

public FileOutputStream(FileDescriptor fdObj)

Parameters

`fdObj`

The `FileDescriptor` of the file to use as output.

Throws

`SecurityException`

If the application does not have permission to write to the specified file.

`NullPointerException`

If `FileDescriptor` is `null`.

Description

This constructor creates a `FileOutputStream` that sends its output to the file identified by the given `FileDescriptor`.

Public Instance Methods

close

public native void close() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`OutputStream.close()`

Description

This method closes this file output stream and releases any resources used by it.

getFD

public final FileDescriptor getFD() throws IOException

Throws

`IOException`

If there is no `FileDescriptor` associated with this object.

Description

This method returns the file descriptor associated with the data source of this `FileOutputStream`.

write

public native void write(int b) throws IOException

Parameters

`b`

The value to write to the stream.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`OutputStream.write(int)`

Description

This method writes a byte containing the low-order eight bits of the given value to the output stream.

public void write(byte[] b) throws IOException

Parameters

`b`

An array of bytes to write to the stream.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`OutputStream.write(byte[])`

Description

This method writes the entire contents of the given array to the output stream.

public void write(byte[] b, int off, int len) throws IOException

Parameters

`b`

An array of bytes to write to the stream.

`off`

An offset into the byte array.

`len`

The number of bytes to write.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`OutputStream.write(byte[], int, int)`

Description

This method writes `len` bytes from the given array, starting at element `off`, to the output stream.

Protected Instance Methods

finalize

protected void finalize() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method is called when the `FileOutputStream` is garbage-collected to ensure that `close()` is called. If the stream has a valid file descriptor, the `close()` method is called to free the system resources used by this stream.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`flush()`	`OutputStream`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`BufferedOutputStream`, `DataOutputStream`, `File`, `FileDescriptor`, `FileNotFoundException`, `IOException`, `NullPointerException`, `OutputStream`, `PrintStream`, `RandomAccessFile`, `SecurityException`

PREVIOUS HOME NEXT
FileNotFoundException BOOK INDEX FileReader

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

FilterInputStream

Name

FilterInputStream

Synopsis

Class Name:

`java.io.FilterInputStream`

Superclass:

`java.io.InputStream`

Immediate Subclasses:

`java.io.BufferedInputStream`,

`java.io.DataInputStream`,

`java.io.LineNumberInputStream`,

`java.io.PushbackInputStream`,

`java.util.zip.CheckedInputStream`,

java.util.zip.InflaterInputStream

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The FilterInputStream class is the superclass of all of the input stream classes that filter input. Each of the subclasses of FilterInputStream works by wrapping an existing input stream, called the *underlying input stream*, and providing additional functionality. The methods of FilterInputStream simply override the methods of InputStream with versions that call the corresponding methods of the underlying stream.

FilterInputStream cannot be instantiated directly; it must be subclassed. An instance of one of the subclasses of FilterInputStream is constructed with another InputStream object. The methods of a subclass of FilterInputStream should override some methods in order to extend their behavior or provide some sort of filtering.

Class Summary

```
public class java.io.FilterInputStream extends java.io.InputStream {
  // Variables
  protected InputStream in;
  // Constructors
  protected FilterInputStream(InputStream in);
  // Instance Methods
  public int available();
  public void close();
  public synchronized void mark(int readlimit);
  public boolean markSupported();
  public int read();
  public int read(byte[] b);
  public int read(byte[] b, int off, int len);
  public synchronized void reset();
  public long skip(long n);
}
```

Variables

in

protected InputStream in

Description

The underlying stream that this `FilterInputStream` wraps or filters.

Constructors

FilterInputStream

protected FilterInputStream(InputStream in)

Parameters

`in`

The input stream to filter.

Description

This constructor creates a `FilterInputStream` that gets its data from `in`.

Instance Methods

available

public int available() throws IOException

Returns

The number of bytes that can be read without blocking.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`InputStream.available()`

Description

This method calls the `available()` method of the underlying stream and returns the result.

close

public void close() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`InputStream.close()`

Description

This method calls the `close()` method of the underlying stream, which releases any system resources associated with this object.

mark

public synchronized void mark(int readlimit)

Parameters

`readlimit`

The maximum number of bytes that can be read before the saved position becomes invalid.

Overrides

`InputStream.mark()`

Description

This method calls the `mark()` method of the underlying stream. If the underlying stream supports `mark()` and `reset()`, this method causes the `FilterInputStream` to remember its current position. A subsequent call to `reset()` causes the object to return to that saved position, and thus re-read a portion of the input.

markSupported

public boolean markSupported()

Returns

`true` if this stream supports `mark()` and `reset()`; `false` otherwise.

Overrides

`InputStream.markSupported()`

Description

This method calls the `markSupported()` method of the underlying stream and returns the result.

read

public int read() throws IOException

Returns

The next byte of data or `-1` if the end of the stream is encountered.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`InputStream.read()`

Description

This method calls the `read()` method of the underlying stream and returns the result. This method blocks until some data is available, the end of the stream is detected, or an exception is thrown.

public int read(byte[] b) throws IOException

Parameters

`b`

An array of bytes to be filled from the stream.

Returns

The actual number of bytes read or `-1` if the end of the stream is encountered immediately.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`InputStream.read(byte[])`

Description

This method reads bytes of input to fill the given array. It does this by calling `read(b, 0, b.length)`, which allows subclasses to only override `read(byte[], int, int)` and have `read(byte[])` work automatically. The method blocks until some data is available.

public int read(byte[] b, int off, int len) throws IOException

Parameters

`b`

An array of bytes to be filled from the stream.

`off`

An offset into the byte array.

`len`

The number of bytes to read.

Returns

The actual number of bytes read or `-1` if the end of the stream is encountered immediately.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`InputStream.read(byte[], int, int)`

Description

This method reads up to `len` bytes of input into the given array starting at index `off`. It does this by calling the `read(byte[], int, int)` method of the underlying stream and returning the result. The method blocks until some data is available.

reset

public synchronized void reset() throws IOException

Throws

`IOException`

If there was no previous call to the `mark()` method or the saved position has been invalidated.

Overrides

`InputStream.reset()`

Description

This method calls the `reset()` method of the underlying stream. If the underlying stream supports `mark()` and `reset()`, this method sets the position of the `FilterInputStream` to a position that was saved by a previous call to `mark()`. Subsequent bytes read from this `FilterInputStream` will begin from the saved position and continue normally.

skip

public long skip(long n) throws IOException

Parameters

`n`

The number of bytes to skip.

Returns

The actual number of bytes skipped.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`InputStream.skip()`

Description

This method skips `n` bytes of input. It calls the `skip()` method of the underlying stream.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`BufferedInputStream`, `CheckedInputStream`, `DataInputStream`, `FilterInputStream`, `InflaterInputStream`, `InputStream`, `IOException`, `LineNumberInputStream`, `PushbackInputStream`

PREVIOUS | HOME | NEXT
FileWriter | BOOK INDEX | FilterOutputStream

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

FilterOutputStream

Name

FilterOutputStream

Synopsis

Class Name:

`java.io.FilterOutputStream`

Superclass:

`java.io.ObjectStream`

Immediate Subclasses:

`java.io.BufferedOutputStream`,

`java.io.DataOutputStream`,

`java.io.PrintStream`,

`java.util.zip.CheckedOutputStream`,

`java.util.zip.DeflaterOutputStream`

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The `FilterOutputStream` class is the superclass of all of the output stream classes that filter output. Each of the subclasses of `FilterOutputStream` works by wrapping an existing output stream, called the *underlying output stream*, and providing additional functionality. The methods of `FilterOutputStream` simply override the methods of `OutputStream` with versions that call the corresponding methods of the underlying stream.

`FilterOutputStream` cannot be instantiated directly; it must be subclassed. An instance of one of the subclasses of `FilterOutputStream` is constructed with another `OutputStream` object. The methods of a subclass of `FilterOutputStream` should override some methods in order to extend their behavior or provide some sort of filtering.

Class Summary

```
public class java.io.FilterOutputStream extends java.io.OutputStream {
  // Variables
  protected OutputStream out;
  // Constructors
  public FilterOutputStream(OutputStream out);
  // Instance Methods
  public void close();
  public void flush();
  public void write(int b);
  public void write(byte[] b);
  public void write(byte[] b, int off, int len);
}
```

Variables

out

protected OutputStream out

Description

The underlying stream that this `FilterOutputStream` wraps or filters.

Constructors

FilterOutputStream

public FilterOutputStream(OutputStream out)

Parameters

`out`

The output stream to filter.

Description

This constructor creates a `FilterOutputStream` that sends its data to `out`.

Instance Methods

close

public void close() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`OutputStream.close()`

Description

This method calls the `close()` method of the underlying stream, which releases any system resources associated with this object.

flush

public void flush() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`OutputStream.flush()`

Description

This method calls the `flush()` method of the underlying stream, which forces any bytes that may be buffered by this `FilterOutputStream` to be written to the underlying device.

write

public void write(int b) throws IOException

Parameters

`b`

The value to write.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`OutputStream.write(int)`

Description

This method writes a byte containing the low-order eight bits of the given integer value. The method calls the `write(int)` method of the underlying stream.

public void write(byte[] b) throws IOException

Parameters

`b`

An array of bytes to write to the stream.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`OutputStream.write(byte[])`

Description

This method writes the bytes contained in the given array. To accomplish this, it calls `write(b, 0, b.length)`. Subclasses need override only `write(byte[], int, int)` for this method to work automatically.

public void write(byte[] b, int off, int len) throws IOException

Parameters

`b`

An array of bytes to write to the stream.

`off`

An offset into the byte array.

`len`

The number of bytes to write.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`OutputStream.write(byte[], int, int)`

Description

This method writes `len` bytes contained in the given array, starting at offset `off`. It does this by calling `write(int)` for each element to be written in the array. This is inefficient, so subclasses should override `write(byte[], int, int)` with a method that efficiently writes a block of data.

Inherited Methods

Method	**Inherited From**	**Method**	**Inherited From**
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`BufferedOutputStream`, `CheckedOutputStream`, `DataOutputStream`, `DeflaterOutputStream`, `IOException`, `OutputStream`, `PrintStream`

PREVIOUS HOME NEXT
FilterInputStream BOOK INDEX FilterReader

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

FilterReader

Name

FilterReader

Synopsis

Class Name:

```
java.io.FilterReader
```

Superclass:

```
java.io.Reader
```

Immediate Subclasses:

```
java.io.PushbackReader
```

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `FilterReader` class is the superclass of all of the reader classes that filter input. A subclass of `FilterReader` works by wrapping an existing reader, called the *underlying reader*, and providing additional functionality. The methods of `FilterReader` simply override the methods of `Reader` with versions that call the corresponding methods of the underlying reader.

`FilterReader` cannot be instantiated directly; it must be subclassed. An instance of a subclass of `FilterReader` is constructed with another `Reader` object. The methods of a subclass of `FilterReader` should override some methods in order to extend their behavior or provide some sort of filtering.

`FilterReader` is like `FilterInputStream`, except that it deals with a character stream instead of a byte stream.

Class Summary

```
public abstract class java.io.FilterReader extends java.io.Reader {
  // Variables
  protected Reader in;
  // Constructors
  protected FilterReader(Reader in);
  // Instance Methods
  public void close();
  public void mark(int readAheadLimit);
  public boolean markSupported();
  public int read();
  public int read(char[] cbuf, int off, int len);
  public boolean ready();
  public void reset();
  public long skip(long n);
}
```

Variables

in

protected Reader in

Description

The underlying reader that this `FilterReader` wraps or filters.

Constructors

FilterReader

protected FilterReader(Reader in)

Parameters

`in`

The input reader to filter.

Description

This constructor creates a `FilterReader` that gets data from `in`.

Instance Methods

close

public void close() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`Reader.close()`

Description

This method calls the `close()` method of the underlying reader, which releases any system resources associated with this object.

mark

public void mark(int readAheadLimit) throws IOException

Parameters

`readAheadLimit`

The maximum number of characters that can be read before the saved position becomes invalid.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`Reader.mark()`

Description

This method calls the `mark()` method of the underlying reader. If the underlying reader supports `mark()` and `reset()`, this method causes the `FilterReader` to remember its current position. A subsequent call to `reset()` causes the object to return to that saved position, and thus re-read a portion of the input.

markSupported

public boolean markSupported()

Returns

`true` if this reader supports `mark()` and `reset()`; `false` otherwise.

Overrides

`Reader.markSupported()`

Description

This method calls the `markSupported()` method of the underlying reader and returns the result.

read

public int read() throws IOException

Returns

The next character of data or `-1` if the end of the stream is encountered.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`Reader.read()`

Description

This method calls the `read()` method of the underlying reader and returns the result. The method blocks until data is available, the end of the stream is detected, or an exception is thrown.

public int read(char[] cbuf, int off, int len) throws IOException

Parameters

`cbuf`

An array of characters to be filled from the stream.

`off`

An offset into the array.

`len`

The number of characters to read.

Returns

The actual number of characters read or `-1` if the end of the stream is encountered immediately.
Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`Reader.read(char[], int, int)`

Description

This method reads up to `len` characters of input into the given array starting at index `off`. It does this by calling the `read(char[], int, int)` method of the underlying reader and returning the result. The method blocks until some data is available.

ready

public boolean ready() throws IOException

Returns

A `boolean` value that indicates whether the reader is ready to be read.

Throws

`IOException`

If the stream is closed.

Overrides

`Reader.ready()`

Description

This method calls the `ready()` method of the underlying reader and returns the result. If the underlying stream is ready, this method returns `true`. The underlying stream is ready if the next `read()` is guaranteed not to block.

reset

public void reset() throws IOException

Throws

`IOException`

If the stream is closed, `mark()` has not been called, or the saved position has been invalidated.

Overrides

`Reader.reset()`

Description

This method calls the `reset()` method of the underlying reader. If the underlying reader supports `mark()` and `reset()`, this method sets the position of the `FilteredReader` to a position that was saved by a previous call to `mark()`. Subsequent characters read from this `FilteredReader` will begin from the saved position and continue normally.

skip

public long skip(long n) throws IOException

Parameters

`n`

The number of characters to skip.

Returns

The actual number of characters skipped.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`Reader.skip()`

Description

This method skips `n` characters of input. It calls the `skip()` method of the underlying reader.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`FilterInputStream`, `IOException`, `PushbackReader`, `Reader`

PREVIOUS
FilterOutputStream

HOME
BOOK INDEX

NEXT
FilterWriter

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

FilterWriter

Name

FilterWriter

Synopsis

Class Name:

`java.io.FilterWriter`

Superclass:

`java.io.Writer`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `FilterWriter` class is the superclass of all of the writer classes that filter output. A subclass of `FilterWriter` works by wrapping an existing writer, called the *underlying writer*, and providing additional functionality. The methods of `FilterWriter` simply override the methods of `Writer` with versions that call the corresponding methods of the underlying writer.

`FilterWriter` cannot be instantiated directly; it must be subclassed. An instance of a subclass of `FilterWriter` is constructed with another `Writer` object. The methods of a subclass of `FilterWriter` should override some methods in order to extend their behavior or provide some sort of filtering.

`FilterWriter` is like `FilterOutputStream`, except that it deals with a character stream instead of a byte stream.

Class Summary

```
public abstract class java.io.FilterWriter extends java.io.Writer {
  // Variables
  protected Writer out;
  // Constructors
  protected FilterWriter(Writer out);
  // Instance Methods
  public void close();
  public void flush();
  public void write(int c);
  public void write(char[] cbuf, int off, int len);
  public void write(String str, int off, int len);
}
```

Variables

out

protected Writer out

Description

> The underlying writer that this `FilterWriter` wraps or filters.

Constructors

FilterWriter

public FilterWriter(Writer out)

Parameters

`out`

The output writer to filter.

Description

This constructor creates a `FilterWriter` that sends data to `out`.

Instance Methods

close

public void close() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`Writer.close()`

Description

This method calls the `close()` method of the underlying writer, which releases any system resources associated with this object.

flush

public void flush() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`Writer.flush()`

Description

This method calls the `flush()` method of the underlying writer, which forces any characters that may be buffered by this `FilterWriter` to be written to the underlying device.

write

public void write(int c) throws IOException

Parameters

`c`

The value to write.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`Writer.write(int)`

Description

This method writes a character containing the low-order 16 bits of the given integer value. It calls the `write(int)` method of the underlying writer.

`public void write(char[] cbuf, int off, int len) throws IOException`

Parameters

`cbuf`

An array of characters to write to the stream.

`off`

An offset into the array.

`len`

The number of characters to write.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`Writer.write(char[], int, int)`

Description

This method writes `len` characters contained in the given array, starting at offset `off`. It does this by calling the `write(char[], int, int)` method of the underlying writer.

public void write(String str, int off, int len) throws IOException

Parameters

`str`

A string to write to the stream.

`off`

An offset into the string.

`len`

The number of characters to write.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`Writer.write(String, int, int)`

Description

This method writes `len` characters contained in the given string, starting at offset `off`. It does this by calling the `write(String, int, int)` method of the underlying writer.

Inherited Methods

Method	**Inherited From**	**Method**	**Inherited From**
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`FilterOutputStream`, `IOException`, `String`, `Writer`

PREVIOUS
FilterReader

HOME
BOOK INDEX

NEXT
InputStream

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

InputStream

Name

InputStream

Synopsis

Class Name:

```
java.io.InputStream
```

Superclass:

```
java.lang.Object
```

Immediate Subclasses:

```
java.io.ByteArrayInputStream,

java.io.FileInputStream,

java.io.FilterInputStream,

java.io.ObjectInputStream,

java.io.PipedInputStream,

java.io.SequenceInputStream,
```

java.io.StringBufferInputStream

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The InputStream class is an abstract class that is the superclass of all classes that represent input byte streams. InputStream defines the basic input methods that all input streams provide. A similar hierarchy of classes, based around Reader, deals with character streams instead of byte streams.

InputStream is designed so that read(byte[]) and read(byte[], int, int) both call read(). Thus, a subclass can simply override read(), and all the read methods will work. However, for efficiency sake, read(byte[], int, int) should also be overridden with a method that can read a block of data more efficiently than reading each byte separately.

InputStream also defines a mechanism for marking a position in the stream and returning to it later, via the mark() and reset() methods. Another method, markSupported(), indicates whether or not this mark-and-reset functionality is available in a particular subclass.

Class Summary

```
public abstract class java.io.InputStream extends java.lang.Object {
  // Instance Methods
  public abstract int available();
  public void close();
  public synchronized void mark(int readlimit);
  public boolean markSupported();
  public abstract int read();
  public int read(byte[] b);
  public int read(byte[] b, int off, int len);
  public synchronized void reset();
  public long skip(long n);
}
```

Instance Methods

available

public abstract int available() throws IOException

Returns

The number of bytes that can be read without blocking.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method returns the number of bytes that can be read without having to wait for more data to become available, or in other words, blocking.

A subclass of `InputStream` must implement this method.

close

public void close() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method closes the input stream and releases any resources associated with it.

The implementation of the `close()` method in `InputStream` does nothing; a subclass should override this method to handle cleanup for the stream.

mark

public synchronized void mark(int readlimit)

Parameters

`readlimit`

The maximum number of bytes that can be read before the saved position can become invalid.

Description

This method tells this `InputStream` object to remember its current position, so that the position can be restored by a call to the `reset()` method. The `InputStream` can read `readlimit` bytes beyond the marked position before the mark becomes invalid.

The implementation of the `mark()` method in `InputStream` does nothing; a subclass must override the method to provide the mark-and-reset functionality.

markSupported

public boolean markSupported()

Returns

`true` if this input stream supports `mark()` and `reset()`; `false` otherwise.

Description

This method returns a `boolean` value that indicates whether or not this object supports mark-and-reset functionality.

The `markSupported()` method in `InputStream` always returns `false`. A subclass that implements the mark-and-reset functionality should override the method to return `true`.

read

public abstract int read() throws IOException

Returns

The next byte of data or `-1` if the end of the stream is encountered.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method reads the next byte of input. The byte is returned as an integer in the range 0 to 255. The method blocks until the byte is read, the end of stream is encountered, or an exception is thrown.

A subclass of `InputStream` must implement this method.

public int read(byte[] b) throws IOException

Parameters

`b`

An array of bytes to be filled from the stream.

Returns

The actual number of bytes read or `-1` if the end of the stream is encountered immediately.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method reads bytes of input to fill the given array by calling `read(b, 0, b.length)`. The method blocks until some data is available.

A subclass does not usually need to override this method as it can override `read(byte[], int, int)` and have `read(byte[])` work automatically.

public int read(byte[] b, int off, int len) throws IOException

Parameters

`b`

An array of bytes to be filled from the stream.

`off`

An offset into the array.

`len`

The number of bytes to read.

Returns

The actual number of bytes read or `-1` if the end of the stream is encountered immediately.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method reads up to `len` bytes of input into the given array starting at index `off`. The method blocks until some data is available.

The implementation of this method in `InputStream` uses `read()` repeatedly to fill the array. Although it is not strictly necessary, a subclass should override this method to read a block of data more efficiently.

reset

public synchronized void reset() throws IOException

Throws

`IOException`

If there was no previous call to the `mark()` method or the saved position has been invalidated.

Description

This method restores the position of the stream to the position that was saved by a previous call to `mark()`.

The implementation of the `reset()` method in `InputStream` throws an exception to indicate that mark-and-reset functionality is not supported by default. A subclass must override the method to provide the functionality.

skip

public long skip(long n) throws IOException

Parameters

`n`

The number of bytes to skip.

Returns

The actual number of bytes skipped.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method skips `n` bytes of input. In other words, it moves the position of the stream forward by `n` bytes.

The implementation of the `skip()` method in `InputStream` simply calls `read(b)` where `b` is a byte array `n` bytes long. A subclass may want to override this method to implement a more efficient skipping algorithm.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`ByteArrayInputStream`, `FileInputStream`, `FilterInputStream`, `IOException`, `ObjectInputStream`, `PipedInputStream`, `SequenceInputStream`, `StringBufferInputStream`

InputStreamReader

Name

InputStreamReader

Synopsis

Class Name:

```
java.io.InputStreamReader
```

Superclass:

```
java.io.Reader
```

Immediate Subclasses:

```
java.io.FileReader
```

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `InputStreamReader` class is a bridge between the byte-oriented world of the `InputStream` class and the character-oriented world of the `Reader` class. The `InputStreamReader` represents a character stream, but it gets its input from an underlying byte stream. An encoding scheme is responsible for translating the bytes to Unicode characters. An `InputStreamReader` can be created using an explicit encoding scheme or a default encoding scheme.

For example, to read an ISO-8859-5 byte stream as a Unicode character stream, you can construct an `InputStreamReader` with the encoding "8859_5" as follows:

```
InputStreamReader inr = new InputStreamReader(in, "8859_5");
```

Each time you read from an `InputStreamReader` object, bytes may be read from the underlying byte stream. To improve efficiency, you may want to wrap the `InputStreamReader` in a `BufferedReader`.

Class Summary

```
public class java.io.InputStreamReader extends java.io.Reader {
  // Constructors
  public InputStreamReader(InputStream in);
  public InputStreamReader(InputStream in, String enc);
  // Instance Methods
  public void close();
  public String getEncoding();
  public int read();
  public int read(char[] cbuf, int off, int len);
  public boolean ready();
}
```

Constructors

InputStreamReader

public InputStreamReader(InputStream in)

Parameters

`in`

The input stream to use.

Description

This constructor creates an `InputStreamReader` that gets its data from `in` and translates bytes to characters using the system's default encoding scheme.

`public InputStreamReader(InputStream in, String enc) throws UnsupportedEncodingException`

Parameters

`in`

The input stream to use.

`enc`

The name of an encoding scheme.

Throws

`UnsupportedEncodingException`

If `enc` is not a supported encoding scheme.

Description

This constructor creates an `InputStreamReader` that gets its data from `in` and translates bytes to characters using the given encoding scheme.

Instance Methods

close

public void close() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`Reader.close()`

Description

This method calls the `close()` method of the underlying input stream, which releases any system resources associated with this object.

getEncoding

public String getEncoding()

Returns

A `String` that contains the name of the character encoding scheme of this reader.

Description

This method returns the name of the character encoding scheme this `InputStreamReader` is currently using.

read

public int read() throws IOException

Returns

The next character of data or `-1` if the end of the stream is encountered.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`Reader.read()`

Description

This method reads a character of input. The method returns the next character that has been read and converted from the underlying byte-oriented `InputStream`. The `InputStreamReader` class uses buffering internally, so this method returns immediately unless the buffer is empty. If the buffer is empty, a new block of bytes is read from the `InputStream` and converted to characters. The method blocks until the character is read, the end of stream is encountered, or an exception is thrown.

public int read(char[] cbuf, int off, int len) throws IOException

Parameters

`cbuf`

An array of characters to be filled from the stream.

`off`

An offset into the array.

`len`

The number of characters to read.

Returns

The actual number of characters read or `-1` if the end of the stream is encountered immediately.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`Reader.read(char[], int, int)`

Description

This method reads up to `len` characters of input into the given array starting at index `off`. The `InputStreamReader` class uses buffering internally, so this method returns immediately if there is enough data in the buffer. If there is not enough data, a new block of bytes is read from the `InputStream` and converted to characters. The method blocks until some data is available.

ready

public boolean ready() throws IOException

Returns

`true` if the reader is ready to be read; `false` otherwise.

Throws

`IOException`

If the reader is closed or any other kind of I/O error occurs.

Overrides

`Reader.ready()`

Description

This method returns a `boolean` value that indicates whether or not the reader is ready to be read. If there is data available in the internal buffer or if there are bytes available to be read from the underlying byte stream, the method returns `true`. Otherwise it returns `false`.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`

`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`mark()`	`Reader`
`markSupported()`	`Reader`	`read(char[])`	`Reader`
`reset()`	`Reader`	`skip(long)`	`Reader`
`toString()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`BufferedReader`, `FileReader`, `InputStream`, `IOException`, , `Reader`, `UnsupportedEncodingException`

PREVIOUS HOME NEXT
InputStream BOOK INDEX InterruptedIOException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

InvalidClassException

Name

InvalidClassException

Synopsis

Class Name:

`java.io.InvalidClassException`

Superclass:

`java.io.ObjectStreamException`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

An `InvalidClassException` is thrown during object serialization. It indicates that the run-time environment does not support a serialized class for one of the following reasons:

- The version of the class does not match the serial version of the class in the stream.
- The class contains unknown data types.

An `InvalidClassException` can also indicate one of these problems with the class itself:

- The class implements only one of `writeObject()` and `readObject()`.
- The class is not `public`.
- The class does not have an accessible constructor that takes no arguments.

Class Summary

```
public class java.io.InvalidClassException
             extends java.io.ObjectStreamException {
  // Variables
  public String classname;
  // Constructors
  public InvalidClassException(String reason);
  public InvalidClassException(String cname, String reason);
  // Instance Methods
  public String getMessage();
}
```

Variables

classname

public String classname

Description

> The name of the class that caused the exception.

Constructors

InvalidClassException

public InvalidClassException(String reason)

Parameters

`reason`

The reason the exception was thrown.

Description

This constructor creates an `InvalidClassException` with the specified reason string.

public InvalidClassException(String cname, String reason)

Parameters

`cname`

The name of the class.

`reason`

The reason the exception was thrown.

Description

This constructor creates an `InvalidClassException` with the specified class name and reason string.

Instance Methods

getMessage

public String getMessage()

Returns

The reason string for this exception.

Overrides

`Throwable.getMessage()`

Description

This method returns the reason string for this exception. If a class name has also been specified, it is prepended to the reason string with a semicolon.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Exception`, `ObjectStreamException`, `Throwable`

PREVIOUS HOME NEXT
InterruptedIOException BOOK INDEX InvalidObjectException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

InvalidObjectException

Name

InvalidObjectException

Synopsis

Class Name:

`java.io.InvalidObjectException`

Superclass:

`java.io.ObjectStreamException`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

An `InvalidObjectException` is thrown by an object to indicate that it cannot validate itself during object deserialization.

Class Summary

```
public class java.io.InvalidObjectException
             extends java.io.ObjectStreamException {
  // Constructors
  public InvalidObjectException(String reason);
}
```

Constructors

InvalidObjectException

public InvalidObjectException(String reason)

Parameters

`reason`

The detail message.

Description

This constructor creates an `InvalidObjectException` with the specified detail message, which should be the name of the class.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`

`wait(long, int)`	`Object`

See Also

`Exception`, `ObjectStreamException`, `Throwable`

PREVIOUS | HOME | NEXT
InvalidClassException | BOOK INDEX | IOException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

IOException

Name

IOException

Synopsis

Class Name:

`java.io.IOException`

Superclass:

`java.lang.Exception`

Immediate Subclasses:

`java.io.CharConversionException`,

`java.io.EOFException`,

`java.io.FileNotFoundException`,

`java.io.InterruptedIOException`,

`java.io.ObjectStreamException`,

`java.io.SyncFailedException`,

`java.io.UnsupportedEncodingException`,

`java.io.UTFDataFormatException`,

`java.net.MalformedURLException`,

`java.net.ProtocolException`,

`java.net.SocketException`,

`java.net.UnknownHostException`,

`java.net.UnknownServiceException`,

`java.util.zip.ZipException`

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The `IOException` class is the superclass for all of the exceptions that represent anything that can go wrong with input or output.

Class Summary

```
public class java.io.IOException extends java.lang.Exception {
  // Constructors
  public IOException();
  public IOException(String s);
}
```

Constructors

IOException

public IOException()

Description

This constructor creates an `IOException` with no detail message.

public IOException(String s)

Parameters

`s`

The detail message.

Description

This constructor creates an `IOException` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`CharConversionException`, `EOFException`, `Exception`, `FileNotFoundException`, `InterruptedException`, `MalformedURLException`, `ObjectStreamException`, `ProtocolException`, `SocketException`, `SyncFailedException`, `Throwable`, `UnknownHostException`, `UnknownServiceException`, `UnsupportedEncodingException`, `UTFDataFormatException`, `ZipException`

PREVIOUS | HOME | NEXT
InvalidObjectException | BOOK INDEX | LineNumberInputStream

LineNumberInputStream

Name

LineNumberInputStream

Synopsis

Class Name:

```
java.io.LineNumberInputStream
```

Superclass:

```
java.io.FilterInputStream
```

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

Deprecated as of JDK 1.1

Description

The `LineNumberInputStream` class is an `InputStream` that keeps track of line numbers. The line number starts at 0 and is incremented each time an end-of-line character is encountered. `LineNumberInputStream` recognizes `"\n"`, `"\r"`, or `"\r\n"` as the end of a line. Regardless of the end-of-line character it reads, `LineNumberInputStream` returns only `"\n"`. The current line number is returned by `getLineNumber()`. The `mark()` and `reset()` methods are supported, but only work if the underlying stream supports `mark()` and `reset()`.

The `LineNumberInputStream` class is deprecated as of JDK 1.1 because it does not perform any byte to character conversions. Incoming bytes are directly compared to end-of-line characters. If you are developing new code, you should use `LineNumberReader` instead.

Class Summary

```
public class java.io.LineNumberInputStream
             extends java.io.FilterInputStream {
  // Constructors
  public LineNumberInputStream(InputStream in);
  // Instance Methods
  public int available();
  public int getLineNumber();
  public void mark(int readlimit);
  public int read();
  public int read(byte[] b, int off, int len);
  public void reset();
  public void setLineNumber(int lineNumber);
  public long skip(long n);
}
```

Constructors

LineNumberInputStream

public LineNumberInputStream(InputStream in)

Parameters

`in`

The input stream to use.

Description

This constructor creates a `LineNumberInputStream` that gets its data from `in`.

Instance Methods

available

public int available() throws IOException

Returns

The number of bytes that can be read without blocking.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterInputStream.available()`

Description

This method returns the number of bytes of input that can be read without having to wait for more input to become available.

getLineNumber

public int getLineNumber()

Returns

The current line number.

Description

This method returns the current line number.

mark

public void mark(int readlimit)

Parameters

`readlimit`

The maximum number of bytes that can be read before the saved position becomes invalid.

Overrides

`FilterInputStream.mark()`

Description

This method tells the `LineNumberInputStream` to remember its current position. A subsequent call to `reset()` causes the object to return to that saved position and thus reread a portion of the input. The method calls the `mark()` method of the underlying stream, so it only works if the underlying stream supports `mark()` and `reset()`.

read

public int read() throws IOException

Returns

The next byte of data or `-1` if the end of the stream is encountered.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterInputStream.read()`

Description

This method reads a byte of input from the underlying stream. If `"\n"`, `"\r"`, or `"\r\n"` is read from the stream, `"\n"` is returned. Otherwise, the byte read from the underlying stream is returned verbatim. The method blocks until the byte is read, the end of stream is encountered, or an exception is thrown.

public int read(byte[] b, int off, int len) throws IOException

Parameters

`b`

An array of bytes to be filled from the stream.

`off`

An offset into the byte array.

`len`

The number of bytes to read.

Returns

The actual number of bytes read or `-1` if the end of the stream is encountered immediately.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterInputStream.read(byte[], int, int)`

Description

This method reads up to `len` bytes of input into the given array starting at index `off`. If `"\n"`, `"\r"`, or `"\r\n"` is read from the stream, `"\n"` is returned. The method does this by repeatedly calling `read()`, which is not efficient, especially if the underlying stream is not buffered. The method blocks until some data is available.

reset

public void reset() throws IOException

Throws

`IOException`

If there was no previous call to this `FilterInputStream`'s `mark()` method or the saved position has been invalidated.

Overrides

`FilterInputStream.reset()`

Description

This method calls the `reset()` method of the underlying stream. If the underlying stream supports `mark()` and `reset()`, this method sets the position of the stream to a position that was saved by a previous call to `mark()`. Subsequent bytes read from this stream will begin from the saved position and continue normally. The method also restores the line number to its correct value for the mark location. The method only works if the underlying stream supports `mark()` and `reset()`.

setLineNumber

public void setLineNumber(int lineNumber)

Parameters

`lineNumber`

The new line number.

Description

This method sets the current line number of the `LineNumberInputStream`. The method does not change the position of the stream.

skip

public long skip(long n) throws IOException

Parameters

`n`

The number of bytes to skip.

Returns

The actual number of bytes skipped.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterInputStream.skip()`

Description

This method skips `n` bytes of input. Note that since `LineNumberInputStream` returns `"\r\n"` as a single character, `"\n"`, this method may skip over more bytes than you expect.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`close()`	`FilterInputStream`
`equals(Object)`	`Object`	`finalize()`	`Object`

`getClass()`	`Object`	`hashCode()`	`Object`
`markSupported()`	`FilterInputStream`	`notify()`	`Object`
`notifyAll()`	`Object`	`read(byte[])`	`FilterInputStream`
`toString()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`FilterInputStream`, `InputStream`, `IOException`, `LineNumberReader`

PREVIOUS | HOME | NEXT
IOException | BOOK INDEX | LineNumberReader

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

LineNumberReader

Name

LineNumberReader

Synopsis

Class Name:

`java.io.LineNumberReader`

Superclass:

`java.io.BufferedReader`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `LineNumberReader` class is a `BufferedReader` that keeps track of line numbers. The line number starts at 0 and is incremented each time an end-of-line character is encountered. `LineNumberReader` recognizes `"\n"`, `"\r"`, or `"\r\n"` as the end of a line. Regardless of the end-of-line character it reads, `ReaderInputStream` returns only `"\n"`. The current line number is returned by `getLineNumber()`.

The `LineNumberReader` class is the JDK 1.1 replacement for `LineNumberInputStream`. Not only does it correctly handle byte to character conversions (via `Reader`), it implements `read(byte[], int, int)` and `skip()` more efficiently than its predecessor.

Class Summary

```
public class java.io.LineNumberReader extends java.io.BufferedReader {
  // Constructors
  public LineNumberReader(Reader in);
  public LineNumberReader(Reader in, int sz);
  // Instance Methods
  public int getLineNumber();
  public void mark(int readAheadLimit);
  public int read();
  public int read(char[] cbuf, int off, int len);
  public String readLine();
  public void reset();
  public void setLineNumber(int lineNumber);
  public long skip(long n);
}
```

Constructors

LineNumberReader

public LineNumberReader(Reader in)

Parameters

`in`

The reader to use.

Description

This constructor creates a `LineNumberReader` that gets its data from `in` and uses a default sized buffer. The default buffer size for `BufferedReader` is 8192 characters.

public LineNumberReader(Reader in, int sz)

Parameters

`in`

The reader to use.

`sz`

The buffer size.

Description

This constructor creates a `LineNumberReader` that gets its data from `in` and uses a buffer of the given size.

Instance Methods

getLineNumber

public int getLineNumber()

Returns

The current line number.

Description

This method returns the current line number.

mark

public void mark(int readAheadLimit) throws IOException

Parameters

`readAheadLimit`

The maximum number of characters that can be read before the saved position becomes invalid.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`BufferedReader.mark()`

Description

This method causes the `LineNumberReader` to remember its current position. A subsequent call to `reset()` causes the object to return to that saved position and thus reread a portion of the input.

read

public int read() throws IOException

Returns

The next character of data or `-1` if the end of the stream is encountered.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`BufferedReader.read()`

Description

This method reads a character of input from the underlying reader. If `"\n"`, `"\r"`, or `"\r\n"` is read from the stream, `"\n"` is returned. Otherwise the character read from the underlying `BufferedReader` is returned verbatim. The method blocks until it reads the character, the end of stream is encountered, or an exception is thrown.

public int read(char[] cbuf, int off, int len) throws IOException

Parameters

`cbuf`

An array of characters to be filled from the stream.

`off`

An offset into the array.

`len`

The number of characters to read.

Returns

The actual number of characters read or `-1` if the end of the stream is encountered immediately.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`BufferedReader.read(char[], int, int)`

Description

This method reads up to `len` characters of input into the given array starting at index `off`. This method, unlike `read()`, returns end-of-line characters exactly as they come from the underlying `BufferedReader`. The method blocks until data is available.

readLine

public String readLine() throws IOException

Returns

A `String` containing the line just read, or `null` if the end of the stream has been reached.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`BufferedReader.readLine()`

Description

This method reads a line of text. Lines are terminated by `"\n"`, `"\r"`, or `"\r\n"`. The line terminators are not returned with the line string.

reset

public void reset() throws IOException

Throws

`IOException`

If the reader is closed, `mark()` has not been called, or the saved position has been invalidated.

Overrides

`BufferedReader.reset()`

Description

This method sets the position of the reader to a position that was saved by a previous call to `mark()`. Subsequent characters read from this reader will begin from the saved position and continue normally. The method also restores the line number to its correct value for the mark location.

setLineNumber

public void setLineNumber(int lineNumber)

Parameters

`lineNumber`

The new line number.

Description

This method sets the current line number of the `LineNumberReader`. The method does not change the position of the reader.

skip

public long skip(long n) throws IOException

Parameters

`n`

The number of characters to skip.

Returns

The actual number of bytes skipped.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`BufferedReader.skip()`

Description

This method skips `n` characters of input.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`close()`	`BufferedReader`
`equals(Object)`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`markSupported()`	`BufferedReader`	`read(char[])`	`Reader`
`ready()`	`BufferedReader`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`BufferedReader`, `Reader`, `IOException`, `LineNumberInputStream`

PREVIOUS
LineNumberInputStream

HOME
BOOK INDEX

NEXT
NotActiveException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

NotActiveException

Name

NotActiveException

Synopsis

Class Name:

java.io.NotActiveException

Superclass:

java.io.ObjectStreamException

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

A `NotActiveException` is thrown to indicate that an inappropriate method is being called when serialization or deserialization is not in progress.

Class Summary

```
public class java.io.NotActiveException
             extends java.io.ObjectStreamException {
  // Constructors
  public NotActiveException();
  public NotActiveException(String reason);
}
```

Constructors

NotActiveException

public NotActiveException()

Description

This constructor creates a `NotActiveException` with no detail message.

public NotActiveException(String reason)

Parameters

`reason`

The detail message.

Description

This constructor creates a `NotActiveException` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`

`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Exception`, `ObjectStreamException`, `Throwable`

PREVIOUS HOME NEXT
LineNumberReader BOOK INDEX NotSerializableException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

ObjectInput

Name

ObjectInput

Synopsis

Interface Name:

`java.io.ObjectInput`

Super-interface:

`java.io.DataInput`

Immediate Sub-interfaces:

None

Implemented By:

`java.io.ObjectInputStream`

Availability:

New as of JDK 1.1

Description

The `ObjectInput` interface extends the `DataInput` interface for object serialization. While `DataInput` defines methods for reading primitive types from a stream, `ObjectInput` defines methods for reading objects and arrays of bytes.

Interface Declaration

```
public abstract interface java.io.ObjectInput extends java.io.DataInput {
  // Methods
  public abstract int available();
  public abstract void close();
  public abstract int read();
  public abstract int read(byte[] b);
  public abstract int read(byte[] b, int off, int len);
  public abstract Object readObject();
  public abstract long skip(long n);
}
```

Methods

available

public abstract int available() throws IOException

Returns

> The number of bytes that can be read without blocking.

Throws

> `IOException`
>
> > If any kind of I/O error occurs.

Description

> This method returns the number of bytes that can be read from the stream without accessing a physical device, like a disk or a network.

close

public abstract void close() throws IOException

Throws

> `IOException`
>
> > If any kind of I/O error occurs.

Description

This method closes the stream and releases any system resources associated with it.

read

public abstract int read() throws IOException

Returns

The `next byte` of data or `-1` if the end of the stream is encountered.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method returns the next byte of data from the stream. The method blocks until the byte is read, the end of stream is detected, or an exception is thrown.

public abstract int read(byte[] b) throws IOException

Parameters

`b`

An array of bytes to be filled from the stream.

Returns

The actual number of bytes read or `-1` if the end of the stream is encountered immediately.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method reads bytes from the stream to fill the given array. The method blocks until some data is available.

`public abstract int read(byte[] b, int off, int len) throws IOException`

Parameters

`b`

An array of bytes to be filled from the stream.

`off`

An offset into the array.

`len`

The number of bytes to read.

Returns

The actual number of bytes read or `-1` if the end of the stream is encountered immediately.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method reads up to `len` bytes of input into the given array starting at index `off`. The method blocks until some data is available.

readObject

`public abstract Object readObject() throws ClassNotFoundException, IOException`

Returns

An `Object` that has been deserialized from the stream.

Throws

`ClassNotFoundException`

If the class of the serialized object cannot be found in the run-time environment.

`IOException`

If any kind of I/O error occurs.

Description

This method reads and returns an object instance from the stream; in other words, it deserializes an object from the stream. The class that implements this interface determines exactly how the object is to be read.

skip

public abstract long skip(long n) throws IOException

Parameters

`n`

The number of bytes to skip.

Returns

The actual number of bytes skipped.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method skips `n` bytes of input.

Inherited Methods

Method	Inherited From	Method	Inherited From
`readBoolean()`	`DataInput`	`readByte()`	`DataInput`
`readChar()`	`DataInput`	`readDouble()`	`DataInput`
`readFloat(byte[])`	`DataInput`	`readFully(byte[])`	`DataInput`
`readFully(byte[], int, int)`	`DataInput`	`readInt()`	`DataInput`
`readLine()`	`DataInput`	`readLong()`	`DataInput`
`readShort()`	`DataInput`	`readUnsignedByte()`	`DataInput`
`readUnsignedChar()`	`DataInput`	`readUTF()`	`DataInput`

`skipBytes(int)`	`DataInput`

See Also

`DataInput`, `ObjectInputStream`

ObjectInputStream

Name

ObjectInputStream

Synopsis

Class Name:

`java.io.ObjectInputStream`

Superclass:

`java.io.InputStream`

Immediate Subclasses:

None

Interfaces Implemented:

`java.io.ObjectInput`

Availability:

New as of JDK 1.1

Description

The `ObjectInputStream` class can read both primitive types and object instances from an underlying `InputStream`. The objects and other data must have been written by an `ObjectOutputStream`. These two classes can provide persistent storage of objects when they are used in conjunction with `FileInputStream` and `FileOutputStream`. The classes can also be used with socket streams to pass objects across the network.

Only objects that are instances of classes that implement the `Serializable` or `Externalizable` interfaces can be deserialized from an input stream. The default deserialization mechanism is implemented by `readObject()`. When an object is deserialized, the non-`static` and non-`transient` fields of the object are restored to the values they had when the object was serialized, including any other objects referenced by the object (except for those objects that do not implement the `Serializable` interface themselves). Graphs of objects are restored using a reference sharing mechanism. New object instances are always allocated during the deserialization process, to prevent existing objects from being overwritten. Deserialized objects are returned as instances of type `Object`, so they should be cast to the appropriate type. Strings and arrays are objects in Java, so they are treated as objects during deserialization.

For example, the following code opens a file called *color.ser* and reads a `Color` object:

```
FileInputStream fileIn;
ObjectInputStream in;
Color color;
try {
    fileIn = new FileInputStream("color.ser");
    in = new ObjectInputStream(fileIn);
    color = (Color)in.readObject();
    in.close();
}
catch (Exception e) {
    System.out.println("Error reading: " + e);
}
```

Classes that have `transient` instance variables may require special handling to reconstruct the values of these variables when objects are deserialized. Special handling may also be necessary to correctly deserialize objects that were serialized with a different version of their class than is in use when they are deserialized. Classes that require special handling during serialization and deserialization must implement the following methods (with these exact signatures):

```
private void readObject(ObjectOutputStream stream)
             throws IOException, ClassNotFoundException
private void writeObject(ObjectOutputStream stream) throws IOException
```

The `writeObject()` method is responsible for writing the state of the object for the particular class so that it can be restored by `readObject()`. The `readObject()` method registers an object validation callback by calling `registerValidation()` as its first action. The `readObject()` method doesn't need to handle reading the state for the object's superclass or any of its subclasses except in the case where the superclass doesn't itself implement the `Serializable` interface. In this case, the nonserializable class must have a no-argument constructor that can be called to initialize its fields, and it is the responsibility of the subclass to restore the state of its superclass.

A class that inherits the implementation of `Serializable` prevents itself from being serialized by defining `readObject()` and `writeObject()` methods that throw `NotSerializableException` objects.

If a class needs complete control over the contents and formatting of the serialized form of its objects, it should implement the `Externalizable` interface.

Class Summary

```
public class java.io.ObjectInputStream extends java.io.InputStream
             implements java.io.ObjectInput {
  // Constructors
  public ObjectInputStream(InputStream in);
  // Public Instance Methods
  public int available();
  public void close();
  public final void defaultReadObject();
  public int read();
  public int read(byte[] data, int offset, int length);
  public boolean readBoolean();
  public byte readByte();
  public char readChar();
  public double readDouble();
  public float readFloat();
  public void readFully(byte[] data);
  public void readFully(byte[] data, int offset, int size);
  public int readInt();
  public String readLine();
  public long readLong();
  public final Object readObject();
  public short readShort();
  public int readUnsignedByte();
  public int readUnsignedShort();
  public String readUTF();
  public synchronized void
         registerValidation(ObjectInputValidation obj, int prio);
  public int skipBytes(int len);
  // Protected Instance Methods
  protected final boolean enableResolveObject(boolean enable);
  protected void readStreamHeader();
  protected Class resolveClass(ObjectStreamClass v);
  protected Object resolveObject(Object obj);
}
```

Constructors

ObjectInputStream

`public ObjectInputStream(InputStream in) throws IOException, StreamCorruptedException`

Parameters

`in`

The underlying input stream.

Throws

`IOException`

If any kind of I/O error occurs.

`StreamCorruptedException`

If the stream header is not correct.

Description

This constructor creates an `ObjectInputStream` that reads from the given input stream. The constructor attempts to read the stream header, which consists of a magic number and a version number, and if something goes wrong, an appropriate exception is thrown. If all of the bytes of the stream header are not available, the constructor does not return until they become available.

Public Instance Methods

available

public int available() throws IOException

Returns

The number of bytes that can be read without blocking.

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`ObjectInput.available()`

Overrides

`InputStream.available()`

Description

This method returns the number of bytes that can be read without having to wait for more data to become available.

close

public void close() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`ObjectInput.close()`

Overrides

`InputStream.close()`

Description

This method closes the stream and releases any system resources that are associated with it.

defaultReadObject

`public final void defaultReadObject() throws IOException, ClassNotFoundException, NotActiveException`

Throws

`IOException`

If any kind of I/O error occurs.

`ClassNotFoundException`

If the class of the object being read cannot be found.

`NotActiveException`

If serialization is not active.

Description

This method reads the fields of the current object that are not `static` and not `transient`. The method can only be called from the `private` `readObject()` method of an object that is being deserialized; it throws a `NotActiveException` if it is called at any other time. This method implements the default deserialization mechanism.

read

public int read() throws IOException

Returns

The next byte of data or `-1` if the end of the stream is encountered.

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`ObjectInput.read()`

Overrides

`InputStream.read()`

Description

This method reads the next byte from the stream. The method blocks until some data is available, the end of the stream is detected, or an exception is thrown.

`public int read(byte[] data, int offset, int length) throws IOException`

Parameters

`data`

Array of bytes to be filled from the stream.

`offset`

An offset into the byte array.

`length`

The number of bytes to read.

Returns

The number of bytes read or `-1` if the end of the stream is encountered immediately.

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`ObjectInput.read(byte[], int, int)`

Overrides

`InputStream.read(byte[], int, int)`

Description

This method reads up to `length` bytes of input into the given array starting at index `offset`. The method blocks until there is some input available.

readBoolean

public boolean readBoolean() throws IOException

Returns

The `boolean` value read from the stream.

Throws

`EOFException`

If the end of the file is encountered.

`IOException`

If any other kind of I/O error occurs.

Implements

`DataInput.readBoolean()`

Description

This method reads a byte as a `boolean` value from the underlying input stream. A byte that contains a zero is read as `false`. A byte that contains any other value is read as `true`. The method blocks until the byte is read, the end of the stream is encountered, or an exception is thrown.

readByte

public byte readByte() throws IOException

Returns

The `byte` value read from the stream.

Throws

`EOFException`

If the end of the file is encountered.

`IOException`

If any other kind of I/O error occurs.

Implements

`DataInput.readByte()`

Description

This method reads a signed 8-bit value, a `byte`, from the underlying input stream. The method blocks until the byte is read, the end of the stream is encountered, or an exception is thrown.

readChar

public char readChar() throws IOException

Returns

The `char` value read from the stream.

Throws

`EOFException`

If the end of the file is encountered.

`IOException`

If any other kind of I/O error occurs.

Implements

`DataInput.readChar()`

Description

This method reads a 16-bit Unicode character from the stream. The method reads two bytes from the underlying input stream and then creates a `char` value using the first byte read as the most significant byte. The method blocks until the two bytes are read, the end of the stream is encountered, or an exception is thrown.

readDouble

public double readDouble() throws IOException

Returns

The `double` value read from the stream.

Throws

`EOFException`

If the end of the file is encountered.

`IOException`

If any other kind of I/O error occurs.

Implements

`DataInput.readDouble()`

Description

This method reads a 64-bit `double` quantity from the stream. The method reads a `long` value from the underlying input stream as if using the `readLong()` method. The `long` value is then converted to a `double` using the `longBitsToDouble()` method in `Double`. The method blocks until the necessary eight bytes are read, the end of the stream is encountered, or an exception is thrown.

readFloat

public float readFloat() throws IOException

Returns

The `float` value read from the stream.

Throws

`EOFException`

If the end of the file is encountered.

`IOException`

If any other kind of I/O error occurs.

Implements

`DataInput.readFloat()`

Description

This method reads a 32-bit `float` quantity from the stream. The method reads an `int` value from the underlying input stream as if using the `readInt()` method. The `int` value is then converted to a `float` using the `intBitsToFloat()` method in `Float`. The method blocks until the necessary four bytes are read, the end of the stream is encountered, or an exception is thrown.

readFully

public void readFully(byte[] b) throws IOException

Parameters

`b`

The array to fill.

Throws

`EOFException`

If the end of the file is encountered.

`IOException`

If any other kind of I/O error occurs.

Implements

`DataInput.readFully(byte[])`

Description

This method reads bytes into the given array `b` until the array is full. The method reads repeatedly from the underlying stream to fill the array. The method blocks until all of the bytes are read, the end of the stream is encountered, or an exception is thrown.

`public void readFully(byte[] data, int offset, int size) throws IOException`

Parameters

`data`

The array to fill.

`offset`

An offset into the array.

`length`

The number of bytes to read.

Throws

`EOFException`

If the end of the file is encountered.

`IOException`

If any other kind of I/O error occurs.

Implements

`DataInput.readFully(byte[], int, int)`

Description

This method reads `len` bytes into the given array, starting at offset `off`. The method reads repeatedly from the underlying stream to fill the array. The method blocks until all of the bytes are read, the end of the stream is encountered, or an exception is thrown.

readInt

public int readInt() throws IOException

Returns

The `int` value read from the stream.

Throws

`EOFException`

If the end of the file is encountered.

`IOException`

If any other kind of I/O error occurs.

Implements

`DataInput.readInt()`

Description

This method reads a signed 32-bit `int` quantity from the stream. The method reads four bytes from the underlying input stream and then creates an `int` quantity, using the first byte read as the most significant byte. The method blocks until the four bytes are read, the end of the stream is encountered, or an exception is thrown.

readLine

public String readLine() throws IOException

Returns

A `String` that contains the line read from the stream.

Throws

`EOFException`

If the end of the file is encountered.

`IOException`

If any other I/O error occurs.

Implements

`DataInput.readLine()`

Description

This method reads the next line of text from the stream. The method reads bytes of data from the underlying input stream until it encounters a line terminator. A line terminator is a carriage return (`"\r"`), a newline character (`"\n"`), a carriage return immediately followed by a newline character, or the end of the stream. The method blocks until a line terminator is read, the end of the stream is encountered, or an exception is thrown. Note that this method calls the `readLine()` method of `DataInputStream`, which is deprecated in 1.1.

readLong

public long readLong() throws IOException

Returns

The `long` value read from the stream.

Throws

`EOFException`

If the end of the file is encountered.

`IOException`

If any other kind of I/O error occurs.

Implements

`DataInput.readLong()`

Description

This method reads a signed 64-bit `long` quantity from the stream. The method reads eight bytes from the underlying input stream and then creates a `long` quantity, using the first byte read as the most significant byte. The method blocks until the eight bytes are read, the end of the stream is encountered, or an exception is thrown.

readObject

`public final Object readObject() throws OptionalDataException, ClassNotFoundException, IOException`

Returns

An `Object` that has been deserialized from the stream.

Throws

`ClassNotFoundException`

If the object being deserialized has an unrecognized class.

`InvalidClassException`

If there is a problem with the class of the deserialized object.

`StreamCorruptedException`

If the stream serialization information is not correct.

`OptionalDataException`

If the stream contains primitive data instead of an object.

`IOException`

If any kind of I/O error occurs.

Implements

`ObjectInput.readObject()`

Description

This method deserializes an object from the stream and returns a reference to the object. The non-`static` and non-`transient` fields of the object are restored to the values they had when the object was serialized. If the object contains references to other objects, these objects are also deserialized (as long as they implement the `Serializable` interface). Graphs of objects are restored using a reference-sharing mechanism. New object instances are always allocated during the deserialization process, to prevent existing objects from being overwritten. Deserialized objects are returned as instances of type `Object`, so they should be cast to the appropriate type.

Once an object has been completely restored (i.e., all of its fields and any objects it references have been restored), any object validation callbacks for the object or any of the objects it references are called in an order based on their priority. An object validation callback is registered by the `private` `readObject()` method for an object.

readShort

public short readShort() throws IOException

Returns

> The `short` value read from the stream.

Throws

> `EOFException`
>
> > If the end of the file is encountered.
>
> `IOException`
>
> > If any other kind of I/O error occurs.

Implements

> `DataInput.readShort()`

Description

> This method reads a signed 16-bit `short` quantity from the stream. The method reads two bytes from the underlying input stream and then creates a `short` quantity, using the first byte read as the most significant byte. The method blocks until the two bytes are read, the end of the stream is encountered, or an exception is thrown.

readUnsignedByte

public int readUnsignedByte() throws IOException

Returns

> The unsigned `byte` value read from the stream.

Throws

> `EOFException`
>
> > If the end of the file is encountered.
>
> `IOException`
>
> > If any other kind of I/O error occurs.

Implements

> `DataInput.readUnsignedByte()`

Description

This method reads an unsigned 8-bit quantity from the stream. The method reads a byte from the underlying input stream and returns that byte. The method blocks until the byte is read, the end of the stream is encountered, or an exception is thrown.

readUnsignedShort

public int readUnsignedShort() throws IOException

Returns

The unsigned `short` value read from the stream.

Throws

`EOFException`

If the end of the file is encountered.

`IOException`

If any other kind of I/O error occurs.

Implements

`DataInput.readUnsignedShort()`

Description

This method reads an unsigned 16-bit quantity from the stream. The method reads two bytes from the underlying input stream and creates an unsigned `short` quantity using the first byte read as the most significant byte. The method blocks until the two bytes are read, the end of the stream is encountered, or an exception is thrown.

readUTF

public String readUTF() throws IOException

Returns

The `String` read from the stream.

Throws

`EOFException`

If the end of the file is encountered.

`IOException`

If any other kind of I/O error occurs.

`UTFDataFormatException`

If the bytes do not represent a valid UTF-8 encoding.

Implements

`DataInput.readUTF()`

Description

This method reads a UTF-8 encoded string from the stream. See the description of `DataInputStream.readUTF(DataInput)` for more information.

registerValidation

`public synchronized void registerValidation( ObjectInputValidation obj, int prio) throws NotActiveException, InvalidObjectException`

Parameters

`obj`

The object requesting validation.

`prio`

The priority of the validation callback; use zero as a default.

Throws

`NotActiveException`

If `serialization` is not active.

`InvalidObjectException`

If `obj` is `null`.

Description

This method may be called from an object's `private readObject()` method to register a validation callback. An object performs internal validation by implementing the `ObjectInputValidation` interface and registering itself with the `ObjectInputStream` via this function. When `ObjectInputStream` has completely deserialized an object (i.e., restored all of its fields and any objects it references), the stream calls `ObjectInputValidation.validateObject()` for every object that has an object validation callback. Objects that register with higher priority values get validated before objects that register with lower priority values. Within a priority value, the callbacks are not processed in any particular order.

skipBytes

public int skipBytes(int len) throws IOException

Parameters

`len`

The number of bytes to skip.

Returns

The actual number of skipped bytes.

Throws

`EOFException`

If the end of the file is encountered.

`IOException`

If any other kind I/O error occurs.

Implements

`DataInput.skipBytes()`

Description

This method skips over `n` bytes in the underlying input stream. The method blocks until all of the bytes are skipped, the end of the stream is encountered, or an exception is thrown.

Protected Instance Methods

enableResolveObject

```
protected final boolean enableResolveObject(boolean enable) throws SecurityException
```

Parameters

`enable`

A `boolean` value that specifies whether or not object replacement is enabled.

Returns

`true` if object replacement was previously enabled; `false` otherwise.

Throws

`SecurityException`

If `enable` is `true` and `getClassLoader()` called on the class of the stream does not return `null`.

Description

This method determines if a trusted subclass of `ObjectInputStream` is allowed to replace deserialized objects. If the method is called with `true`, object replacement is enabled. Each time an object is deserialized, `resolveObject()` is called to give the `ObjectInputStream` a chance to replace the object. A trusted stream is one whose class has no `ClassLoader`.

readStreamHeader

`protected void readStreamHeader() throws IOException, StreamCorruptedException`

Throws

`StreamCorruptedException`

If the stream header is not correct.

`IOException`

If any kind of I/O error occurs.

Description

This method attempts to read the stream header, which consists of a magic number and a version number. If something goes wrong, an appropriate exception is thrown. This method is called by the constructor for `ObjectInputStream` and is the source of the exceptions it throws. If you subclass `ObjectInputStream`, you can override this method to provide your own stream header checking.

resolveClass

`protected Class resolveClass(ObjectStreamClass v) throws IOException, ClassNotFoundException`

Parameters

`v`

The `ObjectStreamClass` to be resolved.

Returns

The `Class` that corresponds to the given `ObjectStreamClass`.

Throws

`ClassNotFoundException`

If the class of the given `ObjectStreamClass` cannot be found.

`IOException`

If any kind of I/O error occurs.

Description

This method attempts to find the `Class` object that corresponds to the supplied `ObjectStreamClass`. When a object is deserialized, its class information is read into an `ObjectStreamClass` object, which is then resolved to a `Class` if possible. Subclasses of `ObjectInputStream` can override this method to allow classes to be fetched from alternate sources. The version of the `ObjectStreamClass` and the `Class` must match.

resolveObject

protected Object resolveObject(Object obj) throws IOException

Parameters

`obj`

The object to be replaced.

Returns

A replacement for the given object.

Throws

`IOException`

If any kind of I/O error occurs.

Description

If object replacement is enabled for this `ObjectInputStream` (see `enableResolveObject()`), this method is called with each deserialized object to give the stream a chance to replace the object. In `ObjectInputStream`, this method simply returns the object that was passed to it. Subclasses can override this method to provide more useful functionality.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`mark()`	`InputStream`
`markSupported()`	`InputStream`	`notify()`	`Object`
`notifyAll()`	`Object`	`read(byte[])`	`InputStream`
`reset()`	`InputStream`	`skip(long n)`	`InputStream`
`toString()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`Class`, `ClassNotFoundException`, `DataInput`, `Double`, `EOFException`, `Externalizable`, `Float`, `InputStream`, `InvalidClassException`, `IOException`, `NotActiveException`, `ObjectInput`, `ObjectInputValidation`, `ObjectOuputStream`, `ObjectStreamClass`, `OptionalDataException`, `SecurityException`, `Serializable`, `StreamCorruptedException`, `String`, `UTFDataFormatException`

ObjectOutput

Name

ObjectOutput

Synopsis

Interface Name:

`java.io.ObjectOutput`

Super-interface:

`java.io.DataOutput`

Immediate Sub-interfaces:

None

Implemented By:

`java.io.ObjectOutputStream`

Availability:

New as of JDK 1.1

Description

The `ObjectOutput` interface extends the `DataOutput` interface for object serialization. While

`DataOutput` defines methods for reading primitive types from a stream, `ObjectOutput` defines methods for writing objects and arrays of bytes.

Interface Declaration

```
public abstract interface java.io.ObjectOutput extends java.io.DataOutput {
  // Methods
  public abstract void close();
  public abstract void flush();
  public abstract void write(int b);
  public abstract void write(byte[] b);
  public abstract void write(byte[] b, int off, int len);
  public abstract void writeObject(Object obj);
}
```

Methods

close

public abstract void close() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method closes the stream and releases any system resources associated with it.

flush

public abstract void flush() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Description

If the stream uses a buffer, this method forces any bytes that may be buffered by the output stream to be written to the underlying physical device.

write

public abstract void write(int b) throws IOException

Parameters

`b`

The value to write.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`DataOutput.write(int)`

Description

This method writes the lowest eight bits of the given integer `b` to the stream.

public abstract void write(byte[] b) throws IOException

Parameters

`b`

An array of bytes to write to the stream.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`DataOutput.write(byte[])`

Description

This method writes all of the 8-bit bytes in the given array to the stream.

`public abstract void write(byte[] b, int off, int len) throws IOException`

Parameters

`b`

An array of bytes to write to the stream.

`off`

An offset into the byte array.

`len`

The number of bytes to write.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`DataOutput.write(byte[], int, int)`

Description

This method writes `len` bytes from the given array, starting `off` elements from the beginning of the array, to the stream.

writeObject

public abstract void writeObject(Object obj) throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method writes the given object to the stream, or in other words, it serializes an object to the stream. The class that implements this interface determines how the object is written.

Inherited Methods

Method	Inherited From	Method	Inherited From
`writeBoolean(boolean)`	`DataOutput`	`writeByte(int)`	`DataOutput`
`writeBytes(String)`	`DataOutput`	`writeChar(int)`	`DataOutput`
`writeChars(String)`	`DataOutput`	`writeDouble(double)`	`DataOutput`
`writeFloat(float)`	`DataOutput`	`writeInt(int)`	`DataOutput`
`writeLong(long)`	`DataOutput`	`writeShort(int)`	`DataOutput`
`writeUTF(String)`	`DataOutput`		

See Also

`DataOutput`, `ObjectOutputStream`

PREVIOUS HOME NEXT
ObjectInputValidation BOOK INDEX ObjectOutputStream

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

ObjectOutputStream

Name

ObjectOutputStream

Synopsis

Class Name:

`java.io.ObjectOutputStream`

Superclass:

`java.io.OutputStream`

Immediate Subclasses:

None

Interfaces Implemented:

`java.io.ObjectOutput`

Availability:

New as of JDK 1.1

Description

The `ObjectOutputStream` class can write both primitive types and object instances to an underlying `OutputStream`. The objects and other data can then be read by an `ObjectInputStream`. These two classes provide persistent storage of objects when they are used in conjunction with `FileInputStream` and `FileOutputStream`. The classes can also be used with socket streams to pass objects across the network.

Only objects that are instances of classes that implement the `Serializable` or `Externalizable` interfaces can be serialized to an output stream. The default serialization mechanism is implemented by `writeObject()`. When an object is serialized, the class of the object is encoded, along with the class name, the signature of the class, the values of the non-`static` and non-`transient` fields of the object, including any other objects referenced by the object (except those that do not implement the `Serializable` interface themselves). Multiple references to the same object are encoded using a reference sharing mechanism, so that a graph of the object can be restored appropriately. Strings and arrays are objects in Java, so they are treated as objects during serialization.

For example, the following code opens a file called *color.ser* and writes out a `Color` object:

```
FileOutputStream fileOut;
ObjectOutputStream out;
try {
    fileOut = new FileOutputStream("color.ser");
    out = new ObjectOutputStream(fileOut);
    out.writeObject(Color.blue);
    out.close();
}
catch (IOException e) {
    System.out.println("Error writing: " + e);
}
```

Classes that require special handling during serialization and deserialization must implement the following methods (with these exact signatures):

```
private void readObject(ObjectOutputStream stream)
             throws IOException, ClassNotFoundException
private void writeObject(ObjectOutputStream stream) throws IOException
```

The `writeObject()` method is responsible for writing the state of the object for the particular class so that it can be restored by `readObject()`. The `writeObject()` method does not need to handle writing the state for the object's superclass or any of its subclasses except in the case where the superclass does not itself implement the `Serializable` interface. In this case, the nonserializable class must have a no-argument constructor that can be called to initialize its fields, and it is the responsibility of the subclass to save the state of its superclass.

A class that inherits the implementation of `Serializable` prevents itself from being serialized by defining `readObject()` and `writeObject()` methods that throw `NotSerializableException` objects.

If a class needs complete control over the contents and formatting of the serialized form of its objects, it should implement the `Externalizable` interface.

Class Summary

```
public class java.io.ObjectOutputStream extends java.io.OutputStream
             implements java.io.ObjectOutput {
  // Constructors
  public ObjectOutputStream(OutputStream out);
  // Instance Methods
  public void close();
  public final void defaultWriteObject();
  public void flush();
  public void reset();
  public void write(int data);
  public void write(byte[] b);
  public void write(byte[] b, int off, int len);
  public void writeBoolean(boolean data);
  public void writeByte(int data);
  public void writeBytes(String data);
  public void writeChar(int data);
  public void writeChars(String data);
  public void writeDouble(double data);
  public void writeFloat(float data);
  public void writeInt(int data);
  public void writeLong(long data);
  public final void writeObject(Object obj);
  public void writeShort(int data);
  public void writeUTF(String data);
  // Protected Instance Methods
  protected void annotateClass(Class cl);
  protected void drain();
  protected final boolean enableReplaceObject(boolean enable);
  protected Object replaceObject(Object obj);
  protected void writeStreamHeader();
}
```

Constructors

ObjectOutputStream

public ObjectOutputStream(OutputStream out) throws IOException

Parameters

`out`

The underlying output stream.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This constructor creates an `ObjectOutputStream` that writes to the given output stream. The constructor writes the stream header, which consists of a magic number and version number, in preparation for serialization.

Instance Methods

close

public void close() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`ObjectOutput.close()`

Overrides

`OutputStream.close()`

Description

This method closes the stream and releases any system resources that are associated with it.

defaultWriteObject

public final void defaultWriteObject() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

`NotActiveException`

If serialization is not active.

Description

This method writes the fields of the object that are not `static` or `transient`. The method can only be called from the `private` `writeObject()` method of an object that is being serialized; it throws a `NotActiveException` if it is called at any other time. This method implements the default serialization mechanism.

flush

public void flush() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`ObjectOutput.flush()`

Overrides

`OutputStream.flush()`

Description

This method takes any buffered output and forces it to be written to the underlying stream.

reset

public void reset() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method sets the state of the `ObjectOutputStream` to the same state as when it was created. As objects are serialized to the stream, the `ObjectOutputStream` remembers which ones are already serialized. If the program requests that already serialized objects be written again, the `ObjectOutputStream` just writes out a reference to the previous object. Calling `reset()` causes the `ObjectOutputStream` to forget what it has done before, so all subsequent objects are fully serialized.

write

public void write(int data) throws IOException

Parameters

`data`

The value to write.

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`ObjectOutput.write(int)`

Overrides

`OutputStream.write(int)`

Description

This method writes the lowest eight bits of `b` to the underlying stream as a `byte`.

public void write(byte[] b) throws IOException

Parameters

`b`

An array of bytes to write.

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`ObjectOutput.write(byte[])`

Overrides

`OutputStream.write(byte[])`

Description

This method writes the given array of bytes to the underlying stream.

public void write(byte[] b, int off, int len) throws IOException

Parameters

`b`

An array of bytes to write to the stream.

`off`

An offset into the byte array.

`len`

The number of bytes to write.

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`ObjectOutput.write(byte[], int, int)`

Overrides

`OutputStream.write(byte[], int, int)`

Description

This method writes `len` bytes from the given array, starting `off` elements from the beginning of the array, to the underlying stream.

writeBoolean

public void writeBoolean(boolean data) throws IOException

Parameters

`data`

The `boolean` value to write.

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`DataOutput.writeBoolean()`

Description

If `data` is `true`, this method writes a byte that contains the value `1` to the underlying stream. If `data` is `false`, the method writes a byte that contains the value 0.

writeByte

public void writeByte(int data) throws IOException

Parameters

`data`

The value to write.

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`DataOutput.writeByte()`

Description

This method writes an 8-bit `byte` to the underlying stream, using the lowest eight bits of the given integer `data`.

writeBytes

public void writeBytes(String data) throws IOException

Parameters

`data`

The `String` to write.

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`DataOutput.writeBytes()`

Description

This method writes the characters in the given `String` to the underlying stream as a sequence of 8-bit bytes. The high-order bytes of the characters in the string are ignored.

writeChar

public void writeChar(int data) throws IOException

Parameters

`data`

The value to write.

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`DataOutput.writeChar()`

Description

This method writes a 16-bit `char` to the underlying stream, using the lowest two bytes of the given integer `data`.

writeChars

public void writeChars(String data) throws IOException

Parameters

`data`

The `String` to write.

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`DataOutput.writeChars()`

Description

This method writes the characters in the given `String` object to the underlying stream as a sequence of 16-bit characters.

writeDouble

public void writeDouble(double data) throws IOException

Parameters

`data`

The `double` value to write.

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`DataOutput.writeDouble()`

Description

This method writes a 64-bit `double` to the underlying stream. The `double` value is converted to a `long` using `doubleToLongBits()` of `Double`; the `long` value is then written to the underlying stream as eight bytes with the highest byte first.

writeFloat

public void writeFloat(float data) throws IOException

Parameters

`data`

The `float` value to write.

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`DataOutput.writeFloat()`

Description

This method writes a 32-bit `float` to the underlying stream. The `float` value is converted to a `int` using `floatToIntBits()` of `Float`; the `int` value is then written to the underlying stream as four bytes with the highest byte first.

writeInt

public void writeInt(int data) throws IOException

Parameters

`data`

The `int` value to write.

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`DataOutput.writeInt()`

Description

This method writes a 32-bit `int` to the underlying stream. The value is written as four bytes with the highest byte first.

writeLong

public void writeLong(long data) throws IOException

Parameters

`data`

The `long` value to write.

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`DataOutput.writeLong()`

Description

This method writes a 64-bit `long` to the underlying stream. The value is written as eight bytes with the highest byte first.

writeObject

`public final void writeObject(Object obj) throws IOException, InvalidClassException, NotSerializableException`

Parameters

`obj`

The object to be serialized.

Throws

`InvalidClassException`

If there is a problem with the class of the object.

`NotSerializableException`

If the object does not implement `Serializable` or `Externalizable`.

`IOException`

If any kind of I/O error occurs.

Implements

`ObjectOutput.writeObject()`

Description

This method serializes the given object to the stream. The class of the object is encoded, along with the class name, the signature of the class, the values of the non-`static` and non-`transient` fields of the object, including any other objects referenced by the object (except those that do not implement the `Serializable` interface themselves). Multiple references to the same object are encoded using a reference sharing mechanism, so that a graph of object can be restored appropriately.

writeShort

public void writeShort(int data) throws IOException

Parameters

`data`

The value to write.

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`DataOutput.writeShort()`

Description

This method writes a 16-bit `short` to the underlying stream, using the lowest two bytes of the given integer `data`.

writeUTF

public void writeUTF(String data) throws IOException

Parameters

`data`

The `String` to write.

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`DataOutput.writeUTF()`

Description

This method writes the given `String` to the underlying stream using the UTF-8 encoding. See the description of `DataOutputStream.writeUTF(String)` for more information.

Protected Instance Methods

annotateClass

protected void annotateClass(Class cl) throws IOException

Parameters

`cl`

The class to be serialized.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method is called once for each unique class during serialization. The implementation in `ObjectOutputStream` does nothing; subclasses can override this method to write out more information about a class. A corresponding subclass of `ObjectInputStream` should override the `resolveClass()` method to read the extra class information.

drain

protected void drain() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method is a helper method for `flush()`. It forces a write of any buffered data in the `ObjectOutputStream`, but does not call `flush()` on the underlying stream.

enableReplaceObject

```
protected final boolean enableReplaceObject(boolean enable) throws SecurityException
```

Parameters

`enable`

A `boolean` value that specifies whether or not object replacement is enabled.

Returns

`true` if object replacement was previously enabled; `false` otherwise.

Throws

`SecurityException`

If `enable` is `true` and `getClassLoader()`called on the class of the stream does not return `null`.

Description

This method determines if a trusted subclass of `ObjectOutputStream` is allowed to replace serialized objects. If the method is called with `true`, replacement is enabled. Each time an object is serialized, `replaceObject()` is called to give the `ObjectOutputStream` a chance to replace the object. A trusted stream is one whose class has no `ClassLoader`.

replaceObject

protected Object replaceObject(Object obj) throws IOException

Parameters

`obj`

The object to be replaced.

Returns

A replacement for the given object.

Throws

`IOException`

If any kind of I/O error occurs.

Description

If object replacement is enabled for this `ObjectOutputStream` (see `enableReplaceObject()`), this method is called with each object to be serialized to give the stream a chance to replace the object. In `ObjectOutputStream`, this method simply returns the object that was passed to it. Subclasses can override this method to provide more useful functionality.

writeStreamHeader

protected void writeStreamHeader() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method writes the serialization stream header, which consists of a magic number and a version number. This method is called by the constructor for `ObjectOutputStream`. If you subclass `ObjectOutputStream`, you can override this method to provide your own stream header.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Class`, `DataOutput`, `Double`, `Externalizable`, `Float`, `InvalidClassException`, `IOException`, `NotActiveException`, `NotSerializableException`, `ObjectInputStream`, `ObjectOutput`, `OutputStream`, `SecurityException`, `Serializable`, `String`

PREVIOUS | HOME | NEXT

ObjectOutput | BOOK INDEX | ObjectStreamClass

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

ObjectStreamClass

Name

ObjectStreamClass

Synopsis

Class Name:

    `java.io.ObjectStreamClass`

Superclass:

    `java.lang.Object`

Immediate Subclasses:

    None

Interfaces Implemented:

    None

Availability:

    New as of JDK 1.1

Description

The `ObjectStreamClass` class represents a Java class during object serialization. When an object is deserialized, its class information is read into an `ObjectStreamClass`, which is then resolved to a `Class` if possible. An `ObjectStreamClass` instance contains the name and version information for a class.

Class Summary

```
public class java.io.ObjectStreamClass extends java.lang.Object
             implements java.io.Serializable {
  // Class Methods
  public static ObjectStreamClass lookup(Class cl);
  // Instance Methods
  public Class forClass();
  public String getName();
  public long getSerialVersionUID();
  public String toString();
}
```

Class Methods

lookup

public static ObjectStreamClass lookup(Class cl)

Parameters

`cl`

The `Class` to find.

Returns

An `ObjectStreamClass` that corresponds to the given `Class`.

Description

This method finds an `ObjectStreamClass` for the given `Class`. If the appropriate

ObjectStreamClass does not already exist, this method creates an ObjectStreamClass for the given Class. The method returns null if cl is not serializable.

Instance Methods

forClass

public Class forClass()

Returns

The Class that corresponds to this ObjectStreamClass.

Description

This method returns the Class in the run-time system that corresponds to this ObjectStreamClass. If there is no corresponding class, null is returned.

getName

public String getName()

Returns

The class name.

Description

This method returns the name of the class this ObjectStreamClass represents.

getSerialVersionUID

public long getSerialVersionUID()

Returns

The class version.

Description

This method returns the version of the class this `ObjectStreamClass` represents.

toString

public String toString()

Returns

`A string representation of this` object.

Overrides

`Object.toString()`

Description

This method returns a string that contains the class name and version information for this `ObjectStreamClass`.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`Class`, `ObjectInputStream`, `ObjectOutputStream`, `Serializable`

PREVIOUS
ObjectOutputStream

HOME
BOOK INDEX

NEXT
ObjectStreamException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

OutputStream

Name

OutputStream

Synopsis

Class Name:

```
    java.io.OutputStream
```

Superclass:

```
    java.lang.Object
```

Immediate Subclasses:

```
    java.io.ByteArrayOutputStream,

    java.io.FileOutputStream,

    java.io.FilterOutputStream,

    java.io.ObjectOutputStream,

    java.io.PipedOutputStream
```

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The `OutputStream` class is an `abstract` class that is the superclass of all classes that represent output byte streams. `OutputStream` defines the basic output methods that all output streams provide. A similar hierarchy of classes, based around `Writer`, deals with character streams instead of byte streams.

`OutputStream` is designed so that `write(byte[])` and `write(byte[], int, int)` call `write(int b)`. Thus, a subclass can simply override `write()`, and all the `write` methods will work. However, for efficiency's sake, `write(byte[], int, int)` should also be overridden with a method that can write a block of data more efficiently than writing each byte separately.

Some `OutputStream` subclasses may implement buffering to increase efficiency. `OutputStream` provides a method, `flush()`, that tells the `OutputStream` to write any buffered output to the underlying device, which may be a disk drive or a network.

Class Summary

```
public abstract class java.io.OutputStream extends java.lang.Object {
  // Instance Methods
  public void close();
  public void flush();
  public abstract void write(int b);
  public void write(byte[] b);
  public void write(byte[] b, int off, int len);
}
```

Instance Methods

close

public void close() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method closes the output stream and releases any resources associated with it.

The implementation of the `close()` method in `OutputStream` does nothing; a subclass should override this method to handle cleanup for the stream.

flush

public void flush() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method forces any bytes that may be buffered by the output stream to be written.

The implementation of `flush()` in `OutputStream` does nothing; a subclass should override this method as needed.

write

public abstract void write(int b) throws IOException

Parameters

`b`

The value to write to the stream.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method writes a byte of output. The method blocks until the byte is actually written.

A subclass of `OutputStream` must implement this method.

public void write(byte[] b) throws IOException

Parameters

`b`

An array of bytes to write to the stream.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method writes the bytes from the given array by calling `write(b, 0, b.length)`. The method blocks until the bytes are actually written.

A subclass does not usually need to override this method, as it can override `write(byte[], int, int)` and have `write(byte[])` work automatically.

public void write(byte[] b, int off, int len) throws IOException

Parameters

`b`

An array of bytes to write to the stream.

`off`

An offset into the byte array.

`len`

The number of bytes to write.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method writes `len` bytes of output from the given array, starting at offset `off`. The method blocks until the bytes are actually written.

The implementation of this method in `OutputStream` uses `write(int)` repeatedly to write the bytes. Although it is not strictly necessary, a subclass should override this method to write a block of data more efficiently.

Inherited Methods

Method	**Inherited From**	**Method**	**Inherited From**
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`ByteArrayOutputStream`, `FileOutputStream`, `FilterOutputStream`, `IOException`, `ObjectOutputStream`, `PipedOutputStream`

OutputStreamWriter

Name

OutputStreamWriter

Synopsis

Class Name:

`java.io.OutputStreamWriter`

Superclass:

`java.io.Writer`

Immediate Subclasses:

`java.io.FileWriter`

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `OutputStreamWriter` class is a bridge between the byte-oriented world of the `OutputStream` class and the character-oriented world of the `Writer` class. The `OutputStreamWriter` represents a character stream, but it sends its output to an underlying byte stream. A character encoding scheme is responsible for translating the Unicode characters to bytes. An `OutputStreamWriter` can be created using an explicit encoding scheme or a default encoding scheme.

For example, to write a Unicode character stream as an ISO-8859-5 byte stream, you can construct an `OutputStreamWriter` with the encoding `8859_5` as follows:

```
OutputStreamWriter outr = new OutputStreamWriter(out, "8859_5");
```

Each time you write to an `OutputStreamWriter` object, bytes may be written to the underlying byte stream. To improve efficiency, you may want to wrap the `OutputStreamWriter` in a `BufferedWriter`.

Class Summary

```
public class java.io.OutputStreamWriter extends java.io.Writer {
  // Constructors
  public OutputStreamWriter(OutputStream out);
  public OutputStreamWriter(OutputStream out, String enc);
  // Instance Methods
  public void close();
  public void flush();
  public String getEncoding();
  public void write(int c);
  public void write(char[] cbuf, int off, int len);
  public void write(String str, int off, int len);
}
```

Constructors

OutputStreamWriter

public OutputStreamWriter(OutputStream out)

Parameters

`out`

The output stream to use.

Description

This constructor creates an `OutputStreamWriter` that writes its data to `out` and translates characters to bytes using the system's default encoding scheme.

`public OutputStreamWriter(OutputStream out, String enc) throws UnsupportedEncodingException`

Parameters

`out`

The output stream to use.

`enc`

The name of an encoding scheme.

Throws

`UnsupportedEncodingException`

If `enc` is not a supported encoding scheme.

Description

This constructor creates an `OutputStreamWriter` that writes its data to `out` and translates characters to bytes using the given encoding scheme.

Instance Methods

close

public void close() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`Writer.close()`

Description

This method calls the `close()` method of the underlying output stream, which releases any system resources associated with this object.

flush

public void flush() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`Writer.flush()`

Description

This method writes out any buffered data in the internal buffer and calls the `flush()` method of the underlying output stream, which forces any bytes that may be buffered to be written to the underlying device.

getEncoding

public String getEncoding()

Returns

A `String` that contains the name of the character encoding scheme of this writer.

Description

This method returns the name of the character encoding scheme this `OutputStreamWriter` is currently using.

write

public void write(int c) throws IOException

Parameters

`c`

The value to write.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`Writer.write(int)`

Description

This method converts the given character to bytes using the current encoding scheme and places the converted bytes into an internal buffer. When the buffer fills up, it is written to the underlying byte stream.

`public void write(char[] cbuf, int off, int len) throws IOException`

Parameters

`cbuf`

An array of characters to write.

`off`

An offset into the character array.

`len`

The number of characters to write.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`Writer.write(char[], int, int)`

Description

This method converts `len` characters from the array `cbuf` to bytes, starting at offset `off`, using the current encoding scheme. The method places the converted bytes into an internal buffer. When the buffer fills up, it is written to the underlying byte stream.

public void write(String str, int off, int len) throws IOException

Parameters

`str`

The string to be written.

`off`

An offset into start in the string.

`len`

The number of characters to write.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`Writer.write(String, int, int)`

Description

This method converts `len` characters from the string `str` to bytes, starting at offset `off`, using the current encoding scheme. The method places the converted bytes into an internal buffer. When the buffer fills up, it is written to the underlying byte stream.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`	`write(char[])`	`Writer`
`write(String)`	`Writer`		

See Also

`BufferedWriter`, `FileWriter`, `IOException`, `OutputStream`, `UnsupportedEncodingException`, `Writer`

PipedInputStream

Name

PipedInputStream

Synopsis

Class Name:

`java.io.PipedInputStream`

Superclass:

`java.io.InputStream`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The `PipedInputStream` class represents half of a communication pipe; a `PipedInputStream` must be connected to a PipedOutputStream. When the two halves of a communication pipe are connected, data written to the PipedOutputStream can be read from the PipedInputStream. The communication pipe formed by a PipedInputStream and a PipedOutputStream should be used to communicate between threads. If both ends of a pipe are used by the same thread, the thread can hang.

Class Summary

```
public class java.io.PipedInputStream extends java.io.InputStream {
  // Variables
  protected byte[] buffer;                           // New in 1.1
  protected int in;                                  // New in 1.1
  protected int out;                                 // New in 1.1
  protected final static int PIPE_SIZE;              // New in 1.1
  // Constructors
  public PipedInputStream();
  public PipedInputStream(PipedOutputStream src);
  // Public Instance Methods
  public synchronized int available();               // New in 1.1
  public void close();
  public void connect(PipedOutputStream src);
  public synchronized int read();
  public synchronized int read(byte[] b, int off, int len);

  // Protected Instance Methods
  protected synchronized void receive(int b);        // New in 1.1
}
```

Variables

buffer

protected byte[] buffer

Availability

> New as of JDK 1.1

Description

> The internal data buffer. The buffer receives data from the connected PipedOutputStream and supplies data for the calls to read().

in

protected int in

Availability

New as of JDK 1.1

Description

An index into the buffer that points to the byte after the last byte of valid data. A value of `-1` indicates that the buffer is empty.

out

protected int out

Availability

New as of JDK 1.1

Description

An index into the buffer that points to the next byte that will be returned by `read()`.

PIPE_SIZE

public final static int PIPE_SIZE = 1024

Availability

New as of JDK 1.1

Description

The size of the internal data buffer. The buffer receives data from the connected `PipedOutputStream` and supplies data for the calls to `read()`.

Constructors

PipedInputStream

public PipedInputStream()

Description

This constructor creates a `PipedInputStream` that is not connected to a `PipedOutputStream`. The created object must be connected to a `PipedOutputStream` before it can be used.

public PipedInputStream(PipedOutputStream src) throws IOException

Parameters

`src`

The `PipedOutputStream` to connect.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This constructor creates a `PipedInputStream` that receives data from the given `PipedOutputStream`.

Public Instance Methods

available

public synchronized int available() throws IOException

Availability

New as of JDK 1.1

Returns

The number of bytes that can be read without blocking.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`InputStream.available()`

Description

This method returns the number of bytes that can be read without having to wait for more data to become available. More data becomes available in the `PipedInputStream` when data is written to the connected `PipedOutputStream`.

close

public void close() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`InputStream.close()`

Description

This method closes the stream and releases the system resources that are associated with it.

connect

public void connect(PipedOutputStream src) throws IOException

Parameters

`src`

The `PipedOutputStream` to connect.

Throws

`IOException`

If another `PipedOutputStream` is already connected to this `PipedInputStream`.

Description

This method connects the given `PipedOutputStream` to this `PipedInputStream` object. If there is already a connected `PipedOutputStream`, an exception is thrown.

read

public synchronized int read() throws IOException

Returns

The next byte of data or `-1` if the end of the stream is encountered.

Throws

`IOException`

If the pipe is broken. In other words, if this `PipedInputStream` is closed or if the connected `PipedOutputStream` is dead.

`InterruptedIOException`

While this method is waiting for input, if the `interrupted()` method of the thread that invoked this method is called.

Overrides

`InputStream.read()`

Description

This method returns the next byte from the pipe buffer. If the buffer is empty, the method waits until data is written to the connected `PipedOutputStream`. The method blocks until the byte is read, the end of the stream is encountered, or an exception is thrown.

`public synchronized int read(byte b[], int off, int len) throws IOException`

Parameters

`b`

An array of bytes to be filled.

`off`

An offset into the byte array.

`len`

The number of bytes to read.

Returns

The actual number of bytes read or `-1` if the end of the stream is encountered immediately.

Throws

`IOException`

If the pipe is broken. In other words, if this `PipedInputStream` is closed or if the connected `PipedOutputStream` is dead.

`InterruptedIOException`

While this method is waiting for buffer space to become available, if the `interrupted()` method of the thread that invoked this method is called.

Overrides

`InputStream.read(byte[], int, int)`

Description

This method copies bytes from the pipe buffer into the given array `b`, starting at index `off` and continuing for `len` bytes. If there is at least one byte in the buffer, the method returns as many bytes as are in the buffer (up to `len`). If the buffer is empty, the method blocks until data is written to the connected `PipedOutputStream`.

Protected Instance Methods

receive

protected synchronized void receive(int b) throws IOException

Availability

New as of JDK 1.1

Parameters

`b`

The byte being received.

Throws

`IOException`

If the pipe is broken. In other words, if this `PipedInputStream` is closed.

Description

This method is called by the connected `PipedOutputStream` object to provide the given value as a byte of input to this `PipedInputStream` object.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`mark(int)`	`InputStream`
`markSupported()`	`InputStream`	`notify()`	`Object`
`notifyAll()`	`Object`	`read(byte[])`	`InputStream`
`reset()`	`InputStream`	`skip(long)`	`InputStream`
`toString()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`InputStream`, `IOException`, `PipedOutputStream`

PREVIOUS | HOME | NEXT
OutputStreamWriter | BOOK INDEX | PipedOutputStream

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

PipedOutputStream

Name

PipedOutputStream

Synopsis

Class Name:

`java.io.PipedOutputStream`

Superclass:

`java.io.OutputStream`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The PipedOutputStream class represents half of a communication pipe; a PipedOutputStream must be connected to a PipedOutputStream. When the two halves of a communication pipe are connected, data written to the PipedOutputStream can be read from the PipedInputStream. The communication pipe formed by a PipedOutputStream and a PipedInputStream should be used to communicate between threads. If both ends of a pipe are used by the same thread, the thread can hang.

Class Summary

```
public class java.io.PipedOutputStream extends java.io.OutputStream {
  // Constructors
  public PipedOutputStream();
  public PipedOutputStream(PipedInputStream snk);
  // Instance Methods
  public void close();
  public void connect(PipedInputStream snk);
  public synchronized void flush();               // New in 1.1
  public void write(int b);
  public void write(byte[] b, int off, int len);
}
```

Constructors

PipedOutputStream

public PipedOutputStream()

Description

> This constructor creates a PipedOutputStream that is not connected to a PipedInputStream. The created object must be connected to a PipedInputStream before it can be used.

public PipedOutputStream(PipedInputStream snk)

Parameters

`snk`

The `PipedInputStream` to connect.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This constructor creates a `PipedOutputStream` that sends data to the given `PipedInputStream`.

Instance Methods

close

public void close() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`OutputStream.close()`

Description

This method closes the stream and releases the system resources that are associated with it.

connect

public void connect(PipedInputStream snk) throws IOException

Parameters

`snk`

The `PipedInputStream` to connect.

Throws

`IOException`

If another `PipedInputStream` is already connected to this `PipedOutputStream` or this `PipedOutputStream` is already connected.

Description

This method connects this `PipedOutputStream` object to the given `PipedInputStream`. If this `PipedOutputStream` or `snk` is already connected, an exception is thrown.

flush

public synchronized void flush() throws IOException

Availability

New as of JDK 1.1

Throws

`IOException`

If any kind of I/O error occurs.

`InterruptedIOException`

While this method is waiting for buffer space to become available, if the `interrupted()` method of the thread that invoked this method is called.

Overrides

`OutputStream.flush()`

Description

This method flushes the stream, which tells the connected `PipedInputStream` to notify its readers to read any available data.

write

public void write(int b) throws IOException

Parameters

`b`

The value to write.

Throws

`IOException`

If any kind of I/O error occurs.

`InterruptedIOException`

While this method is waiting for buffer space to become available, if the `interrupted()` method of the thread that invoked this method is called.

Overrides

`OutputStream.write(int)`

Description

This method writes a byte of output. The method passes the given value directly to the connected `PipedInputStream`.

public void write(byte b[], int off, int len) throws IOException

Parameters

`b`

An array of bytes to write to the stream.

`off`

An offset into the byte array.

`len`

The number of bytes to write.

Throws

`IOException`

If any kind of I/O error occurs.

`InterruptedIOException`

While this method is waiting for buffer space to become available, if the `interrupted()` method of the thread that invoked this method is called.

Overrides

`OutputStream.write(byte[], int, int)`

Description

This method writes `len` bytes of output from the given array, starting at offset `off`. The method passes the given data to the connected `PipedInputStream`.

Inherited Methods

Method	**Inherited From**	**Method**	**Inherited From**
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`

`wait(long, int)`	`Object`	`write(byte[])`	`OutputStream`

See Also

`IOException`, `OutputStream`, `PipedInputStream`

PREVIOUS
PipedInputStream

HOME
BOOK INDEX

NEXT
PipedReader

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

PipedReader

Name

PipedReader

Synopsis

Class Name:

    `java.io.PipedReader`

Superclass:

    `java.io.Reader`

Immediate Subclasses:

    None

Interfaces Implemented:

    None

Availability:

    New as of JDK 1.1

Description

The `PipedReader` class represents half of a communication pipe; a `PipedReader` must be connected to a `PipedWriter`. When the two halves of a communication pipe are connected, data written to the `PipedWriter` can be read from the `PipedReader`. The communication pipe formed by a `PipedReader` and a `PipedWriter` should be used to communicate between threads. If both ends of a pipe are used by the same thread, the thread can hang.

The `PipedReader` class is the character-based equivalent of the byte-based `PipedInputStream`.

Class Summary

```
public class java.io.PipedReader extends java.io.Reader {
  // Constructors
  public PipedReader();
  public PipedReader(PipedWriter src);
  // Instance Methods
  public void close();
  public void connect(PipedWriter src);
  public int read(char[] cbuf, int off, int len);
}
```

Constructors

PipedReader

public PipedReader ()

Description

> This constructor creates a `PipedReader` that is not connected to a `PipedWriter`. The created object must be connected to a `PipedWriter` before it can be used.

public PipedReader(PipedWriter src) throws IOException

Parameters

> `src`

The `PipedWriter` to connect.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This constructor creates a `PipedReader` that receives data from the given `PipedWriter`.

Instance Methods

close

public void close() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`Reader.close()`

Description

This method closes the reader and releases the system resources that are associated with it.

connect

public void connect(PipedWriter src) throws IOException

Parameters

`src`

The `PipedWriter` to connect.

Throws

`IOException`

If another `PipedWriter` is already connected to this `PipedReader`.

Description

This method connects the given `PipedWriter` to this `PipedReader` object. If there is already a connected `PipedWriter`, an exception is thrown.

read

public int read(char[] cbuf, int off, int len) throws IOException

Parameters

`cbuf`

An array of characters to be filled.

`off`

An offset into the array.

`len`

The number of characters to read.

Returns

The actual number of characters read or `-1` if the end of the stream is encountered immediately.

Throws

`IOException`

If the pipe is broken. In other words, if this `PipedReader` is closed or if the connected

PipedWriter is dead.

InterruptedIOException

While this method is waiting for input, if the interrupted() method of the thread that invoked this method is called.

Overrides

Reader.read(char[], int, int)

Description

This method copies characters from the pipe buffer into the given array cbuf, starting at index off and continuing for len characters. If there is at least one character in the buffer, the method returns as many characters as are in the buffer (up to len). If the buffer is empty, the method blocks until data is written to the connected PipedWriter.

Inherited Methods

Method	Inherited From	Method	Inherited From
clone()	Object	equals(Object)	Object
finalize()	Object	getClass()	Object
hashCode()	Object	mark(int)	Reader
markSupported()	Reader	notify()	Object
notifyAll()	Object	read()	Reader
read(char[])	Reader	reset()	Reader
skip(long)	Reader	toString()	Object
wait()	Object	wait(long)	Object
wait(long, int)	Object		

See Also

IOException, PipedInputStream, PipedWriter, Reader

PipedWriter

Name

PipedWriter

Synopsis

Class Name:

`java.io.PipedWriter`

Superclass:

`java.io.Writer`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `PipedWriter` class represents half of a communication pipe; a `PipedReader` must be connected to a `PipedWriter`. When the two halves of a communication pipe are connected, data written to the `PipedWriter` can be read from the `PipedReader`. The communication pipe formed by a `PipedWriter` and a `PipedReader` should be used to communicate between threads. If both ends of a pipe are used by the same thread, the thread can hang.

The `PipedWriter` class is the character-based equivalent of the byte-based `PipedOutputStream`.

Class Summary

```
public class java.io.PipedWriter extends java.io.Writer {
  // Constructors
  public PipedWriter();
  public PipedWriter(PipedReader sink);
  // Instance Methods
  public void close();
  public void connect(PipedReader sink);
  public void flush();
  public void write(char[] cbuf, int off, int len;
}
```

Constructors

PipedWriter

public PipedWriter()

Description

> This constructor creates a `PipedWriter` that is not connected to a `PipedReader`. The created object must be connected to a `PipedReader` before it can be used.

public PipedWriter(PipedReader sink)

Parameters

> `sink`

The `PipedReader` to connect.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This constructor creates a `PipedWriter` that sends data to the given `PipedReader`.

Instance Methods

close

public void close() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`Writer.close()`

Description

This method closes the writer and releases the system resources that are associated with it.

connect

public void connect(PipedReader sink) throws IOException

Parameters

`sink`

The `PipedReader` to connect.

Throws

`IOException`

If another `PipedReader` is already connected to this `PipedWriter` or this `PipedWriter` is already connected.

Description

This method connects this `PipedWriter` object to the given `PipedReader`. If this `PipedWriter` or `sink` is already connected, an exception is thrown.

flush

public void flush() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

`InterruptedIOException`

While this method is waiting for buffer space to become available, if the `interrupted()` method of the thread that invoked this method is called.

Overrides

`Writer.flush()`

Description

This method flushes the writer, which tells the connected `PipedReader` to notify its readers to read any available data.

write

```
public void write(char[] cbuf, int off, int len) throws IOException
```

Parameters

`cbuf`

An array of characters to write to the stream.

`off`

An offset into the character array.

`len`

The number of characters to write.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`Writer.write(char[], int, int)`

Description

This method writes `len` characters of output from the given array, starting at offset `off`. The method passes the given data to the connected `PipedReader`.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`

`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`	`write(int)`	`Writer`
`write(char[])`	`Writer`	`write(String)`	`Writer`
`write(String, int, int)`	`Writer`		

See Also

`IOException`, `PipedOutputStream`, `PipedReader`, `Writer`

PREVIOUS PipedReader

HOME

BOOK INDEX

NEXT PrintStream

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

PrintStream

Name

PrintStream

Synopsis

Class Name:

```
java.io.PrintStream
```

Superclass:

```
java.io.FilterOutputStream
```

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The PrintStream class provides support for writing string representations of primitive data types and objects to an underlying output stream. As of JDK 1.1, PrintStream uses the system's default encoding scheme to convert characters to bytes and uses the system's own specific line separator, rather than the newline character, for separating lines of text. Although this class is not officially deprecated, its constructors are, and you should use PrintWriter instead of PrintStream in new code.

Prior to JDK 1.1, PrintStream did not handle Unicode characters. Any PrintStream methods that wrote characters only wrote the low eight bits of each character. In addition, prior to JDK 1.1, PrintStream used the newline character to separate lines of text, regardless of the platform. These problems have been corrected as of JDK 1.1.

All of the methods of PrintStream that write multiple times to the underlying output stream handle synchronization internally, so that PrintStream objects are thread-safe.

A PrintStream object is often used to write to a BufferedOutputStream object. Note that you can specify that the PrintStream be flushed every time it writes the line separator or the newline character by using the constructor that takes a boolean argument.

PrintStream objects are often used to report errors. For this reason, the methods of this class do not throw exceptions. Instead, the methods catch any exceptions thrown by any downstream OutputStream objects and set an internal flag, so that the object can remember that a problem occurred. You can query the internal flag by calling the checkError() method.

Class Summary

```
public class java.io.PrintStream extends java.io.FilterOutputStream {
  // Constructors
  public PrintStream(OutputStream out);           // Deprecated in 1.1
  public PrintStream(OutputStream out,
                     boolean autoFlush);          // Deprecated in 1.1
  // Public Instance Methods
  public boolean checkError();
  public void close();
  public void flush();
  public void print(boolean b);
  public void print(char c);
  public void print(char[] s);
  public void print(double d);
  public void print(float f);
  public void print(int i);
  public void print(long l);
  public void print(String s);
  public void print(Object obj);
  public void println();
  public void println(boolean b);
  public void println(char c);
  public void println(char[] s);
  public void println(double d);
  public void println(float f);
  public void println(int i);
  public void println(long l);
  public void println(Object obj);
  public void println(String s);
  public void write(int b);
  public void write(byte[] buf, int off, int len);
  // Protected Instance Methods
  protected void setError();                      // New in 1.1
}
```

Constructors

PrintStream

public PrintStream(OutputStream out)

Availability

Deprecated as of JDK 1.1

Parameters

`out`

The output stream to use.

Description

This constructor creates a `PrintStream` object that sends output to the given `OutputStream`.

public PrintStream(OutputStream out, boolean autoflush)

Availability

Deprecated as of JDK 1.1

Parameters

`out`

The output stream to use.

`autoflush`

A `boolean` value that indicates whether or not the print stream is flushed every time a newline is output.

Description

This constructor creates a `PrintStream` object that sends output to the given `OutputStream`. If `autoflush` is `true`, every time the `PrintStream` object writes a newline character or line separator, it calls its `flush()` method. Note that this is different than with a `PrintWriter` object, which only calls its `flush()` method when a `println()` method is called.

Public Instance Methods

checkError

public boolean checkError()

Returns

`true` if any error has occurred; `false` otherwise.

Description

This method flushes any buffered output and returns `true` if any error has occurred. Once the error flag for a `PrintStream` object has been set, it is never cleared.

close

public void close()

Overrides

`FilterOutputStream.close()`

Description

This method closes this print stream and releases any resources associated with the object. The method does this by calling the `close()` method of the underlying output stream and catching any exceptions that are thrown.

flush

public void flush()

Overrides

`FilterOutputStream.flush()`

Description

This method flushes this print stream, forcing any bytes that may be buffered to be written to the underlying output stream. The method does this by calling the `flush()` method of the underlying output stream and catching any exceptions that are thrown.

print

public void print(boolean b)

Parameters

`b`

The `boolean` value to print.

Description

This method writes `"true"` to the underlying output stream if `b` is true; otherwise it writes `"false"`.

public void print(char c)

Parameters

`c`

The `char` value to print.

Description

This method writes the given character to the underlying output stream.

public void print(char[] s)

Parameters

`s`

The `char` array to print.

Description

This method writes the characters in the given array to the underlying output stream.

public void print(double d)

Parameters

`d`

The `double` value to print.

Description

This method writes a string representation of the given `double` value to the underlying output stream. The string representation is identical to the one returned by calling `Double.toString(d)`.

public void print(float f)

Parameters

`f`

The `float` value to print.

Description

This method writes a string representation of the given `float` value to the underlying output stream. The string representation is identical to the one returned by calling `Float.toString(f)`.

public void print(int i)

Parameters

`i`

The `int` value to print.

Description

This method writes a string representation of the given `int` value to the underlying output stream. The string representation is identical to the one returned by calling `Integer.toString(i)`.

public void print(long l)

Parameters

`l`

The `long` value to print.

Description

This method writes a string representation of the given `long` value to the underlying output stream. The string representation is identical to the one returned by calling `Long.toString(l)`.

public void print(Object obj)

Parameters

`obj`

The `Object` to print.

Description

This method writes the string representation of the given `Object` to the underlying output stream. The string representation is that returned by calling the `toString()` method of `Object`.

public void print(String s)

Parameters

`s`

The `String` to print.

Description

This method writes the given `String` to the underlying output stream. If `String` is `null`, the method writes `"null"`.

println

public void println()

Description

This method writes a line separator to the underlying output stream.

public void println(boolean b)

Parameters

`b`

The `boolean` value to print.

Description

This method writes `"true"` to the underlying output stream if `b` is true, otherwise it writes `"false"`. In either case, the string is followed by a line separator.

public void println(char c)

Parameters

`c`

The `char` value to print.

Description

This method writes the given character, followed by a line separator, to the underlying output stream.

public void println(char[] s)

Parameters

`s`

The `char` array to print.

Description

This method writes the characters in the given array, followed by a line separator, to the underlying output stream.

public void println(double d)

Parameters

`d`

The `double` value to print.

Description

This method writes a string representation of the given `double` value, followed by a line separator, to the underlying output stream. The string representation is identical to the one returned by calling `Double.toString(d)`.

public void println(float f)

Parameters

`f`

The `float` value to print.

Description

This method writes a string representation of the given `float` value, followed by a line separator, to the underlying output stream. The string representation is identical to the one returned by calling `Float.toString(f)`.

public void println(int i)

Parameters

`i`

The `int` value to print.

Description

This method writes a string representation of the given `int` value, followed by a line separator, to the underlying output stream. The string representation is identical to the one returned by calling `Integer.toString(i)`.

public void println(long l)

Parameters

`l`

The `long` value to print.

Description

This method writes a string representation of the given `long` value, followed by a line separator, to the underlying output stream. The string representation is identical to the one returned by calling `Long.toString(l)`.

public void println(Object obj)

Parameters

`obj`

The `Object` to print.

Description

This method writes the string representation of the given `Object`, followed by a line separator, to the underlying output stream. The string representation is that returned by calling the `toString()` method of `Object`.

public void println(String s)

Parameters

`s`

The `String` to print.

Description

This method writes the given `String`, followed by a line separator, to the underlying output stream. If `String` is `null`, the method writes `"null"` followed by a line separator.

write

public void write(int b)

Parameters

`b`

The value to write to the stream.

Overrides

`FilterOutputStream.write(int)`

Description

This method writes the lowest eight bits of `b` to the underlying stream as a byte. The method does this by calling the `write()` method of the underlying output stream and catching any exceptions that are thrown. If necessary, the method blocks until the byte is written.

public void write(byte b[], int off, int len)

Parameters

`b`

An array of bytes to write to the stream.

`off`

An offset into the byte array.

`len`

The number of bytes to write.

Overrides

`FilterOutputStream.write(byte[], int, int)`

Description

This method writes the lowest eight bits of each of `len` bytes from the given array, starting `off` elements from the beginning of the array, to the underlying output stream. The method does this by calling `write(b, off, len)` for the underlying output stream and catching any exceptions that are thrown. If necessary, the method blocks until the bytes are written.

Protected Instance Methods

setError

protected void setError()

Availability

New as of JDK 1.1

Description

This method sets the error state of the `PrintStream` object to `true`. Any subsequent calls to `getError()` return `true`.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`	`write(byte[])`	`FilterOutputStream`

See Also

`Double`, `FilterOutputStream`, `Float`, `Integer`, `Long`, `OutputStream`

PrintWriter

Name

PrintWriter

Synopsis

Class Name:

`java.io.PrintWriter`

Superclass:

`java.io.Writer`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `PrintWriter` class provides support for writing string representations of primitive data types and objects to an underlying output stream. `PrintWriter` uses the system's default encoding scheme to convert characters to bytes. `PrintWriter` also uses the system's own specific line separator, rather than the newline character, for separating lines of text. This line separator is equivalent to the value returned by:

```
System.getProperty("line.separator")
```

A `PrintWriter` object can be created using a `Writer` object or an `OutputStream` object as its underlying stream. When a `PrintWriter` is created using an `OutputStream`, the constructor creates the intermediate `OutputStreamWriter` that handles the conversion of characters to bytes using the default character encoding.

All of the methods of `PrintWriter` that write multiple times to the underlying output stream handle synchronization internally, so that `PrintWriter` objects are thread-safe.

A `PrintWriter` object is often used to write to a `BufferedWriter` object. Note that you can specify that the `PrintWriter` should be flushed every time a `println()` method is called by using a constructor that takes a `boolean` argument.

`PrintWriter` objects are often used to report errors. For this reason, the methods of this class do not throw exceptions. Instead, the methods catch any exceptions thrown by any downstream `OutputStream` or `Writer` objects and set an internal flag, so that the object can remember that a problem occurred. You can query the internal flag by calling the `checkError()` method.

Class Summary

```
public class java.io.PrintWriter extends java.io.Writer {
  // Constructors
  public PrintWriter(OutputStream out);
  public PrintWriter(OutputStream out, boolean autoFlush);
  public PrintWriter(Writer out);
  public PrintWriter(Writer out, boolean autoFlush);
  // Public Instance Methods
  public boolean checkError();
  public void close();
  public void flush();
  public void print(boolean b);
  public void print(char c);
  public void print(char[] s);
  public void print(double d);
  public void print(float f);
  public void print(int i);
  public void print(long l);
  public void print(Object obj);
  public void print(String s);
  public void println();
  public void println(boolean b);
  public void println(char c);
  public void println(char[] s);
  public void println(double d);
  public void println(float f);
  public void println(int i);
  public void println(long l);
  public void println(Object obj);
  public void println(String s);
  public void write(int c);
  public void write(char[] buf);
  public void write(char[] buf, int off, int len);
  public void write(String s);
  public void write(String s, int off, int len);

  // Protected Instance Methods
  protected void setError();
}
```

Constructors

PrintWriter

public PrintWriter(OutputStream out)

Parameters

`out`

The output stream to use.

Description

This constructor creates a `PrintWriter` object that sends output to the given `OutputStream`. The constructor creates the intermediate `OutputStreamWriter` that converts characters to bytes using the default character encoding.

public PrintWriter(OutputStream out, boolean autoFlush)

Parameters

`out`

The output stream to use.

`autoFlush`

A `boolean` value that indicates whether or not the print stream is flushed every time a `println()` method is called.

Description

This constructor creates a `PrintWriter` object that sends output to the given `OutputStream`. The constructor creates the intermediate `OutputStreamWriter` that converts characters to bytes using the default character encoding. If `autoFlush` is `true`, every time a `println()` method is called, the `PrintWriter` object calls its `flush()` method. This behavior is different from that of a `PrintStream` object, which calls its `flush()` method each time a line separator or newline character is written.

public PrintWriter(Writer out)

Parameters

`out`

The output stream to use.

Description

This constructor creates a `PrintWriter` object that sends output to the given `Writer`.

public PrintStream(Writer out, boolean autoFlush)

Parameters

`out`

The output stream to use.

`autoFlush`

A `boolean` value that indicates whether or not the print stream is flushed every time a `println()` method is called.

Description

This constructor creates a `PrintWriter` object that sends output to the given `Writer`. If `autoFlush` is `true`, every time a `println()` method is called, the `PrintWriter` object calls its `flush()` method. Note that this behavior is different from that of a `PrintStream` object, which calls its `flush()` method every time a newline character or line separator is written.

Public Instance Methods

checkError

public boolean checkError()

Returns

`true` if any error has occurred; `false` otherwise.

Description

This method flushes any buffered output and returns `true` if an error occurs. Once the error flag for a `PrintWriter` object is set, it's never cleared.

close

public void close()

Overrides

`Writer.close()`

Description

This method closes this print stream and releases any resources associated with the object. The method does this by calling the `close()` method of the underlying output stream and catching any exceptions that are thrown.

flush

public void flush()

Overrides

`Writer.flush()`

Description

This method flushes this print stream, forcing any bytes that may be buffered to be written to the underlying output stream. The method does this by calling the `flush()` method of the underlying output stream and catching any exceptions that are thrown.

print

public void print(boolean b)

Parameters

`b`

The `boolean` value to print.

Description

This method writes `"true"` to the underlying output stream if `b` is true; otherwise it writes `"false"`.

public void print(char c)

Parameters

`c`

The `char` value to print.

Description

This method writes the given character to the underlying output stream.

public void print(char[] s)

Parameters

`s`

The `char` array to print.

Description

This method writes the characters in the given array to the underlying output stream.

public void print(double d)

Parameters

`d`

The `double` value to print.

Description

This method writes a string representation of the given `double` value to the underlying output stream. The string representation is identical to the one returned by calling `Double.toString(d)`.

public void print(float f)

Parameters

`f`

The `float` value to print.

Description

This method writes a string representation of the given `float` value to the underlying output stream. The string representation is identical to the one returned by calling `Float.toString(f)`.

public void print(int i)

Parameters

`i`

The `int` value to print.

Description

This method writes a string representation of the given `int` value to the underlying output stream. The string representation is identical to the one returned by calling `Integer.toString(i)`.

public void print(long l)

Parameters

`l`

The `long` value to print.

Description

This method writes a string representation of the given `long` value to the underlying output stream. The string representation is identical to the one returned by calling `Long.toString(l)`.

public void print(Object obj)

Parameters

`obj`

The `Object` to print.

Description

> This method writes the string representation of the given `Object` to the underlying output stream. The string representation is that returned by calling the `toString()` method of `Object`.

public void print(String s)

Parameters

> `s`
>
> > The `String` to print.

Description

> This method writes the given `String` to the underlying output stream. If `String` is `null`, the method writes `"null"`.

println

public void println()

Description

> This method writes a line separator to the underlying output stream.

public void println(boolean b)

Parameters

> `b`
>
> > The `boolean` value to print.

Description

> This method writes `"true"` to the underlying output stream if `b` is true, otherwise it writes `"false"`. In either case, the string is followed by a line separator.

public void println(char c)

Parameters

`c`

The `char` value to print.

Description

This method writes the given character, followed by a line separator, to the underlying output stream.

public void println(char[] s)

Parameters

`s`

The `char` array to print.

Description

This method writes the characters in the given array, followed by a line separator, to the underlying output stream.

public void println(double d)

Parameters

`d`

The `double` value to print.

Description

This method writes a string representation of the given `double` value, followed by a line separator, to the underlying output stream. The string representation is identical to the one returned by calling `Double.toString(d)`.

public void println(float f)

Parameters

`f`

The `float` value to print.

Description

This method writes a string representation of the given `float` value, followed by a line separator, to the underlying output stream. The string representation is identical to the one returned by calling `Float.toString(f)`.

public void println(int i)

Parameters

`i`

The `int` value to print.

Description

This method writes a string representation of the given `int` value, followed by a line separator, to the underlying output stream. The string representation is identical to the one returned by calling `Integer.toString(i)`.

public void println(long l)

Parameters

`l`

The `long` value to print.

Description

This method writes a string representation of the given `long` value, followed by a line separator, to the underlying output stream. The string representation is identical to the one returned by calling `Long.toString(l)`.

public void println(Object obj)

Parameters

`obj`

The `Object` to print.

Description

This method writes the string representation of the given `Object`, followed by a line separator, to the underlying output stream. The string representation is that returned by calling the `toString()` method of `Object`.

public void println(String s)

Parameters

`s`

The `String` to print.

Description

This method writes the given `String`, followed by a line separator, to the underlying output stream. If `String` is `null`, the method writes `"null"` followed by a line separator.

write

public void write(int c)

Parameters

`c`

The value to write to the stream.

Overrides

`Writer.write(int)`

Description

This method writes the character specified by the lowest two bytes of the given integer `c` to the underlying stream. The method does this by calling the `write()` method of the underlying output stream and catching any exceptions that are thrown. If necessary, the method blocks until the character is written.

public void write(char[] buf)

Parameters

`buf`

An array of characters to write to the stream.

Overrides

`Writer.write(char[])`

Description

This method writes the given array of characters to the underlying output stream. The method does this by calling `write(buf, 0, buf.length)` for the underlying output stream and catching any exceptions that are thrown. If necessary, the method blocks until the characters are written.

public void write(char[] buf, int off, int len)

Parameters

`buf`

An array of characters to write to the stream.

`off`

An offset into the array.

`len`

The number of characters to write.

Overrides

`Writer.write(char[], int, int)`

Description

This method writes `len` characters from the given array, starting `off` elements from the beginning of the array, to the underlying output stream. The method does this by calling `write(buf, off, len)` for the underlying output stream and catching any exceptions that are thrown. If necessary, the method blocks until the characters are written.

public void write(String s)

Parameters

`s`

A `String` to write to the stream.

Overrides

`Writer.write(String)`

Description

This method writes the given `String` to the underlying output stream. The method does this by calling `write(s, 0, s.length)` for the underlying output stream and catching any exceptions that are thrown. If necessary, the method blocks until the `String` is written.

public void write(String s, int off, int len)

Parameters

`s`

A `String` to write to the stream.

`off`

An offset into the string.

`len`

The number of characters to write.

Overrides

`Writer.write(String, int, int)`

Description

This method writes `len` characters from the given `String`, starting `off` elements from the beginning of the string, to the underlying output stream. The method does this by calling `write(s, off, len)` for the underlying output stream and catching any exceptions that are thrown. If necessary, the method blocks until the characters of the `String` are written.

Protected Instance Methods

setError

protected void setError()

Description

This method sets the error state of the `PrintWriter` object to `true`. Any subsequent calls to `getError()` will return `true`.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Double`, `Float`, `Integer`, `Long`, `OutputStream`, `OutputStreamWriter`, `Writer`

PREVIOUS HOME NEXT
PrintStream BOOK INDEX PushbackInputStream

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

PREVIOUS

NEXT

PushbackInputStream

Name

PushbackInputStream

Synopsis

Class Name:

`java.io.PushbackInputStream`

Superclass:

`java.io.FilterInputStream`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The `PushbackInputStream` class represents a byte stream that allows data to be pushed back into the stream. In other words, after data has been read from a PushbackInputStream, it can be pushed back into the stream so that it can be reread. This functionality is useful for implementing things like parsers that need to read data and then return it to the input stream.

The PushbackInputStream has been enhanced as of JDK 1.1 to support a pushback buffer that is larger than one byte. Prior to JDK 1.1, the class supported only a one-byte buffer using the protected variable pushBack. As of 1.1, that variable has been replaced by the buf and pos variables.

Class Summary

```
public class java.io.PushbackInputStream extends java.io.FilterInputStream {
  // Variables
  protected byte[] buf;                            // New in 1.1
  protected int pos;                               // New in 1.1
  // Constructors
  public PushbackInputStream(InputStream in);
  public PushbackInputStream(InputStream in,
                             int size);            // New in 1.1
  // Instance Methods
  public int available();
  public boolean markSupported();
  public int read();
  public int read(byte[] b, int off, int len);
  public void unread(int b);
  public void unread(byte[] b);                    // New in 1.1
  public void unread(byte[] b, int off, int len); // New in 1.1
}
```

Variables

buf

protected byte[] buf

Availability

New as of JDK 1.1

Description

The buffer that holds data that has been pushed back.

pos

protected int pos

Availability

New as of JDK 1.1

Description

The position of pushed-back data in the buffer. When there is no pushed-back data, `pos` is `buf.length`. As data is pushed back, `pos` decreases. As pushed-back data is read, `pos` increases. When the pushback buffer is full, `pos` is 0.

Constructors

PushbackInputStream

public PushbackInputStream(InputStream in)

Parameters

`in`

The input stream to wrap.

Description

This constructor creates a `PushbackInputStream` that reads from the given `InputStream`, using a pushback buffer with the default size of one byte.

public PushBackInputStream(InputStream in, int size)

Availability

New as of JDK 1.1

Parameters

`in`

The input stream to wrap.

`size`

The size of the pushback buffer.

Description

This constructor creates a `PushbackInputStream` that reads from the given `InputStream`, using a pushback buffer of the given size.

Instance Methods

available

public int available() throws IOException

Returns

The number of bytes that can be read without blocking.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterInputStream.available()`

Description

This method returns the number of bytes that can be read without having to wait for more data to become available. This is `b` + `u`, where `b` is the number of bytes in the pushback buffer and `u` is the number of available bytes in the underlying stream.

markSupported

public boolean markSupported()

Returns

The `boolean` value `false`.

Overrides

`FilterInputStream.markSupported()`

Description

This method returns `false` to indicate that this class does not support `mark()` and `reset()`.

read

public int read() throws IOException

Returns

The next byte of data, or `-1` if the end of the stream is encountered.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterInputStream.read()`

Description

This method reads a byte of data. If there is any data in the pushback buffer, the method returns the next byte in the pushback buffer. Otherwise, it calls the `read()` method of the underlying stream. The method blocks until the byte is read, the end of the stream is encountered, or an exception is thrown.

public int read(byte b[], int off, int len) throws IOException

Parameters

`b`

An array of bytes to be filled from the stream.

`off`

An offset into the byte array.

`len`

The number of bytes to read.

Returns

The actual number of bytes read, or `-1` if the end of the stream is encountered immediately.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterInputStream.read(byte[], int, int)`

Description

This method copies bytes from the stream into the given array `b`, starting at index `off` and continuing for `len` bytes. If the array can be populated solely from the pushback buffer, the method returns immediately. Otherwise, the `read(byte[], int, int)` method of the underlying stream is called to make up the difference. The method blocks until some data is available.

unread

public void unread(int b) throws IOException

Parameters

`b`

The value to push back.

Throws

`IOException`

If the pushback buffer is full.

Description

This method puts the given byte into the pushback buffer.

public void unread(byte[] b) throws IOException

Availability

New as of JDK 1.1

Parameters

`b`

An array of bytes to push back.

Throws

`IOException`

If the pushback buffer is full.

Description

This method puts all of the bytes in the given array into the pushback buffer.

public void unread(byte[] b, int off, int len) throws IOException

Availability

New as of JDK 1.1

Parameters

`b`

An array of bytes to push back.

`off`

An offset into the array.

`len`

The number of bytes to push back.

Throws

`IOException`

If the pushback buffer is full.

Description

This method puts `len` bytes from the given array, starting at offset `off`, into the pushback buffer.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`close()`	`FilterInputStream`
`equals(Object)`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`hashCode()`	`Object`
`mark(int)`	`FilterInputStream`	`notify()`	`Object`
`notifyAll()`	`Object`	`read(byte[])`	`FilterInputStream`
`reset()`	`FilterInputStream`	`skip(long)`	`FilterInputStream`
`toString()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`FilterInputStream`, `InputStream`, `IOException`

PREVIOUS HOME NEXT
PrintWriter BOOK INDEX PushbackReader

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

PushbackReader

Name

PushbackReader

Synopsis

Class Name:

```
java.io.PushbackReader
```

Superclass:

```
java.io.FilterReader
```

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `PushbackReader` class represents a character stream that allows data to be pushed back into the stream. In other words, after data has been read from a `PushbackReader`, it can be pushed back into the stream so that it can be reread. This functionality is useful for implementing things like parsers that need to read data and then return it to the input stream. `PushbackReader` is the character-oriented equivalent of `PushbackInputStream`.

Class Summary

```
public class java.io.PushbackReader extends java.io.FilterReader {
  // Constructors
  public PushbackReader(Reader in);
  public PushbackReader(Reader in, int size);
  // Instance Methods
  public void close();
  public boolean markSupported();
  public int read();
  public int read(char[] cbuf, int off, int len);
  public boolean ready();
  public void unread(int c);
  public void unread(char[] cbuf);
  public void unread(char[] cbuf, int off, int len);
}
```

Constructors

PushbackReader

public PushbackReader(Reader in)

Parameters

`in`

The reader to wrap.

Description

This constructor creates a `PushbackReader` that reads from the given `Reader`, using a pushback buffer with the default size of one byte.

public PushbackReader(Reader in, int size)

Parameters

`in`

The reader to wrap.

`size`

The size of the pushback buffer.

Description

This constructor creates a `PushbackReader` that reads from the given `Reader`, using a pushback buffer of the given size.

Instance Methods

close

public void close() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterReader.close()`

Description

This method closes the reader and releases the system resources that are associated with it.

markSupported

public boolean markSupported()

Returns

The `boolean` value `false`.

Overrides

`FilterReader.markSupported()`

Description

This method returns `false` to indicate that this class does not support `mark()` and `reset()`.

read

public int read() throws IOException

Returns

The next character of data or `-1` if the end of the stream is encountered.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterReader.read()`

Description

This method reads a character of data. If there is any data in the pushback buffer, the method returns the next character in the pushback buffer. Otherwise, it calls the `read()` method of the underlying stream. The method blocks until the character is read, the end of the stream is encountered, or an exception is thrown.

public int read(char[] cbuf, int off, int len) throws IOException

Parameters

`cbuf`

An array of characters to be filled from the stream.

`off`

An offset into the array.

`len`

The number of characters to read.

Returns

The actual number of characters read or `-1` if the end of the stream is encountered immediately.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterReader.read(char[], int, int)`

Description

This method copies characters from the stream into the given array `cbuf`, starting at index `off` and continuing for `len` characters. If the array can be populated solely from the pushback buffer, the method returns immediately. Otherwise, the `read(char[], int, int)` method of the underlying stream is called to make up the difference. The method blocks until some data is available.

ready

public boolean ready() throws IOException

Returns

A `boolean` value that indicates whether the stream is ready to be read.

Throws

`IOException`

If the stream is closed.

Overrides

`FilterReader.ready()`

Description

If there is data in the pushback buffer, or if the underlying stream is ready, this method returns `true`. The underlying stream is ready if the next `read()` is guaranteed not to block.

unread

public void unread(int c) throws IOException

Parameters

`c`

The value to push back.

Throws

`IOException`

If the pushback buffer is full.

Description

This method puts the given character into the pushback buffer.

public void unread(char[] cbuf) throws IOException

Parameters

`cbuf`

An array of characters to push back.

Throws

`IOException`

If the pushback buffer is full.

Description

This method puts all of the characters in the given array into the pushback buffer.

public void unread(char[] cbuf, int off, int len) throws IOException

Parameters

`cbuf`

An array of characters to push back.

`off`

An offset into the array.

`len`

The number of characters to push back.

Throws

`IOException`

If the pushback buffer is full.

Description

This method puts `len` characters from the given array, starting at offset `off`, into the pushback buffer.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`mark(int)`	`FilterReader`
`notify()`	`Object`	`notifyAll()`	`Object`
`read(char[])`	`FilterReader`	`reset()`	`FilterReader`
`skip(long)`	`FilterReader`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`FilterReader`, `IOException`, `Reader`

PREVIOUS HOME NEXT
PushbackInputStream BOOK INDEX RandomAccessFile

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

RandomAccessFile

Name

RandomAccessFile

Synopsis

Class Name:

`java.io.RandomAccessFile`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

`java.io.DataInput`, `java.io.DataOutput`

Availability:

JDK 1.0 or later

Description

The RandomAccessFile class reads data from and writes data to a file. The file is specified using a File object or a String that represents a pathname. Both constructors take a mode parameter that specifies whether the file is being opened solely for reading, or for reading and writing. Each of the constructors can throw a SecurityException if the application does not have permission to access the specified file using the given mode.

Unlike FileInputStream and FileOutputStream, RandomAccessFile supports random access to the data in the file; the seek() method allows you to alter the current position of the file pointer to any location in the file. RandomAccessFile implements both the DataInput and DataOutput interfaces, so it supports reading and writing of all the primitive data types.

Class Summary

```
public class java.io.RandomAccessFile extends java.lang.Object
             implements java.io.DataInput, java.io.DataOutput {
  // Constructors
  public RandomAccessFile(File file, String mode);
  public RandomAccessFile(String name, String mode);
  // Instance Methods
  public native void close();
  public final FileDescriptor getFD();
  public native long getFilePointer();
  public native long length();
  public native int read();
  public int read(byte[] b);
  public int read(byte[] b, int off, int len);
  public final boolean readBoolean();
  public final byte readByte();
  public final char readChar();
  public final double readDouble();
  public final float readFloat();
  public final void readFully(byte[] b);
  public final void readFully(byte[] b, int off, int len);
  public final int readInt();
  public final String readLine();
  public final long readLong();
  public final short readShort();
  public final String readUTF();
  public final int readUnsignedByte();
  public final int readUnsignedShort();
  public native void seek(long pos);
  public int skipBytes(int n);
  public native void write(int b);
  public void write(byte[] b);
  public void write(byte[] b, int off, int len);
  public final void writeBoolean(boolean v);
  public final void writeByte(int v);
  public final void writeBytes(String s);
  public final void writeChar(int v);
  public final void writeChars(String s);
  public final void writeDouble(double v);
  public final void writeFloat(float v);
  public final void writeInt(int v);
  public final void writeLong(long v);
  public final void writeShort(int v);
  public final void writeUTF(String str);
}
```

Constructors

RandomAccessFile

public RandomAccessFile(File file, String mode) throws IOException

Parameters

`file`

The file to be accessed.

`mode`

The mode of access to the file: either `"r"` for read access or `"rw"` for read/write access.

Throws

`IOException`

If any kind of I/O error occurs.

`IllegalArgumentException`

If `mode` is not `"r"` or `"rw"`.

`SecurityException`

If the application does not have permission to read the named file, or if `mode` is `"rw"` and the application does not have permission to write to the named file.

Description

This constructor creates a `RandomAccessFile` to access the specified `File` in the specified mode.

public RandomAccessFile(String name, String mode) throws IOException

Parameters

`name`

A `String` that contains the pathname of the file to be accessed. The path must conform to the requirements of the native operating system.

`mode`

The mode of access to the file: either `"r"` for read access or `"rw"` for read/write access.

Throws

`IOException`

If any kind of I/O error occurs.

`IllegalArgumentException`

If `mode` is not `"r"` or `"rw"`.

`SecurityException`

If the application does not have permission to read the named file, or if `mode` is `"rw"` and the application does not have permission to write to the named file.

Description

This constructor creates a `RandomAccessFile` to access the file with the specified `name` in the specified mode.

Instance Methods

close

public native void close() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method closes the file and releases the system resources that are associated with it.

getFD

public final FileDescriptor getFD() throws IOException

Returns

The file descriptor for the file that supplies data for this object.

Throws

`IOException`

If there is no `FileDescriptor` associated with this object.

Description

This method returns the file descriptor associated with this `RandomAccessFile`.

getFilePointer

public native long getFilePointer() throws IOException

Returns

> The current position in the file.

Throws

> `IOException`
>
> > If any kind of I/O error occurs.

Description

> This method returns the current position in the file. The position is the offset, in bytes, from the beginning of the file where the next read or write operation occurs.

length

public native long length() throws IOException

Returns

> The length of the file.

Throws

> `IOException`
>
> > If any kind of I/O error occurs.

Description

> This method returns the length of the file in bytes.

read

public native int read() throws IOException

Returns

The next byte or `-1` if the end of file is encountered.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method reads the next byte from the file. The method blocks until the byte is read, the end of the file is encountered, or an exception is thrown.

public int read(byte b[]) throws IOException

Parameters

`b`

An array of bytes to be filled from the stream.

Returns

The number of bytes read or `-1` if the end of file is encountered immediately.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method reads bytes from the file into the given array. The method reads up to `b.length` bytes of data from the stream. The method blocks until there is some data available.

public int read(byte b[], int off, int len) throws IOException

Parameters

`b`

An array of bytes to be filled.

`off`

An offset into the array.

`len`

The number of bytes to read.

Returns

The number of bytes read or `-1` if the end of file is encountered immediately.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method reads up to `len` bytes from the file into the given array, starting at index `off`. The method blocks until there is some input available.

readBoolean

public final boolean readBoolean() throws IOException

Returns

The `boolean` value read from the file.

Throws

`EOFException`

If the end of the file is encountered.

`IOException`

If any other kind of I/O error occurs.

Implements

`DataInput.readBoolean()`

Description

This method reads a byte as a `boolean` value from the file. A byte that contains a zero is read as `false`. A byte that contains any other value is read as `true`. The method blocks until the byte is read, the end of the file is encountered, or an exception is thrown.

readByte

public final byte readByte() throws IOException

Returns

The `byte` value read from the file.

Throws

`EOFException`

If the end of the file is encountered.

`IOException`

If any other kind of I/O error occurs.

Implements

`DataInput.readByte()`

Description

This method reads a signed 8-bit value, a `byte`, from the file. The method blocks until the byte is read, the end of the file is encountered, or an exception is thrown.

readChar

public final char readChar() throws IOException

Returns

The `char` value read from the file.

Throws

`EOFException`

If the end of the file is encountered.

`IOException`

If any other kind of I/O error occurs.

Implements

`DataInput.readChar()`

Description

This method reads a 16-bit Unicode character from the file. The method reads two bytes from the file and then creates a `char` value using the first byte read as the most significant byte. The method blocks until the two bytes are read, the end of the file is encountered, or an exception is thrown.

readDouble

public final double readDouble() throws IOException

Returns

The `double` value read from the file.

Throws

`EOFException`

If the end of the file is encountered.

`IOException`

If any other kind of I/O error occurs.

Implements

`DataInput.readDouble()`

Description

This method reads a 64-bit `double` quantity from the file. The method reads a `long` value from the file as if using the `readLong()` method. The `long` value is then converted to a `double` using the `longBitsToDouble()` method in `Double`. The method blocks until the necessary eight bytes are read, the end of the file is encountered, or an exception is thrown.

readFloat

public final float readFloat() throws IOException

Returns

The `float` value read from the file.

Throws

`EOFException`

If the end of the file is encountered.

`IOException`

If any other kind of I/O error occurs.

Implements

`DataInput.readFloat()`

Description

This method reads a 32-bit `float` quantity from the file. The method reads an `int` value from the file as if using the `readInt()` method. The `int` value is then converted to a `float` using the `intBitsToFloat()` method in `Float`. The method blocks until the necessary four bytes are read, the end of the file is encountered, or an exception is thrown.

readFully

public final void readFully(byte b[]) throws IOException

Parameters

`b`

The array to fill.

Throws

`EOFException`

If the end of the file is encountered.

`IOException`

If any other kind of I/O error occurs.

Implements

`DataInput.readFully(byte[])`

Description

This method reads bytes into the given array `b` until the array is full. The method reads repeatedly from the file to fill the array. The method blocks until all of the bytes are read, the end of the file is encountered, or an exception is thrown.

public final void readFully(byte b[], int off, int len) throws IOException

Parameters

`b`

The array to fill.

`off`

An offset into the array.

`len`

The number of bytes to read.

Throws

`EOFException`

If the end of the file is encountered.

`IOException`

If any other kind of I/O error occurs.

Implements

`DataInput.readFully(byte[], int, int)`

Description

This method reads `len` bytes into the given array, starting at offset `off`. The method reads repeatedly from the file to fill the array. The method blocks until all of the bytes are read, the end of the file is encountered, or an exception is thrown.

readInt

public final int readInt() throws IOException

Returns

The `int` value read from the stream.

Throws

`EOFException`

If the end of the file is encountered.

`IOException`

If any other kind of I/O error occurs.

Implements

`DataInput.readInt()`

Description

This method reads a signed 32-bit `int` quantity from the file. The method reads four bytes from the file and then creates an `int` quantity, using the first byte read as the most significant byte. The method blocks until the four bytes are read, the end of the file is encountered, or an exception is thrown.

readLine

public final String readLine() throws IOException

Returns

A `String` that contains the line read from the stream.

Throws

`EOFException`

If the end of the file is encountered.

`IOException`

If any other I/O error occurs.

Implements

`DataInput.readLine()`

Description

This method reads the next line of text from the file. The method reads bytes of data from the file until it encounters a line terminator. A line terminator is a carriage return (`'\r'`), a newline character (`'\n'`), a carriage return immediately followed by a newline character, or the end of the file. The method blocks until a line terminator is read, the end of the file is encountered, or an exception is thrown.

The method does not convert bytes to characters correctly.

readLong

public final long readLong() throws IOException

Returns

The `long` value read from the stream.

Throws

`EOFException`

If the end of the file is encountered.

`IOException`

If any other kind of I/O error occurs.

Implements

`DataInput.readLong()`

Description

This method reads a signed 64-bit `long` quantity from the file. The method reads eight bytes from the file and then creates a `long` quantity, using the first byte read as the most significant byte. The method blocks until the eight bytes are read, the end of the file is encountered, or an exception is thrown.

readShort

public final short readShort() throws IOException

Returns

The `short` value read from the stream.

Throws

`EOFException`

If the end of the file is encountered.

`IOException`

If any other kind of I/O error occurs.

Implements

`DataInput.readShort()`

Description

This method reads a signed 16-bit `short` quantity from the file. The method reads two bytes from the file and then creates a `short` quantity, using the first byte read as the most significant byte. The method blocks until the two bytes are read, the end of the file is encountered, or an exception is thrown.

readUnsignedByte

public final int readUnsignedByte() throws IOException

Returns

The unsigned `byte` value read from the stream.

Throws

`EOFException`

If the end of the file is encountered.

`IOException`

If any other kind of I/O error occurs.

Returns

Implements

`DataInput.readUnsignedByte()`

Description

This method reads an unsigned 8-bit quantity from the file. The method reads a byte from the file and returns that byte. The method blocks until the byte is read, the end of the file is encountered, or an exception is thrown.

readUnsignedShort

public final int readUnsignedShort() throws IOException

Returns

The unsigned `short` value read from the stream.

Throws

`EOFException`

If the end of the file is encountered.

`IOException`

If any other kind of I/O error occurs.

Implements

`DataInput.readUnsignedShort()`

Description

This method reads an unsigned 16-bit quantity from the file. The method reads two bytes from the file and creates an unsigned `short` quantity using the first byte read as the most significant byte. The method blocks until the two bytes are read, the end of the file is encountered, or an exception is thrown.

readUTF

public final String readUTF() throws IOException

Returns

The `String` read from the stream.

Throws

`EOFException`

If the end of the file is encountered.

`IOException`

If any other kind of I/O error occurs.

`UTFDataFormatException`

If the bytes do not represent a valid UTF-8 encoding.

Implements

`DataInput.readUTF()`

Description

This method reads a UTF-8 encoded string from the file. The method reads the first two bytes from the file as unsigned `short` values, to get the number of bytes in the encoded string. Then the following bytes are read and interpreted UTF-8 encoded bytes; these bytes are converted into characters for the resulting `String`. This method blocks until all of the bytes in the encoded string have been read, the end of the file is encountered, or an exception is thrown. See Appendix B, *The UTF-8 Encoding* for information about the UTF-8 encoding.

seek

public native void seek(long pos) throws IOException

Parameters

`pos`

The new position in the file.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method sets the current file position to the specified position. The position is the offset, in bytes, from the beginning of the file where the next read or write operation occurs.

skipBytes

public int skipBytes(int n) throws IOException

Parameters

`n`

The number of bytes to skip.

Returns

The actual number of skipped bytes.

Throws

`EOFException`

If EOF is encountered.

`IOException`

If any I/O error occurs.

Implements

`DataInput.skipBytes()`

Description

This method skips over `n` bytes.

write

public native void write(int b) throws IOException

Parameters

`b`

The value to write.

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`DataOutput.write(int)`

Description

This method writes the low-order eight bits of `b` to the file as a `byte`.

public void write(byte b[]) throws IOException

Parameters

`b`

An array of bytes to write.

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`DataOutput.write(byte[])`

Description

This method writes the bytes in the given array to the file.

public void write(byte b[], int off, int len) throws IOException

Parameters

`b`

An array of bytes to write.

`off`

An offset into the byte array.

`len`

The number of bytes to write.

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`DataOutput.write(byte[], int, int)`

Description

This method writes `len` bytes from the given array, starting `off` elements from the beginning of the array, to the file.

writeBoolean

public final void writeBoolean(boolean v) throws IOException

Parameters

`v`

The `boolean` value to write.

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`DataOutput.writeBoolean()`

Description

If `v` is `true`, this method writes a byte that contains the value `1` to the file. If `v` is `false`, the method writes a byte that contains the value 0.

writeByte

public final void writeByte(int v) throws IOException

Parameters

`v`

The value to write.

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`DataOutput.writeByte()`

Description

This method writes an 8-bit `byte` to the file, using the low-order eight bits of the given integer `v`.

writeBytes

public final void writeBytes(String s) throws IOException

Parameters

`s`

The `String` to write.

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`DataOutput.writeBytes()`

Description

This method writes the characters in the given `String` to the file as a sequence of 8-bit bytes. The high-order bytes of the characters in the string are ignored.

writeChar

public final void writeChar(int v) throws IOException

Parameters

`v`

The value to write.

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`DataOutput.writeChar()`

Description

This method writes a 16-bit `char` to the file, using the low-order 16 bits of the given integer `v`.

writeChars

public final void writeChars(String s) throws IOException

Parameters

`s`

The `String` to write.

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`DataOutput.writeChars()`

Description

This method writes the characters in the given `String` object to the file as a sequence of 16-bit characters.

writeDouble

public final void writeDouble(double v) throws IOException

Parameters

`v`

The `double` value to write.

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`DataOutput.writeDouble()`

Description

This method writes a 64-bit `double` to the file. The `double` value is converted to a `long` using `doubleToLongBits()` of `Double`; the `long` value is then written to the file as eight bytes with the high-order byte first.

writeFloat

public final void writeFloat(float v) throws IOException

Parameters

`v`

The `float` value to write.

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`DataOutput.writeFloat()`

Description

This method writes a 32-bit `float` to the file. The `float` value is converted to a `int` using `floatToIntBits()` of `Float`; the `int` value is then written to the file as four bytes with the high-order byte first.

writeInt

public final void writeInt(int v) throws IOException

Parameters

`v`

The `int` value to write.

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`DataOutput.writeInt()`

Description

This method writes a 32-bit `int` to the file. The value is written as four bytes with the high-order byte first.

writeLong

public final void writeLong(long v) throws IOException

Parameters

`v`

The `long` value to write.

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`DataOutput.writeLong()`

Description

This method writes a 64-bit `long` to the file. The value is written as eight bytes with the high-order byte first.

writeShort

public final void writeShort(int v) throws IOException

Parameters

`v`

The value to write.

Throws

`IOException`

If any kind of I/O error occurs.

Implements

`DataOutput.writeShort()`

Description

This method writes a 16-bit `short` to the file, using the low-order 16 bits of the given integer `v`.

writeUTF

public final void writeUTF(String str) throws IOException

Parameters

`str`

The `String` to write.

Throws

`IOException`

If any kind of I/O error occurs.

Implements

DataOutput.writeUTF()

Description

This method writes the given String to the file using the UTF-8 encoding. See Appendix B, *The UTF-8 Encoding* for information about the UTF-8 encoding.

Inherited Methods

Method	Inherited From	Method	Inherited From
clone()	Object	equals(Object)	Object
finalize()	Object	getClass()	Object
hashCode()	Object	notify()	Object
notifyAll()	Object	toString()	Object
wait()	Object	wait(long)	Object
wait(long, int)	Object		

See Also

DataInput, DataOutput, File, FileInputStream, FileOutputStream, Double, Float, Integer, IllegalArgumentException, IOException, Long

Reader

Name

Reader

Synopsis

Class Name:

`java.io.Reader`

Superclass:

`java.lang.Object`

Immediate Subclasses:

`java.io.BufferedReader`, `java.io.CharArrayReader`,

`java.io.FilterReader`, `java.io.InputStreamReader`,

`java.io.PipedReader`, `java.io.StringReader`

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The Reader class is an abstract class that is the superclass of all classes that represent input character streams. Reader defines the basic input methods that all character streams provide. A similar hierarchy of classes, based around InputStream, deals with byte streams instead of character streams.

Reader is designed so that read() and read(char[]) both call read(char[], int, int). Thus, a subclass can simply override read(char[], int, int), and all of the read methods will work. Note that this is different from the design of InputStream, where the read() method is the catch-all method. The design of Reader is cleaner and more efficient.

Reader also defines a mechanism for marking a position in the stream and returning to it later, via the mark() and reset() methods. Another method, markSupported(), tells whether or not this mark-and-reset functionality is available in a particular subclass.

Class Summary

```
public abstract class java.io.Reader extends java.lang.Object {
  // Variables
  protected Object lock;
  // Constructors
  protected Reader();
  protected Reader(Object lock);
  // Instance Methods
  public abstract void close();
  public void mark(int readAheadLimit);
  public boolean markSupported();
  public int read();
  public int read(char[] cbuf);
  public abstract int read(char[] cbuf, int off, int len);
  public boolean ready();
  public void reset();
  public long skip(long n) throws IOException;
}
```

Variables

lock

protected Object lock

Description

The object used to synchronize operations on this `Reader` object. For efficiency's sake, a particular implementation of a character stream can choose to synchronize its operations on something other than instances of itself. Thus, any subclass should synchronize on the `lock` object, instead of using a `synchronized` method or the `this` object.

Constructors

Reader

protected Reader()

Description

This constructor creates a `Reader` that synchronizes on the `Reader` itself, or in other words, on the `this` object.

protected Reader(Object lock)

Parameters

`lock`

The object to use for synchronization.

Description

This constructor creates a `Reader` that synchronizes on the given object.

Instance Methods

close

public abstract void close() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method closes the reader and releases any system resources associated with it.

A subclass of `Reader` must implement this method.

mark

public void mark(int readheadLimit) throws IOException

Parameters

`readAheadLimit`

The maximum number of characters that can be read before the saved position becomes invalid.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method tells this `Reader` object to remember its current position, so that the position can be restored by a call to the `reset()` method. The `Reader` can read `readAheadLimit` characters beyond the marked position before the mark becomes invalid.

The implementation of the `mark()` method in `Reader` simply throws an exception to indicate that the mark-and-reset functionality is not implemented. A subclass must override the method to provide the functionality.

markSupported

public boolean markSupported()

Returns

`true` if this reader supports `mark()` and `reset()`; `false` otherwise.

Description

This method returns a `boolean` value that indicates whether or not this object supports mark-and-reset functionality.

The `markSupported()` method in `Reader` always returns `false`. A subclass that implements the mark-and-reset functionality should override the method to return `true`.

read

public int read() throws IOException

Returns

The next character of data or `-1` if the end of the stream is encountered.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method reads the next character of input. The character is returned as an integer in the range `0x0000` to `0xFFFF`. The method blocks until the character is read, the end of stream is encountered, or an exception is thrown.

The implementation of this method in `Reader` reads the character by calling `read(cb, 0, 1)`, where `cb` is a character array, and returning `cb[0]`. Although it is not strictly necessary, a subclass that wants to provide efficient single-character reads should override this method.

public int read(char[] cbuf) throws IOException

Parameters

`cbuf`

An array of characters to be filled from the stream.

Returns

The actual number of characters read or `-1` if the end of the stream is encountered immediately.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method reads characters of input to fill the given array by calling `read(cbuf, 0, cbuf.length)`. The method blocks until some data is available.

A subclass does not usually need to override this method, as it can override `read(char[], int, int)` and have `read(char[])` work automatically.

`public abstract int read(char[] cbuf, int off, int len) throws IOException`

Parameters

`cbuf`

An array of characters to be filled from the stream.

`off`

An offset into the array.

`len`

The number of characters to read.

Returns

The actual number of characters read or `-1` if the end of the stream is encountered immediately.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method reads up to `len` characters of input into the given array starting at index `off`. The method blocks until some data is available.

A subclass of `Reader` must implement this method.

ready

public boolean ready() throws IOException

Returns

> A `boolean` value that indicates whether the reader is ready to be read.

Throws

> `IOException`
>
> > If any kind of I/O error occurs.

Description

> This method returns `true` if the next `read()` is guaranteed to not block.
>
> The implementation of the `ready()` method in `Reader` always returns `false`. A subclass should override this method as appropriate.

reset

public void reset() throws IOException

Throws

> `IOException`
>
> > If there was no previous call to the `mark()` method or the saved position has been invalidated.

Description

> This method restores the position of the stream to the position that was saved by a previous call to `mark()`.
>
> The implementation of the `reset()` method in `Reader` throws an exception to indicate that mark-and-reset functionality is not supported by default. A subclass must override the method to provide the functionality.

skip

public long skip(long n) throws IOException

Parameters

`n`

The number of characters to skip.

Returns

The actual number of characters skipped.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method skips `n` characters of input. In other words, it moves the position of the stream forward by `n` characters.

The implementation of the `skip()` method in `Reader` simply calls `read(cb, 0, n)` where `cb` is a character array that is at least `n` bytes long. A subclass may want to override this method to implement a more efficient skipping algorithm.

Inherited Methods

Method	**Inherited From**	**Method**	**Inherited From**
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`BufferedReader`, `CharArrayReader`, `FilterReader`, `InputStreamReader`, `IOException`, `PipedReader`, `StringReader`

SequenceInputStream

Name

SequenceInputStream

Synopsis

Class Name:

```
java.io.SequenceInputStream
```

Superclass:

```
java.io.InputStream
```

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The SequenceInputStream class allows a series of InputStream objects to be seamlessly concatenated into one stream. In other words, a SequenceInputStream appears and functions as a single InputStream. Internally, however, the SequenceInputStream reads data from each InputStream in the specified order. When the end of a stream is encountered, data is automatically read from the next stream.

Class Summary

```
public class java.io.SequenceInputStream extends java.io.InputStream {
  // Constructors
  public SequenceInputStream(Enumeration e);
  public SequenceInputStream(InputStream s1, InputStream s2);
  // Instance Methods
  public int available();                         // New in 1.1
  public void close();
  public int read();
  public int read(byte[] buf, int pos, int len);
}
```

Constructors

SequenceInputStream

public SequenceInputStream(Enumeration e)

Parameters

e

An Enumeration of input streams.

Description

This constructor creates a SequenceInputStream that reads from each of the InputStream objects in the given Enumeration. Each object in the Enumeration must be an InputStream.

public SequenceInputStream(InputStream s1, InputStream s2)

Parameters

`s1`

An input stream.

`s2`

Another input stream.

Description

This constructor creates a `SequenceInputStream` that reads first from `s1` and then from `s2`.

Instance Methods

available

public int available() throws IOException

Availability

New as of JDK 1.1

Returns

The number of bytes that can be read without blocking, or 0 if the end of the final stream is encountered.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`InputStream.available()`

Description

This method returns the number of bytes that can be read without having to wait for more data to become available. The method returns the result of calling `available()` on the current stream. If the end of the final stream is encountered, the method returns 0.

close

public void close() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`InputStream.close()`

Description

This method closes the stream and releases the system resources that are associated with it. The method closes all the `InputStream` objects attached to this object.

read

public int read() throws IOException

Returns

The next byte of data or `-1` if the end of the final stream is encountered.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`InputStream.read()`

Description

This method reads the next byte of data from the current stream. When the end of the current stream is encountered, that stream is closed, and the first byte of the next `InputStream` is read. If there are no more `InputStream` objects in the `SequenceInputStream`, `-1` is returned to signify the end of the `SequenceInputStream`. The method blocks until the byte is read, the end of the final stream is encountered, or an exception is thrown.

public int read(byte[] buf, int off, int len) throws IOException

Parameters

`buf`

An array of bytes to be filled from the stream.

`off`

An offset into the byte array.

`len`

The number of bytes to read.

Returns

The actual number of bytes read or `-1` if the end of the final stream is encountered immediately.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`InputStream.read(byte[], int, int)`

Description

This method reads up to `len` bytes of input from the current stream into the given array starting at index `off`. When the end of the current stream is encountered, that stream is closed, and bytes are read from the next `InputStream`. If there are no more `InputStream` objects in the `SequenceInputStream`, `-1` is returned to signify the end of the `SequenceInputStream`. The method blocks until there is some data available.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`mark(int)`	`InputStream`
`markSupported()`	`InputStream`	`notify()`	`Object`
`notifyAll()`	`Object`	`reset()`	`InputStream`
`skip(long)`	`InputStream`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`InputStream`, `IOException`

StreamTokenizer

Name

StreamTokenizer

Synopsis

Class Name:

`java.io.StreamTokenizer`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The `StreamTokenizer` class performs a lexical analysis on an `InputStream` object and breaks the stream into tokens. Although `StreamTokenizer` is not a general-purpose parser, it recognizes tokens that are similar to those used in the Java language. A `StreamTokenizer` recognizes identifiers, numbers, quoted strings, and various comment styles.

A `StreamTokenizer` object can be wrapped around an `InputStream`. In this case, when the `StreamTokenizer` reads bytes from the stream, the bytes are converted to Unicode characters by simply zero-extending the byte values to 16 bits. As of Java 1.1, a `StreamTokenizer` can be wrapped around a `Reader` to eliminate this problem.

The `nextToken()` method returns the next token from the stream. The rest of the methods in `StreamTokenizer` control how the object interprets the characters that it reads and tokenizes them.

The parsing functionality of `StreamTokenizer` is controlled by a table and a number of flags. Each character that is read from the `InputStream` is in the range `'\u0000'` to `'\uFFFF'`. The character value looks up attributes of the character in the table. A character can have zero or more of the following attributes: whitespace, alphabetic, numeric, string quote, and comment character.

By default, a `StreamTokenizer` recognizes the following:

- Whitespace characters between `'\u0000'` and `'\u0020'`
- Alphabetic characters from `'a'` through `'z'`, `'A'` through `'Z'`, and `'\u00A0'` and `'\u00FF'`.
- Numeric characters `'1'`, `'2'`, `'3'`, `'4'`, `'5'`, `'6'`, `'7'`, `'8'`, `'9'`, `'0'`, `'.'`, and `'-'`
- String quote characters "`'`" and "`'`"
- Comment character "`/`"

Class Summary

```
public class java.io.StreamTokenizer extends java.lang.Object {
  // Variables
  public double nval;
  public String sval;
  public int ttype;
  public final static int TT_EOF;
  public final static int TT_EOL;
  public final static int TT_NUMBER;
  public final static int TT_WORD;
  // Constructors
  public StreamTokenizer(InputStream in);           // Deprecated in 1.1
  public StreamTokenizer(Reader in);                // New in 1.1
  // Instance Methods
  public void commentChar(int ch);
  public void eolIsSignificant(boolean flag);
  public int lineno();
  public void lowerCaseMode(boolean flag);
  public int nextToken();
  public void ordinaryChar(int ch);
  public void ordinaryChars(int low, int hi);
  public void parseNumbers();
  public void pushBack();
  public void quoteChar(int ch);
  public void resetSyntax();
  public void slashSlashComments(boolean flag);
  public void slashStarComments(boolean flag);
  public String toString();
  public void whitespaceChars(int low, int hi);
  public void wordChars(int low, int hi);
}
```

Variables

nval

public double nval

Description

> This variable contains the value of a `TT_NUMBER` token.

sval

public String sval

Description

> This variable contains the value of a `TT_WORD` token.

ttype

public int ttype

Description

> This variable indicates the token type. The value is either one of the `TT_` constants defined below or the character that has just been parsed from the input stream.

TT_EOF

public final static int TT_EOF = -1

Description

> This token type indicates that the end of the stream has been reached.

TT_EOL

public final static int TT_EOL = '\n'

Description

> This token type indicates that the end of a line has been reached. The value is not returned by `nextToken()` unless `eolIsSignificant(true)` has been called.

TT_NUMBER

public final static int TT_NUMBER = -2

Description

> This token type indicates that a number has been parsed. The number is placed in `nval`.

TT_WORD

public final static int TT_WORD = -3

Description

> This token type indicates that a word has been parsed. The word is placed in `sval`.

Constructors

StreamTokenizer

public StreamTokenizer(InputStream in)

Availability

Deprecated as of JDK 1.1

Parameters

`in`

The input stream to tokenize.

Description

This constructor creates a `StreamTokenizer` that reads from the given `InputStream`. As of JDK 1.1, this method is deprecated and `StreamTokenizer(Reader)` should be used instead.

public StreamTokenizer(Reader in)

Availability

New as of JDK 1.1

Parameters

`in`

The reader to tokenize.

Description

This constructor creates a `StreamTokenizer` that reads from the given `Reader`.

Instance Methods

commentChar

public void commentChar(int ch)

Parameters

`ch`

The character to use to indicate comments.

Description

This method tells this `StreamTokenizer` to treat the given character as the beginning of a comment that ends at the end of the line. The `StreamTokenizer` ignores all of the characters from the comment character to the end of the line. By default, a `StreamTokenizer` treats the `'/'` character as a comment character. This method may be called multiple times if there are multiple characters that begin comment lines.

To specify that a character is not a comment character, use `ordinaryChar()`.

eolIsSignificant

public void eolIsSignificant(boolean flag)

Parameters

`flag`

A `boolean` value that specifies whether or not this `StreamTokenizer` returns `TT_EOL` tokens.

Description

A `StreamTokenizer` recognizes `"\n"`, `"\r"`, and `"\r\n"` as the end of a line. By default, end-of-line characters are treated as whitespace and thus, the `StreamTokenizer` does not return `TT_EOL` tokens from `nextToken()`. Call `eolIsSignificant(true)` to tell the `StreamTokenizer` to return `TT_EOL` tokens.

lineo

public int lineno()

Returns

The current line number.

Description

This method returns the current line number. Line numbers begin at 1.

lowerCaseMode

public void lowerCaseMode(boolean flag)

Parameters

`flag`

A `boolean` value that specifies whether or not this `StreamTokenizer` returns `TT_WORD` tokens in lowercase.

Description

By default, a `StreamTokenizer` does not change the case of the words that it parses. However if you call `lowerCaseMode(true)`, whenever `nextToken()` returns a `TT_WORD` token, the word in `sval` is converted to lowercase.

nextToken

public int nextToken() throws IOException

Returns

One of the token types (`TT_EOF`, `TT_EOL`, `TT_NUMBER`, or `TT_WORD`) or a character code.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method reads the next token from the stream. The value returned is the same as the value of the variable `ttype`. The `nextToken()` method parses the following tokens:

`TT_EOF`

> The end of the input stream has been reached.

`TT_EOL`

> The end of a line has been reached. The `eolIsSignificant()` method controls whether end-of-line characters are treated as whitespace or returned as `TT_EOL` tokens.

`TT_NUMBER`

> A number has been parsed. The value can be found in the variable `nval`. The `parseNumbers()` method tells the `StreamTokenizer` to recognize numbers distinct from words.

`TT_WORD`

> A word has been parsed. The word can be found in the variable `sval`.

Quoted string

> A quoted string has been parsed. The variable `ttype` is set to the quote character, and `sval` contains the string itself. You can tell the `StreamTokenizer` what characters to use as quote characters using `quoteChar()`.

Character

> A single character has been parsed. The variable `ttype` is set to the character value.

ordinaryChar

public void ordinaryChar(int ch)

Parameters

`ch`

> The character to treat normally.

Description

This method causes this `StreamTokenizer` to treat the given character as an *ordinary* character. This means that the character has no special significance as a comment, string quote, alphabetic, numeric, or whitespace character. For example, to tell the `StreamTokenizer` that the slash does not start a single-line comment, use `ordinaryChar('/')`.

ordinaryChars

public void ordinaryChars(int low, int hi)

Parameters

`low`

The beginning of a range of character values.

`hi`

The end of a range of character values.

Description

This method tells this `StreamTokenizer` to treat all of the characters in the given range as ordinary characters. See the description of `ordinaryChar()` above for more information.

parseNumbers

public void parseNumbers()

Description

This method tells this `StreamTokenizer` to recognize numbers. The `StreamTokenizer` constructor calls this method, so the default behavior of a `StreamTokenizer` is to recognize numbers. This method modifies the syntax table of the `StreamTokenizer` so that the following characters have the numeric attribute: `'1'`, `'2'`, `'3'`, `'4'`, `'5'`, `'6'`, `'7'`, `'8'`, `'9'`, `'0'`, `'.'`, and `'-'`

When the parser encounters a token that has the format of a double-precision floating-point number, the token is treated as a number rather than a word. The `ttype` variable is set to `TT_NUMBER`, and `nval` is set to the value of the number.

To use a `StreamTokenizer` that does not parse numbers, make the above characters ordinary using `ordinaryChar()` or `ordinaryChars()`:

pushBack

public void pushBack()

Description

This method has the effect of pushing the current token back onto the stream. In other words, after a call to this method, the next call to the `nextToken()` method returns the same result as the previous call to the `nextToken()`method without reading any input.

quoteChar

public void quoteChar(int ch)

Parameters

`ch`

The character to use as a delimiter for quoted strings.

Description

This method tells this `StreamTokenizer` to treat the given character as the beginning or end of a quoted string. By default, the single-quote character and the double-quote character are string-quote characters. When the parser encounters a string-quote character, the `ttype` variable is set to the quote character, and `sval` is set to the actual string. The string consists of all the characters after (but not including) the string-quote character up to (but not including) the next occurrence of the same string-quote character, a line terminator, or the end of the stream.

To specify that a character is not a string-quote character, use `ordinaryChar()`.

resetSyntax

public void resetSyntax()

Description

This method resets this `StreamTokenizer`, which causes it to treat all characters as ordinary characters. See the description of `ordinaryChar()` above for more information.

slashSlashComments

public void slashSlashComments(boolean flag)

Parameters

`flag`

A `boolean` value that specifies whether or not this `StreamTokenizer` recognizes double-slash comments (`//`).

Description

By default, a `StreamTokenizer` does not recognize double-slash comments. However, if you call `slashSlashComments(true)`, the `nextToken()` method recognizes and ignores double-slash comments.

slashStarComments

public void slashStarComments(boolean flag)

Parameters

`flag`

A `boolean` value that specifies whether or not this `StreamTokenizer` recognizes slash-star (`/* ... */`) comments.

Description

By default, a `StreamTokenizer` does not recognize slash-star comments. However, if you call `slashStarComments(true)`, the `nextToken()` method recognizes and ignores slash-star comments.

toString

public String toString()

Returns

A `String` representation of the current token.

Overrides

`Object.toString()`

Description

This method returns a string representation of the current token recognized by the `nextToken()` method. This string representation consists of the value of `ttype`, the value of `sval` if the token is a word or the value of `nval` if the token is a number, and the current line number.

whitespaceChars

public void whitespaceChars(int low, int hi)

Parameters

`low`

The beginning of a range of character values.

`hi`

The end of a range of character values.

Description

This method causes this `StreamTokenizer` to treat characters in the specified range as whitespace. The only function of whitespace characters is to separate tokens in the stream.

wordChars

public void wordChars(int low, int hi)

Parameters

`low`

The beginning of a range of character values.

`hi`

The end of a range of character values.

Description

This method causes this `StreamTokenizer` to treat characters in the specified range as characters that are part of a word token, or, in other words, consider the characters to be alphabetic. A word token consists of a sequence of characters that begins with an alphabetic character and is followed by zero or more numeric or alphabetic characters.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`InputStream`, `IOException`, `Reader`, `StringTokenizer`

StringBufferInputStream

Name

StringBufferInputStream

Synopsis

Class Name:

```
java.io.StringBufferInputStream
```

Superclass:

```
java.io.InputStream
```

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

Deprecated as of JDK 1.1

Description

The `StringBufferInputStream` class represents a byte stream whose data source is a `String`. This class is similar to the `ByteArrayInputStream` class, which uses a `byte` array as its data source.

`StringBufferInputStream` is deprecated as of JDK 1.1 because it does not correctly convert characters to bytes. The `StringReader` class should now be used to create a character stream from a `String`.

Class Summary

```
public class java.io.StringBufferInputStream extends java.io.InputStream {
  // Variables
  protected String buffer;
  protected int count;
  protected int pos;
  // Constructor
  public StringBufferInputStream(String s);
  // Instance Methods
  public synchronized int available();
  public synchronized int read();
  public synchronized int read(byte[] b, int off, int len);
  public synchronized void reset();
  public synchronized long skip(long n);
}
```

Variables

buffer

protected String buffer

Description

> The buffer that stores the data for the input stream.

count

protected int count

Description

> The size of the buffer, or in other words, the length of the string.

pos

protected int pos

Description

The current stream position.

Constructors

StringBufferInputStream

public StringBufferInputStream(String s)

Parameters

`s`

The `String` to use.

Description

This constructor creates a `StringBufferInputStream` that uses the given `String` as its data source. Note that the data is not copied, so changes made to the `String` affect the data that the `StringBufferInputStream` returns.

Instance Methods

available

public synchronized int available()

Returns

The number of bytes remaining in the string.

Overrides

`InputStream.available()`

Description

This method returns the number of bytes that are left in the string. This is the length of the string, `count`, minus the current stream position, `pos`.

read

public synchronized int read()

Returns

The next byte of data or `-1` if the end of the string is encountered.

Overrides

`InputStream.read()`

Description

This method returns the next byte from the string. The method takes the next character from the string and returns the low eight bits of that character as a byte, which is not the correct way to convert characters into bytes. The method cannot block.

public synchronized int read(byte b[], int off, int len)

Parameters

`b`

An array of bytes to be filled from the stream.

`off`

An offset into the byte array.

`len`

The number of bytes to read.

Returns

The actual number of bytes read or `-1` if the end of the string is encountered immediately.

Overrides

`InputStream.read(byte[], int, int)`

Description

This method copies bytes from the internal buffer into the given array `b`, starting at index `off` and continuing for `len` bytes. The method takes each character from the string and returns the low eight bits of that character as a byte, which is not the correct way to convert characters into bytes.

reset

public synchronized void reset()

Overrides

`InputStream.reset()`

Description

This method sets the position of the `StringBufferInputStream` back to the beginning of the internal buffer.

skip

public synchronized long skip(long n)

Parameters

`n`

The number of bytes to skip.

Returns

The actual number of bytes skipped.

Overrides

`InputStream.skip()`

Description

This method skips `n` bytes of the string. If you try to skip past the end of the string, the stream is positioned at the end of the string.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`close()`	`InputStream`
`equals(Object)`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`hashCode()`	`Object`
`mark(int)`	`InputStream`	`markSupported()`	`InputStream`
`notify()`	`Object`	`notifyAll()`	`Object`
`read(byte[])`	`InputStream`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`ByteArrayInputStream`, `InputStream`, `IOException`, `String`, `StringReader`

StringReader

Name

StringReader

Synopsis

Class Name:

`java.io.StringReader`

Superclass:

`java.io.Reader`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `StringReader` class represents a character stream whose data source is a `String`. This class is similar to the `CharArrayReader` class, which uses a `char` array as its data source.

`StringReader` is meant to replace the `StringBufferInputStream` class as of JDK 1.1. Unlike `StringBufferInputStream`, `StringReader` handles Unicode characters and supports `mark()` and `reset()`.

Class Summary

```
public class java.io.StringReader extends java.io.Reader {
  // Constructors
  public StringReader(String s);
  // Instance Methods
  public void close();
  public void mark(int readAheadLimit);
  public boolean markSupported();
  public int read();
  public int read(char[] cbuf, int off, int len);
  public boolean ready();
  public void reset();
  public long skip(long ns);
}
```

Constructors

StringReader

public StringReader(String s)

Parameters

`s`

The `String` to use.

Description

This constructor creates a `StringReader` that uses the given `String` as its data source. The data is not copied, so changes made to the `String` affect the data that the `StringReader` returns.

Instance Methods

close

public void close()

Overrides

`Reader.close()`

Description

This method closes the reader by removing the link between this `StringReader` and the `String` it was created with.

mark

public void mark(int readAheadLimit) throws IOException

Parameters

`readAheadLimit`

The maximum number of characters that can be read before the saved position becomes invalid.

Throws

`IOException`

If the stream is closed or any other kind of I/O error occurs.

Overrides

`Reader.mark()`

Description

This method causes the `StringReader` to remember its current position. A subsequent call to `reset()` causes the object to return to that saved position, and thus re-read a portion of the string. Because the data for this stream comes from a `String`, there is no limit on reading ahead, so `readAheadLimit` is ignored.

markSupported

public boolean markSupported()

Returns

The `boolean` value `true`.

Overrides

`Reader.markSupported()`

Description

This method returns `true` to indicate that this class supports `mark()` and `reset()`.

read

public int read() throws IOException

Returns

The next character or `-1` if the end of the string is encountered.

Throws

`IOException`

If the stream is closed or any other kind of I/O error occurs.

Overrides

`Reader.read()`

Description

This method returns the next character from the string. The method cannot block.

public int read(char[] cbuf, int off, int len) throws IOException

Parameters

`cbuf`

An array of characters to be filled from the stream.

`off`

An offset into the character array.

`len`

The number of characters to read.

Returns

The actual number of characters read or `-1` if the end of the string is encountered immediately.

Throws

`IOException`

If the stream is closed or any other kind of I/O error occurs.

Overrides

`Reader.read(char[], int, int)`

Description

This method copies up to `len` characters from the internal buffer into the given array `cbuf`, starting at index `off`.

ready

public boolean ready() throws IOException

Returns

A `boolean` value that indicates whether the stream is ready to be read.

Throws

`IOException`

If the stream is closed or any other kind of I/O error occurs.

Overrides

`Reader.ready()`

Description

If there is any data left to be read from the string, this method returns `true`.

reset

public void reset() throws IOException

Throws

`IOException`

If the stream is closed or any other kind of I/O error occurs.

Overrides

`Reader.reset()`

Description

This method resets the position of the `StringReader` to the position that was saved by calling the `mark()` method. If `mark()` has not been called, the `StringReader` is reset to read from the beginning of the string.

skip

public long skip(long ns) throws IOException

Parameters

`ns`

The number of bytes to skip.

Returns

The actual number of bytes skipped.

Throws

`IOException`

If the stream is closed or any other kind of I/O error occurs.

Overrides

`Reader.skip()`

Description

This method skips `ns` characters of input. If you try to skip past the end of the string, the stream is positioned at the end of the string.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals (Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`read(char[])`	`Reader`
`toString()`	`Object`	`wait()`	`Object`

`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`CharArrayReader`, `IOException`, `Reader`, `String`, `StringBufferInputStream`

PREVIOUS HOME NEXT
StringBufferInputStream BOOK INDEX StringWriter

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

StringWriter

Name

StringWriter

Synopsis

Class Name:

`java.io.StringWriter`

Superclass:

`java.io.Writer`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `StringWriter` class represents a stream whose data is written to a string. This class is similar to the `CharArrayWriter` class, which writes its data to a `char` array. The `StringWriter` class uses a `StringBuffer` to store its data; a `String` can be retrieved with the `toString()` method.

Class Summary

```
public class java.io.StringWriter extends java.io.Writer {
  // Constructors
  public StringWriter();
  protected StringWriter(int initialSize);
  // Instance Methods
  public void close();
  public void flush();
  public StringBuffer getBuffer();
  public String toString();
  public void write(int c);
  public void write(char[] cbuf, int off, int len);
  public void write(String str);
  public void write(String str, int off, int len);
}
```

Constructors

StringWriter

public StringWriter()

Description

> This constructor creates a `StringWriter` with an internal buffer that has a default size of 16 characters. The buffer grows automatically as data is written to the stream.

protected StringWriter (int initialSize)

Parameters

> `initialSize`

The initial buffer size.

Description

This constructor creates a `StringWriter` with an internal buffer that has a size of `initialSize` characters. The buffer grows automatically as data is written to the stream.

Instance Methods

close

public void close()

Overrides

`Writer.close()`

Description

This method does nothing. For most subclassesof `Writer`, this method releases any system resources that are associated with the `Writer` object. However, the `StringWriter`'s internal buffer may be needed for subsequent calls to `toString()`. For this reason, `close()` does nothing, and the internal buffer is not released until the `StringWriter` is garbage collected.

flush

public void flush()

Overrides

`Writer.flush()`

Description

This method does nothing. The `StringWriter` writes data directly into its internal buffer; thus it is never necessary to flush the stream.

getBuffer

public StringBuffer getBuffer()

Returns

A reference to the internal data buffer.

Description

This method returns a reference to the `StringBuffer` object that is used in this `StringWriter`.

toString

public String toString()

Returns

A `String` constructed from the internal data buffer.

Overrides

`Object.toString()`

Description

This method returns a reference to a `String` object created from the characters stored in this object's internal buffer.

write

public void write(int c)

Parameters

`c`

The value to write.

Overrides

`Writer.write(int)`

Description

This method writes the given value into the internal buffer. If the buffer is full, it is expanded.

public void write(char[] cbuf, int off, int len)

Parameters

`cbuf`

An array of characters to write to the stream.

`off`

An offset into the character array.

`len`

The number of characters to write.

Overrides

`Writer.write(char[], int, int)`

Description

This method copies `len` characters to this object's internal buffer from `cbuf`, starting `off` elements from the beginning of the array. If the internal buffer is full, it is expanded.

public void write(String str)

Parameters

`str`

A `String` to write to the stream.

Overrides

`Writer.write(String)`

Description

This method copies the characters of `str` into this object's internal buffer. If the internal buffer is full, it is expanded.

public void write(String str, int off, int len)

Parameters

`str`

A `String` to write to the stream.

`off`

An offset into the string.

`len`

The number of characters to write.

Overrides

`Writer.write(String, int, int)`

Description

This method copies `len` characters to this object's internal buffer from `str`, starting `off` characters from the beginning of the given string. If the internal buffer is full, it is expanded.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`

`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`
`write(char[])`	`Writer`		

See Also

`CharArrayWriter`, `String`, `StringBuffer`, `Writer`

PREVIOUS StringReader | HOME BOOK INDEX | NEXT SyncFailedException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Writer

Name

Writer

Synopsis

Class Name:

`java.io.Writer`

Superclass:

`java.lang.Object`

Immediate Subclasses:

`java.io.BufferedWriter`, `java.io.CharArrayWriter`,

`java.io.FilterWriter`, `java.io.OutputStreamWriter`,

`java.io.PipedWriter`, `java.io.PrintWriter`,

`java.io.StringWriter`

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `Writer` class is an `abstract` class that is the superclass of all classes that represent output character streams. `Writer` defines the basic output methods that all character streams provide. A similar hierarchy of classes, based around `OutputStream`, deals with byte streams instead of character streams.

`Writer` is designed so that `write(int b)` and `write(char[])` both call `write(char[], int, int)`. Thus, a subclass can simply override `write(char[], int, int)` and all of the `write` methods will work. Note that this is different from the design of `OutputStream`, where the `write(int b)` method is the catch-all method. The design of `Writer` is cleaner and more efficient.

Some `Writer` subclasses may implement buffering to increase efficiency. `Writer` provides a method--`flush()`--that tells the `Writer` to write any buffered output to the underlying device, which may be a disk drive or a network.

Class Summary

```
public abstract class java.io.Writer extends java.lang.Object {
  // Variables
  protected Object lock;
  // Constructors
  protected Writer();
  protected Writer(Object lock);
  // Instance Methods
  public abstract void close();
  public abstract void flush();
  public void write(int c);
  public void write(char[] cbuf);
  public abstract void write(char[] cbuf, int off, int len);
  public void write(String str);
  public void write(String str, int off, int len);
}
```

Variables

lock

protected Object lock

Description

The object used to synchronize operations on this `Writer` object. For efficiency's sake, a particular implementation of a character stream can choose to synchronize its operations on something other than instances of itself. Thus, any subclass should synchronize on the `lock` object, instead of using a `synchronized` method or the `this` object.

Constructors

Writer

protected Writer()

Description

This constructor creates a `Writer` that synchronizes on the `Writer` itself, or in other words, on the `this` object.

protected Writer(Object lock)

Parameters

`lock`

The object to use for synchronization.

Description

This constructor creates a `Writer` that synchronizes on the given object.

Instance Methods

close

public abstract void close() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method flushes the writer and then closes it, releasing any system resources associated with it.

A subclass of `Writer` must implement this method.

flush

public void flush() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method forces any characters that may be buffered by this `Writer` to be written to the underlying device.

A subclass of `Writer` must implement this method.

write

public void write(int c) throws IOException

Parameters

`c`

The value to write.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method writes a character containing the lowest sixteen bits of the given integer value.

The implementation of this method in `Writer` writes the character by calling `write(cb, 1)` where `cb` is a character array that contains the given value in `cb[0]`. Although it is not strictly necessary, a subclass that wants to provide efficient single-character writes should override this method.

public void write(char[] cbuf) throws IOException

Parameters

`cbuf`

An array of characters to write to the stream.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method writes the given array of characters to the stream by calling `write(cbuf, 0, cbuf.length)`.

A subclass does not usually need to override this method, as it can override `write(char[], int, int)` and have `write(char[])` work automatically.

`public abstract void write(char[] cbuf, int off, int len) throws IOException`

Parameters

`cbuf`

An array of characters to write to the stream.

`off`

An offset into the array.

`len`

The number of characters to write.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method writes `len` characters contained in the given array starting at index `off`.

A subclass of `Writer` must implement this method.

public void write(String str) throws IOException

Parameters

`str`

A string to write to the stream.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method writes the given string to the stream by calling `write(str,str.length)`.

A subclass does not usually need to override this method, as it can override `write(char[], int, int)` and have it work automatically.

public void write(String str, int off, int len) throws IOException

Parameters

`str`

A string to write to the stream.

`off`

An offset into the string.

`len`

The number of characters to write.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method writes `len` characters contained in the given string starting at index `off`. The method does this by creating an array of characters for the specified portion of the string and then calling `write(cb, 0, cb.length)` on the character array `cb`.

A subclass does not usually need to override this method, as it can override `write(char[], int, int)` and have it work automatically.

Inherited Methods

Method	**Inherited From**	**Method**	**Inherited From**
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`BufferedWriter`, `CharArrayWriter`, `FilterWriter`, `IOException`, `OutputStreamWriter`, `PipedWriter`, `PrintWriter`, `StringWriter`

PREVIOUS WriteAbortedException | HOME BOOK INDEX | NEXT AbstractMethodError

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

ArithmeticException

Name

ArithmeticException

Synopsis

Class Name:

`java.lang.ArithmeticException`

Superclass:

`java.lang.RuntimeException`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

An `ArithmeticException` is thrown to indicate an exceptional arithmetic condition, such as integer division by zero.

Class Summary

```
public class java.lang.ArithmeticException
           extends java.lang.RuntimeException {
  // Constructors
  public ArithmeticException();
  public ArithmeticException(String s);
}
```

Constructors

ArithmeticException

public ArithmeticException()

Description

This constructor creates an `ArithmeticException` with no associated detail message.

public ArithmeticException(String s)

Parameters

`s`

The detail message.

Description

This constructor creates `ArithmeticException` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`

`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Exception`, `RuntimeException`, `Throwable`

PREVIOUS | HOME | NEXT
AbstractMethodError | BOOK INDEX | ArrayIndexOutOfBoundsException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

ArrayIndexOutOfBoundsException

Name

ArrayIndexOutOfBoundsException

Synopsis

Class Name:

`java.lang.ArrayIndexOutOfBoundsException`

Superclass:

`java.lang.IndexOutOfBoundsException`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

An `ArrayIndexOutOfBoundsException` is thrown when an out-of-range index is detected by an array object. An out-of-range index occurs when the index is less than zero or greater than or equal to the size of the array.

Class Summary

```
public class java.lang.ArrayIndexOutOfBoundsException
             extends java.lang.IndexOutOfBoundsException {
  // Constructors
  public ArrayIndexOutOfBoundsException();
  public ArrayIndexOutOfBoundsException(int index);
  public ArrayIndexOutOfBoundsException(String s);
}
```

Constructors

ArrayIndexOutOfBoundsException

public ArrayIndexOutOfBoundsException()

Description

> This constructor creates an `ArrayIndexOutOfBoundsException` with no associated detail message.

public ArrayIndexOutOfBoundsException(int index)

Parameters

> `index`
>
> > The index value that was out-of-bounds

Description

> This constructor creates an `ArrayIndexOutOfBoundsException` with an associated message that mentions the specified index.

public ArrayIndexOutOfBoundsException(String s)

Parameters

> `s`

The detail message.

Description

This constructor creates an `ArrayIndexOutOfBoundsException` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Exception`, `IndexOutOfBoundsException`, `RuntimeException`, `Throwable`

PREVIOUS ArithmeticException | HOME | BOOK INDEX | NEXT ArrayStoreException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

ArrayStoreException

Name

ArrayStoreException

Synopsis

Class Name:

`java.lang.ArrayStoreException`

Superclass:

`java.lang.RuntimeException`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

An `ArrayStoreException` is thrown when there is an attempt to store a value in an array element that is incompatible with the type of the array.

Class Summary

```
public class java.lang.ArrayStoreException
             extends java.lang.RuntimeException {
  // Constructors
  public ArrayStoreException();
  public ArrayStoreException(String s);
}
```

Constructors

ArrayStoreException

public ArrayStoreException()

Description

This constructor creates an `ArrayStoreException` with no associated detail message.

public ArrayStoreException(String s)

Parameters

`s`

The detail message.

Description

This constructor creates an `ArrayStoreException` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`

`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Exception`, `RuntimeException`, `Throwable`

PREVIOUS
ArrayIndexOutOfBoundsException

HOME
BOOK INDEX

NEXT
Boolean

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Boolean

Name

Boolean

Synopsis

Class Name:

```
java.lang.Boolean
```

Superclass:

```
java.lang.Object
```

Immediate Subclasses:

None

Interfaces Implemented:

```
java.io.Serializable
```

Availability:

JDK 1.0 or later

Description

The `Boolean` class provides an object wrapper for a `boolean` value. This is useful when you need to treat a `boolean` value as an object. For example, there are a number of utility methods that take a reference to an `Object` as one of their arguments. You cannot specify a `boolean` value for one of these arguments, but you can provide a reference to a `Boolean` object that encapsulates the `boolean` value. Furthermore, as of JDK 1.1, the `Boolean` class is necessary to support the Reflection API and class literals.

Class Summary

```
public final class java.lang.Boolean {
    // Constants
    public final static Boolean FALSE;
    public final static Boolean TRUE;
    public final static Class TYPE;                    // New in 1.1
    // Constructors
    public Boolean(boolean value);
    public Boolean(String s);
    // Class Methods
    public static boolean getBoolean(String name);
    public static Boolean valueOf(String s);
    // Instance Methods
    public boolean booleanValue();
    public boolean equals(Object obj);
    public int hashCode();
    public String toString();
}
```

Constants

TRUE

public static final Boolean TRUE

Description

> A constant `Boolean` object that has the value `true`.

FALSE

public static final Boolean FALSE

Description

A constant `Boolean` object that has the value `false`.

TYPE

`public static final Class TYPE`

Availability

New as of JDK 1.1

Description

The `Class` object that represents the type `boolean`. It is always true that `Boolean.TYPE == boolean.class`.

Constructors

Boolean

`public Boolean(boolean value)`

Parameters

`value`

The `boolean` value to be made into a `Boolean` object.

Description

Constructs a `Boolean` object with the given value.

`public Boolean(String s)`

Parameters

`s`

The string to be made into a `Boolean` object.

Description

Constructs a `Boolean` object with the value specified by the given string. If the string equals `'true'` (ignoring case), the value of the object is `true`; otherwise it is `false`.

Class Methods

getBoolean

`public static boolean getBoolean(String name)`

Parameters

`name`

The name of a system property.

Returns

The `boolean` value of the system property.

Description

This methods retrieves the `boolean` value of a named system property.

valueOf

`public static Boolean valueOf(String s)`

Parameters

`s`

The string to be made into a `Boolean` object.

Returns

A `Boolean` object with the value specified by the given string.

Description

This method returns a `Boolean` object with the value `true` if the string equals `"true"` (ignoring case); otherwise the value of the object is `false`.

Instance Methods

booleanValue

```
public boolean booleanValue()
```

Returns

The `boolean` value contained by the object.

equals

```
public boolean equals(Object obj)
```

Parameters

`obj`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `obj` is an instance of `Boolean`, and it contains the same value as the object this method is associated with.

hashCode

`public int hashCode()`

Returns

A hashcode based on the `boolean` value of the object.

Overrides

`Object.hashCode()`

toString

`public String toString()`

Returns

`"true"` if the value of the object is `true`; `"false"` otherwise.

Overrides

`Object.toString()`

Description

This method returns a string representation of the `Boolean` object.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`notify()`	`Object`

`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`Class`, `Object`, `Serializable`, `System`

PREVIOUS | HOME | NEXT
ArrayStoreException | BOOK INDEX | Byte

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Byte

Name

Byte

Synopsis

Class Name:

`java.lang.Byte`

Superclass:

`java.lang.Number`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `Byte` class provides an object wrapper for a `byte` value. This is useful when you need to treat a `byte` value as an object. For example, there are a number of utility methods that take a reference to an `Object` as one of their arguments. You cannot specify a `byte` value for one of these arguments, but you can provide a reference to a `Byte` object that encapsulates the `byte` value. Furthermore, the `Byte` class is necessary as of JDK 1.1 to support the Reflection API and class literals.

The `Byte` class also provides a number of utility methods for converting `byte` values to other primitive types and for converting `byte` values to strings and vice versa.

Class Summary

```
public final class java.lang.Byte extends java.lang.Number {
  // Constants
  public static final byte MAX_VALUE;
  public static final byte MIN_VALUE;
  public static final Class TYPE;
  // Constructors
  public Byte(byte value);
  public Byte(String s);
  // Class Methods
  public static Byte decode(String nm);
  public static byte parseByte(String s);
  public static byte parseByte(String s, int radix);
  public static String toString(byte b);
  public static Byte valueOf(String s, int radix);
  public static Byte valueOf(String s);
  // Instance Methods
  public byte byteValue();
  public double doubleValue;
  public boolean equals(Object obj);
  public float floatValue
  public int hashCode();
  public int intValue();
  public long longValue();
  public short shortValue();
  public String toString();
}
```

Constants

MAX_VALUE

public static final byte MAX_VALUE= 127

The largest value that can be represented by a `byte`.

MIN_VALUE

public static final byte MIN_VALUE= -128

The smallest value that can be represented by a `byte`.

TYPE

public static final Class TYPE

The `Class` object that represents the primitive type `byte`. It is always true that `Byte.TYPE` == `byte.class`.

Constructors

Byte

public Byte(byte value)

Parameters

> `value`
>
> > The `byte` value to be encapsulated by this object.

Description

> Creates a `Byte` object with the specified byte value.

public Byte(String s) throws NumberFormatException

Parameters

> `s`
>
> > The string to be made into a `Byte` object.

Throws

> `NumberFormatException`
>
> > If the sequence of characters in the given `String` does not form a valid `byte` literal.

Description

> Constructs a `Byte` object with the value specified by the given string. The string should consist of one or more digit characters. The digit characters can be preceded by a single `-' character. If the string contains any other characters, the constructor throws a `NumberFormatException`.

Class Methods

decode

public static Byte decode(String nm) throws NumberFormatException

Parameters

`nm`

A `String` representation of the value to be encapsulated by a `Byte` object. If the string begins with # or `0x`, it is a radix 16 representation of the value. If the string begins with `0`, it is a radix 8 representation of the value. Otherwise, it is assumed to be a radix 10 representation of the value.

Returns

A `Byte` object that encapsulates the given value.

Throws

`NumberFormatException`

If the `String` contains any non-digit characters other than a leading minus sign or the value represented by the `String` is less than `Byte.MIN_VALUE` or greater than `Byte.MAX_VALUE`.

Description

This method returns a `Byte` object that encapsulates the given value.

parseByte

public static byte parseByte(String s) throws NumberFormatException

Parameters

`s`

The `String` to be converted to a `byte` value.

Returns

The numeric value of the byte represented by the `String` object.

Throws

`NumberFormatException`

If the `String` does not contain a valid representation of a `byte` or the value represented by the `String` is less than `Byte.MIN_VALUE` or greater than `Byte.MAX_VALUE`.

Description

This method returns the numeric value of the `byte` represented by the contents of the given `String` object. The `String` must contain only decimal digits, except that the first character may be a minus sign.

`public static byte parseByte(String s, int radix) throws NumberFormatException`

Parameters

`s`

The `String` to be converted to a `byte` value.

`radix`

The radix used in interpreting the characters in the `String` as digits. This value must be in the range `Character.MIN_RADIX` to `Character.MAX_RADIX`. If `radix` is in the range 2 through 10, only characters for which the `Character.isDigit()` method returns `true` are considered to be valid digits. If `radix` is in the range 11 through 36, characters in the ranges `A' through `Z' and `a' through `z' may be considered valid digits.

Returns

The numeric value of the byte represented by the `String` object in the specified radix.

Throws

`NumberFormatException`

If the `String` does not contain a valid representation of a `byte`, `radix` is not in the appropriate range, or the value represented by the `String` is less than `Byte.MIN_VALUE` or greater than `Byte.MAX_VALUE`.

Description

This method returns the numeric value of the `byte` represented by the contents of the given `String` object in the specified radix. The `String` must contain only valid digits of the specified radix, except that the first character may be a minus sign. The digits are parsed in the specified radix to produce the numeric value.

toString

public String toString(byte b)

Parameters

b

The byte value to be converted to a string.

Returns

The string representation of the given value.

Description

This method returns a String object that contains the decimal representation of the given value.

This method returns a string that begins with `-' if the given value is negative. The rest of the string is a sequence of one or more of the characters `0', `1', `2', `3', `4', `5', `6', `7', `8', and `9'. This method returns "0" if its argument is 0. Otherwise, the string returned by this method does not begin with "0" or "-0".

valueOf

public static Byte valueOf(String s) throws NumberFormatException

Parameters

s

The string to be made into a Byte object.

Returns

The Byte object constructed from the string.

Throws

NumberFormatException

If the String does not contain a valid representation of a byte or the value represented by the String is less than Byte.MIN_VALUE or greater than Byte.MAX_VALUE.

Description

Constructs a `Byte` object with the value specified by the given string. The string should consist of one or more digit characters. The digit characters can be preceded by a single `-' character. If the string contains any other characters, the method throws a `NumberFormatException`.

`public static Byte valueOf(String s, int radix) throws NumberFormatException`

Parameters

`s`

The string to be made into a `Byte` object.

`radix`

The radix used in converting the string to a value. This value must be in the range `Character.MIN_RADIX` to `Character.MAX_RADIX`.

Returns

The `Byte` object constructed from the string.

Throws

`NumberFormatException`

If the `String` does not contain a valid representation of a `byte`, radix is not in the appropriate range, or the value represented by the `String` is less than `Byte.MIN_VALUE` or greater than `Byte.MAX_VALUE`.

Description

Constructs a `Byte` object with the value specified by the given string in the specified radix. The string should consist of one or more digit characters or characters in the range `A' to `Z' or `a' to `z' that are considered digits in the given radix. The digit characters can be preceded by a single `-' character. If the string contains any other characters, the method throws a `NumberFormatException`.

Instance Methods

byteValue

public byte byteValue()

Returns

The value of this object as a `byte`.

Overrides

`Number.byteValue()`

Description

This method returns the value of this object as a `byte`.

doubleValue

public double doubleValue()

Returns

The value of this object as a `double`.

Overrides

`Number.doubleValue()`

Description

This method returns the value of this object as a `double`.

equals

public boolean equals(Object obj)

Parameters

`obj`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `obj` is an instance of `Byte` and it contains the same value as the object this method is associated with.

floatValue

public float floatValue()

Returns

The value of this object as a `float`.

Overrides

`Number.floatValue()`

Description

This method returns the value of this object as a `float`.

hashCode

public int hashCode()

Returns

A hashcode based on the `byte` value of the object.

Overrides

`Object.hashCode()`

Description

This method returns a hashcode computed from the value of this object.

intValue

public int intValue()

Returns

The value of this object as an `int`.

Overrides

`Number.intValue()`

Description

This method returns the value of this object as an `int`.

longValue

public long longValue()

Returns

The value of this object as a `long`.

Overrides

`Number.longValue()`

Description

This method returns the value of this object as a `long`.

shortValue

public short shortValue()

Returns

The value of this object as a `short`.

Overrides

`Number.shortValue()`

Description

This method returns the value of this object as a `short`.

toString

public String toString()

Returns

The string representation of the value of this object.

Overrides

`Object.toString()`

Description

This method returns a `String` object that contains the decimal representation of the value of this object.

This method returns a string that begins with `-` if the given value is negative. The rest of the string is a sequence of one or more of the characters `0', `1', `2', `3', `4', `5', `6', `7', `8', and `9'. This method returns "0" if its argument is `0`. Otherwise, the string returned by this method does not begin with "0" or "-0".

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`Character`, `Class`, `Double`, `Float`, `Integer`, `Long`, `Number`, `Short`, `String`

Character

Name

Character

Synopsis

Class Name:

```
java.lang.Character
```

Superclass:

```
java.lang.Object
```

Immediate Subclasses:

None

Interfaces Implemented:

```
java.io.Serializable
```

Availability:

JDK 1.0 or later

Description

The `Character` class provides an object wrapper for a `char` value. This is useful when you need to treat a char value as an object. For example, there are a number of utility methods that take a reference to an Object as one of their arguments. You cannot specify a char value for one of these arguments, but you can provide a reference to a Character object that encapsulates the char value. Furthermore, as of JDK 1.1, the Character class is necessary to support the Reflection API and class literals.

In Java, Character objects represent values defined by the Unicode standard. Unicode is defined by an organization called the Unicode Consortium. The defining document for Unicode is *The Unicode Standard, Version 2.0* (ISBN 0-201-48345-9). More recent information about Unicode is available at *http://unicode.org*. Appendix A, *The Unicode 2.0 Character Set*, contains a table that lists the characters defined by the Unicode 2.0 standard.

The Character class provides some utility methods, such as methods for determining the type (e.g., uppercase or lowercase, digit or letter) of a character and for converting from uppercase to lowercase. The logic for these utility methods is based on a Unicode attribute table that is part of the Unicode standard. That table is available at *ftp://unicode.org/pub/2.0-Update/UnicodeData-2.0.14.txt*.

Some of the methods in the Character class are concerned with characters that are digits; these characters are used by a number of other classes to convert strings that contain numbers into actual numeric values. The digit-related methods all use a radix value to interpret characters. The *radix* is the numeric base used to represent numbers as characters or strings. Octal is a radix 8 representation, while hexadecimal is a radix 16 representation. The methods that require a radix parameter use it to determine which characters should be treated as valid digits. In radix 2, only the characters `0' and `1' are valid digits. In radix 16, the characters `0' through `9', `a' through `z', and `A' through `Z' are considerd valid digits.

Class Summary

```
public final class java.lang.Character extends java.lang.Object
                                       implements java.io.Serializable {
    // Constants
    public final static byte COMBINING_SPACING_MARK;        // New in 1.1
    public final static byte CONNECTOR_PUNCTUATION;         // New in 1.1
    public final static byte CONTROL;                       // New in 1.1
    public final static byte CURRENCY_SYMBOL;               // New in 1.1
    public final static byte DASH_PUNCTUATION;              // New in 1.1
    public final static byte DECIMAL_DIGIT_NUMBER;          // New in 1.1
    public final static byte ENCLOSING_MARK;                // New in 1.1
    public final static byte END_PUNCTUATION;               // New in 1.1
    public final static byte FORMAT;                        // New in 1.1
    public final static byte LETTER_NUMBER;                 // New in 1.1
    public final static byte LINE_SEPARATOR;                // New in 1.1
    public final static byte LOWERCASE_LETTER;              // New in 1.1
    public final static byte MATH_SYMBOL;                   // New in 1.1
    public final static int MAX_RADIX;
    public final static char MAX_VALUE;
    public final static int MIN_RADIX;
    public final static char MIN_VALUE;
    public final static byte MODIFIER_LETTER;               // New in 1.1
    public final static byte MODIFIER_SYMBOL;               // New in 1.1
    public final static byte NON_SPACING_MARK;              // New in 1.1
    public final static byte OTHER_LETTER;                  // New in 1.1
    public final static byte OTHER_NUMBER;                  // New in 1.1
    public final static byte OTHER_PUNCTUATION;             // New in 1.1
    public final static byte OTHER_SYMBOL;                  // New in 1.1
    public final static byte PARAGRAPH_SEPARATOR;           // New in 1.1
    public final static byte PRIVATE_USE;                   // New in 1.1
    public final static byte SPACE_SEPARATOR;               // New in 1.1
    public final static byte START_PUNCTUATION;             // New in 1.1
    public final static byte SURROGATE;                     // New in 1.1
    public final static byte TITLECASE_LETTER;              // New in 1.1
    public final static Class TYPE;                         // New in 1.1
    public final static byte UNASSIGNED;                    // New in 1.1
    public final static byte UPPERCASE_LETTER;              // New in 1.1
    // Constructors
    public Character(char value);
    // Class Methods
    public static int digit(char ch, int radix);
    public static char forDigit(int digit, int radix);
    public static int getNumericValue(char ch);             // New in 1.1
    public static int getType(char ch);                     // New in 1.1
    public static boolean isDefined(char ch);
    public static boolean isDigit(char ch);
    public static boolean isIdentifierIgnorable(char ch);   // New in 1.1
    public static boolean isISOControl(char ch);            // New in 1.1
    public static boolean isJavaIdentifierPart(char ch);    // New in 1.1
    public static boolean isJavaIdentifierStart(char ch);   // New in 1.1
    public static boolean isJavaLetter(char ch);        // Deprecated in 1.1
    public static boolean isJavaLetterOrDigit(char ch); // Deprecated in 1.1
    public static boolean isLetter(char ch);
    public static boolean isLetterOrDigit(char ch);
    public static boolean isLowerCase(char ch);
    public static boolean isSpace(char ch);             // Deprecated in 1.1
    public static boolean isSpaceChar(char ch);             // New in 1.1
    public static boolean isTitleCase(char ch);
    public static boolean isUnicodeIdentifierPart(char ch); // New in 1.1
    public static boolean isUnicodeIdentifierStart(char ch);// New in 1.1
    public static boolean isUpperCase(char ch);
    public static boolean isWhitespace(char ch);            // New in 1.1
    public static char toLowerCase(char ch);
    public static char toTitleCase(char ch);
    public static char toUpperCase(char ch);
    // Instance Methods
    public char charValue();
```

```
    public boolean equals(Object obj);
    public int hashCode();
    public String toString();
}
```

Constants

COMBINING_SPACING_MARK

`public final static byte COMBINING_SPACING_MARK`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

CONNECTOR_PUNCTUATION

`public final static byte CONNECTOR_PUNCTUATION`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

CONTROL

`public final static byte CONTROL`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

CURRENCY_SYMBOL

`public final static byte CURRENCY_SYMBOL`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

DASH_PUNCTUATION

`public final static byte DASH_PUNCTUATION`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

DECIMAL_DIGIT_NUMBER

`public final static byte DECIMAL_DIGIT_NUMBER`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

ENCLOSING_MARK

`public final static byte ENCLOSING_MARK`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

END_PUNCTUATION

`public final static byte END_PUNCTUATION`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

FORMAT

`public final static byte FORMAT`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

LETTER_NUMBER

`public final static byte LETTER_NUMBER`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

LINE_SEPARATOR

`public final static byte LINE_SEPARATOR`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

LOWERCASE_LETTER

`public final static byte LOWERCASE_LETTER`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

MATH_SYMBOL

`public final static byte MATH_SYMBOL`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

MAX_RADIX

`public static final int MAX_RADIX = 36`

Description

The maximum value that can be specified for a radix.

MAX_VALUE

`public final static char MAX_VALUE = '\ufff'f`

Description

The largest value that can be represented by a `char`.

MIN_RADIX

`public static final int MIN_RADIX = 2`

Description

The minimum value that can be specified for a radix.

MIN_VALUE

`public final static char MIN_VALUE = '\u0000'`

Description

The smallest value that can be represented by a `char`.

MODIFIER_LETTER

`public final static byte MODIFIER_LETTER`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

MODIFIER_SYMBOL

`public final static byte MODIFIER_SYMBOL`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

NON_SPACING_MARK

`public final static byte NON_SPACING_MARK`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

OTHER_LETTER

`public final static byte OTHER_LETTER`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

OTHER_NUMBER

`public final static byte OTHER_NUMBER`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

OTHER_PUNCTUATION

`public final static byte OTHER_PUNCTUATION`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

OTHER_SYMBOL

`public final static byte OTHER_SYMBOL`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

PARAGRAPH_SEPARATOR

`public final static byte PARAGRAPH_SEPARATOR`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

PRIVATE_USE

`public final static byte PRIVATE_USE`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

SPACE_SEPARATOR

`public final static byte SPACE_SEPARATOR`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

START_PUNCTUATION

`public final static byte START_PUNCTUATION`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

SURROGATE

`public final static byte SURROGATE`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

TITLECASE_LETTER

`public final static byte TITLECASE_LETTER`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

TYPE

`public static final Class TYPE`

Availability

New as of JDK 1.1

Description

The `Class` object that represents the type `char`. It is always true that `Character.TYPE == char.class`.

UNASSIGNED

`public final static byte UNASSIGNED`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

UPPERCASE_LETTER

`public final static byte UPPERCASE_LETTER`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

Constructors

Character

`public Character(char value)`

Parameters

`value`

The `char` value to be encapsulated by this object.

Description

Creates a `Character` object with the given `char` value.

Class Methods

digit

`public static int digit(char ch, int radix)`

Parameters

`ch`

A `char` value that is a legal digit in the given radix.

`radix`

The radix used in interpreting the specified character as a digit. If `radix` is in the range 2 through 10, only characters for which the `isDigit()` method returns `true` are considered to be valid digits. If `radix` is in the range 11 through 36, characters in the ranges `A' through `Z' and `a' through `z' may be considered valid digits.

Returns

The numeric value of the digit. This method returns `-1` if the value of `ch` is not considered a valid digit, if `radix` is less than `MIN_RADIX`, or if `radix` is greater than `MAX_RADIX`.

Description

Returns the numeric value represented by a digit character. For example, `digit('7',10)` returns 7. If the value of `ch` is not a valid digit, the method returns `-1`. For example, `digit('7',2)` returns `-1` because `'7'` is not a valid digit in radix 2. A number of methods in other classes use this method to convert strings that contain numbers to actual numeric values. The `forDigit()` method is an approximate inverse of this method.

If `radix` is greater than 10, characters in the range `A' to `A'+`radix`-11 are treated as valid digits. Such a character has the numeric value `ch`-`A'+10. By the same token, if `radix` is greater than 10, characters in the range `a' to `a'+`radix`-11 are treated as valid digits. Such a character has the numeric value `ch`-`a'+10.

forDigit

`public static char forDigit(int digit, int radix)`

Parameters

`digit`

The numeric value represented as a digit character.

`radix`

The radix used to represent the specified value.

Returns

The character that represents the digit corresponding to the specified numeric value. The method returns `\ 0' if `digit` is less than 0, if `digit` is equal to or greater than `radix`, if `radix` is less than `MIN_RADIX`, or if `radix` is greater than `MAX_RADIX`.

Description

This method returns the character that represents the digit corresponding to the specified numeric value. If `digit` is in the range 0 through 9, the method returns `0'+`digit`. If `digit` is in the range 10 through `MAX_RADIX`-1, the method returns `a'+`digit`-10. The method returns `\ 0' if `digit` is less than 0, if `digit` is equal to or greater than `radix`, if `radix` is less than `MIN_RADIX`, or if `radix` is greater than `MAX_RADIX`.

getNumericValue

`public static int getNumericValue(char ch)`

Availability

New as of JDK 1.1

Parameters

`ch`

A `char` value.

Returns

The Unicode numeric value of the character as a nonnegative integer. This method returns `-1` if the character has no numeric value; it returns `-2` if the character has a numeric value that is not a nonnegative integer, such as `1/2`.

Description

This method returns the Unicode numeric value of the specified character as a nonnegative integer.

getType

`public static int getType(char ch)`

Availability

New as of JDK 1.1

Parameters

`ch`

A `char` value.

Returns

An `int` value that represents the Unicode general category type of the character.

Description

This method returns the Unicode general category type of the specified character. The value corresponds to one of the general category constants defined by `Character`.

isDefined

`public static boolean isDefined(char ch)`

Parameters

`ch`

A `char` value to be tested.

Returns

`true` if the specified character has an assigned meaning in the Unicode character set; otherwise `false`.

Description

This method returns `true` if the specified character value has an assigned meaning in the Unicode character set.

isDigit

`public static boolean isDigit(char ch)`

Parameters

`ch`

A `char` value to be tested.

Returns

`true` if the specified character is defined as a digit in the Unicode character set; otherwise `false`.

Description

This method determines whether or not the specified character is a digit, based on the definition of the character in Unicode.

isIdentifierIgnorable

`public static boolean isIdentifierIgnorable(char ch)`

Availability

New as of JDK 1.1

Parameters

`ch`

A `char` value to be tested.

Returns

`true` if the specified character is ignorable in a Java or Unicode identifier; otherwise `false`.

Description

This method determines whether or not the specified character is ignorable in a Java or Unicode identifier. The following characters are ignorable in a Java or Unicode identifier:

`\u0000 - \u0008 \u000E - \u001B \u007F - \u009F`	ISO control characters that aren't whitespace
`\u200C - \u200F`	Join controls
`\u200A - \u200E`	Bidirectional controls
`\u206A - \u206F`	Format controls
`\uFEFF`	Zero-width no-break space

isISOControl

`public static boolean isISOControl(char ch)`

Availability

New as of JDK 1.1

Parameters

`ch`

A `char` value to be tested.

Returns

`true` if the specified character is an ISO control character; otherwise `false`.

Description

This method determines whether or not the specified character is an ISO control character. A character is an ISO control character if it falls in the range `\u0000` through `\u001F` or `\u007F` through `\u009F`.

isJavaIdentifierPart

`public static boolean isJavaIdentifierPart(char ch)`

Availability

New as of JDK 1.1

Parameters

`ch`

A `char` value to be tested.

Returns

`true` if the specified character can appear after the first character in a Java identifier; otherwise `false`.

Description

This method returns `true` if the specified character can appear in a Java identifier after the first character. A character is considered part of a Java identifier if and only if it is a letter, a digit, a currency symbol (e.g., $), a connecting punctuation character (e.g., _), a numeric letter (e.g., a Roman numeral), a combining mark, a nonspacing mark, or an ignorable control character.

isJavaIdentifierStart

`public static boolean isJavaIdentifierStart(char ch)`

Availability

New as of JDK 1.1

Parameters

`ch`

A `char` value to be tested.

Returns

`true` if the specified character can appear as the first character in a Java identifier; otherwise `false`.

Description

This method returns `true` if the specified character can appear in a Java identifier as the first character. A character is considered a start of a Java identifier if and only if it is a letter, a currency symbol (e.g., $), or a connecting punctuation character (e.g., _).

isJavaLetter

`public static boolean isJavaLetter(char ch)`

Availability

Deprecated as of JDK 1.1

Parameters

`ch`

A `char` value to be tested.

Returns

`true` if the specified character can appear as the first character in a Java identifier; otherwise `false`.

Description

This method returns `true` if the specified character can appear as the first character in a Java identifier. A character is considered a Java letter if and only if it is a letter, the character $, or the character _ . This method returns `false` for digits because digits are not allowed as the first character of an identifier.

This method is deprecated as of JDK 1.1. You should use `isJavaIdentifierStart()` instead.

isJavaLetterOrDigit

`public static boolean isJavaLetterOrDigit(char ch)`

Availability

Deprecated as of JDK 1.1

Parameters

ch

A char value to be tested.

Returns

true if the specified character can appear after the first character in a Java identifier; otherwise false.

Description

This method returns true if the specified character can appear in a Java identifier after the first character. A character is considered a Java letter or digit if and only if it is a letter, a digit, the character $, or the character _.

This method is deprecated as of JDK 1.1. You should use isJavaIdentifierPart() instead.

isLetter

public static boolean isLetter(char ch)

Parameters

ch

A char value to be tested.

Returns

true if the specified character is defined as a letter in the Unicode character set; otherwise false.

Description

This method determines whether or not the specified character is a letter, based on the definition of the character in Unicode. This method does not consider character values in ranges that have not been assigned meanings by Unicode to be letters.

isLetterOrDigit

public static boolean isLetterOrDigit(char ch)

Parameters

ch

A `char` value to be tested.

Returns

`true` if the specified character is defined as a letter in the Unicode character set; otherwise `false`.

Description

This method determines whether or not the specified character is a letter or a digit, based on the definition of the character in Unicode. There are some ranges that have not been assigned meanings by Unicode. If a character value is in one of these ranges, this method does not consider the character to be a letter.

isLowerCase

`public static boolean isLowerCase (char ch)`

Parameters

`ch`

A `char` value to be tested.

Returns

`true` if the specified character is defined as lowercase in the Unicode character set; otherwise `false`.

Description

This method determines whether or not the specified character is lowercase. Unicode defines a number of characters that do not have case mappings; if the specified character is one of these characters, the method returns `false`.

isSpace

`public static boolean isSpace(char ch)`

Availability

Deprecated as of JDK 1.1

Parameters

`ch`

A `char` value to be tested.

Returns

`true` if the specified character is defined as whitespace in the ISO-Latin-1 character set; otherwise `false`.

Description

This method determines whether or not the specified character is whitespace. This method recognizes the whitespace characters shown in the following table.

`\u0009`	Horizontal tab
`\u000A`	Newline
`\u000C`	Formfeed
`\u000D`	Carriage return
`\u0020 ` ``	Space

This method is deprecated as of JDK 1.1. You should use `isWhitespace()` instead.

isSpaceChar

`public static boolean isSpaceChar(char ch)`

Availability

New as of JDK 1.1

Parameters

`ch`

A `char` value to be tested.

Returns

`true` if the specified character is a Unicode 2.0 space characters; otherwise `false`.

Description

This method determines if the specified character is a space character according to the Unicode 2.0 specification. A character is considered to be a Unicode space character if and only if it has the general category "Zs", "Zl", or "Zp" in the Unicode specification.

isTitleCase

`public static boolean isTitleCase(char ch)`

Parameters

`ch`

A `char` value to be tested.

Returns

`true` if the specified character is defined as titlecase in the Unicode character set; otherwise `false`.

Description

This method determines whether or not the specified character is a titlecase character. Unicode defines a number of characters that do not have case mappings; if the specified character is one of these characters, the method returns `false`.

Many characters are defined by the Unicode standard as having upper- and lowercase forms. There are some characters defined by the Unicode standard that also have a titlecase form. The glyphs for these characters look like a combination of two Latin letters. The titlecase form of these characters has a glyph that looks like a combination of an uppercase Latin character and a lowercase Latin character; this case should be used when the character appears as the first character of a word in a title. For example, one of the Unicode characters that has a titlecase form looks like the letter `D' followed by the letter `Z'. Here is what the three forms of this letter look like:

Uppercase	`DZ'
Titlecase	`Dz'
Lowercase	`dz'

isUnicodeIdentifierPart

`public static boolean isUnicodeIdentifierPart(char ch)`

Availability

New as of JDK 1.1

Parameters

`ch`

A `char` value to be tested.

Returns

`true` if the specified character can appear after the first character in a Unicode identifier; otherwise `false`.

Description

This method returns `true` if the specified character can appear in a Unicode identifier after the first character. A character is considered part of a Unicode identifier if and only if it is a letter, a digit, a connecting punctuation character (e.g., _), a numeric letter (e.g., a Roman numeral), a combining mark, a nonspacing mark, or an ignorable control character.

isUnicodeIdentifierStart

`public static boolean isUnicodeIdentifierStart(char ch)`

Availability

New as of JDK 1.1

Parameters

`ch`

A `char` value to be tested.

Returns

`true` if the specified character can appear as the first character in a Unicode identifier; otherwise `false`.

Description

This method returns `true` if the specified character can appear in a Unicode identifier as the first character. A character is considered a start of a Unicode identifier if and only if it is a letter.

isUpperCase

`public static boolean isUpperCase(char ch)`

Parameters

`ch`

A `char` value to be tested.

Returns

`true` if the specified character is defined as uppercase in the Unicode character set; otherwise `false`.

Description

This method determines whether or not the specified character is uppercase. Unicode defines a number of characters that do not have case mappings; if the specified character is one of these characters, the method returns `false`.

isWhitespace

`public static boolean isWhitespace(char ch)`

Availability

New as of JDK 1.1

Parameters

`ch`

A `char` value to be tested.

Returns

`true` if the specified character is defined as whitespace according to Java; otherwise `false`.

Description

This method determines whether or not the specified character is whitespace. This method recognizes the following as whitespace:

Unicode category "Zs" except `\u00A0` and `\uFEFF`	Unicode space separators except no-break spaces
Unicode category "Zl"	Unicode line separators
Unicode category "Zp"	Unicode paragraph separators
`\u0009`	Horizontal tab
`\u000A`	Linefeed

`\u000B`	Vertical tab
`\u000C`	Formfeed
`\u000D`	Carriage return
`\u001C`	File separator
`\u001D`	Group separator
`\u001E`	Record separator
`\u001F`	Unit separator

toLowerCase

`public static char toLowerCase(char ch)`

Parameters

`ch`

A `char` value to be converted to lowercase.

Returns

The lowercase equivalent of the specified character, or the character itself if it cannot be converted to lowercase.

Description

This method returns the lowercase equivalent of the specified character value. If the specified character is not uppercase or if it has no lowercase equivalent, the character is returned unmodified. The Unicode attribute table determines if a character has a mapping to a lowercase equivalent.

Some Unicode characters in the range `\u2000` through `\u2FFF` have lowercase mappings. For example, `\u2160` (Roman numeral one) has a lowercase mapping to `\u2170` (small Roman numeral one). The `toLowerCase()` method maps such characters to their lowercase equivalents even though the method `isUpperCase()` does not return `true` for such characters.

toTitleCase

`public static char toTitleCase(char ch)`

Parameters

`ch`

A `char` value to be converted to titlecase.

Returns

The titlecase equivalent of the specified character, or the character itself if it cannot be converted to titlecase.

Description

This method returns the titlecase equivalent of the specified character value. If the specified character has no titlecase equivalent, the character is returned unmodified. The Unicode attribute table is used to determine the character's titlecase equivalent.

Many characters are defined by the Unicode standard as having upper- and lowercase forms. There are some characters defined by the Unicode standard that also have a titlecase form. The glyphs for these characters look like a combination of two Latin letters. The titlecase form of these characters has a glyph that looks like a combination of an uppercase Latin character and a lowercase Latin character; this case should be used when the character appears as the first character of a word in a title. For example, one of the Unicode characters that has a titlecase form looks like the letter `D' followed by the letter `Z'. Here is what the three forms of this letter look like:

Uppercase	`DZ'
Titlecase	`Dz'
Lowercase	`dz'

toUpperCase

`public static char toUpperCase(char ch)`

Parameters

`ch`

A `char` value to be converted to lowercase.

Returns

The uppercase equivalent of the specified character, or the character itself if it cannot be converted to uppercase.

Description

This method returns the uppercase equivalent of the specified character value. If the specified character is not lowercase or if it has no uppercase equivalent, the character is returned unmodified. The Unicode attribute table determines if a character has a mapping to an uppercase equivalent.

Some Unicode characters in the range `\u2000` through `\u2FFF` have uppercase mappings. For example, `\u2170` (small Roman numeral one) has a lowercase mapping to `\u2160` (Roman numeral one). The `toUpperCase()` method maps such characters to their uppercase equivalents even though the method `isLowerCase()` does not return `true` for such characters.

Instance Methods

charValue

`public char charValue()`

Returns

The `char` value contained by the object.

equals

`public boolean equals(Object obj)`

Parameters

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `obj` is an instance of `Character`, and it contains the same value as the object this method is associated with.

hashCode

`public int hashCode()`

Returns

A hashcode based on the `char` value of the object.

Overrides

`Object.hashCode()`

toString

`public String toString()`

Returns

A `String` of length one that contains the character value of the object.

Overrides

`Object.toString()`

Description

This method returns a string representation of the `Character` object.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`Class`, `Object`, `Serializable`

Class

Name

Class

Synopsis

Class Name:

`java.lang.Class`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

`java.io.Seriablizable`

Availability:

JDK 1.0 or later

Description

As of Java 1.1, instances of the `Class` class are used as run-time descriptions of all Java data types, both reference types and primitive types. The `Class` class has also been greatly expanded in 1.1 to provide support for the Reflection API. Prior to 1.1, `Class` just provided run-time descriptions of reference types.

A `Class` object provides considerable information about the data type. You can use the `isPrimitive()` method to find out if a Class object describes a primitive type, while isArray() indicates if the object describes an array type. If a Class object describes a class or interface type, there are numerous methods that return information about the fields, methods, and constructors of the type. This information is returned as java.lang.reflect.Field, java.lang.reflect.Method, and java.lang.reflect.Constructor objects.

There are a number of ways that you can get a Class object for a particular data type:

- If you have an object, you can get the Class object that describes the class of that object by calling the object's getClass()method. Every class inherits this method from the Object class.

- As of Java 1.1, you can get the Class object that describes any Java type using the new class literal syntax. A class literal is simply the name of a type (a class name or a primitive type name) followed by a period and the class keyword. For example:

  ```
  Class s = String.class;
  Class i = int.class;
  Class v = java.util.Vector.class;
  ```

- In Java 1.0, you can get the Class object from the name of a data type using the forName() class method of Class. For example:

  ```
  Class v = Class.forName("java.util.Vector");
  ```

  This technique still works in Java 1.1, but it is more cumbersome (and less efficient) than using a class literal.

You can create an instance of a class using the newInstance() method of a Class object, if the class has a constructor that takes no arguments.

The Class class has no public constructors; it cannot be explicitly instantiated. Class objects are normally created by the ClassLoader class or a ClassLoader object.

Class Summary

```
public final class java.lang.Class extends java.lang.Object
                                   implements java.io.Serializable {
    // Class Methods
    public static native Class forName(String className);
    // Instance Methods
    public Class[] getClasses();                             // New in 1.1
    public native ClassLoader getClassLoader();
    public native Class getComponentType();                  // New in 1.1
    public Constructor
           getConstructor(Class[] parameterTypes);           // New in 1.1
    public Constructor[] getConstructors();                  // New in 1.1
    public Class[] getDeclaredClasses();                     // New in 1.1
    public Constructor
           getDeclaredConstructor(Class[] parameterTypes); // New in 1.1
    public Constructor[] getDeclaredConstructors();          // New in 1.1
```

```
    public Field getDeclaredField(String name);             // New in 1.1
    public Field[] getDeclaredFields();                     // New in 1.1
    public Method getDeclaredMethod(String name,
                  Class[] parameterTypes)                   // New in 1.1
    public Method[] getDeclaredMethods()                    // New in 1.1
    public Class getDeclaringClass();                       // New in 1.1
    public Field getField(String name);                     // New in 1.1
    public Field[] getFields();                             // New in 1.1
    public native Class[] getInterfaces();
    public Method getMethod(String name,
                  Class[] parameterTypes);                  // New in 1.1
    public Method[] getMethods();                           // New in 1.1
    public native int getModifiers();                       // New in 1.1
    public native String getName();
    public URL getResource(String name);                    // New in 1.1
    public InputStream getResourceAsStream(String name);   // New in 1.1
    public native Object[] getSigners();                    // New in 1.1
    public native Class getSuperclass();
    public native boolean isArray();                        // New in 1.1
    public native boolean isAssignableFrom(Class cls);     // New in 1.1
    public native boolean isInstance(Object obj);          // New in 1.1
    public native boolean isInterface();
    public native boolean isPrimitive();                    // New in 1.1
    public native Object newInstance();
    public String toString();
}
```

Class Methods

forName

`public static Class forName(String className) throws ClassNotFoundException`

Parameters

`className`

Name of a class qualified by the name of its package. If the class is defined inside of another class, all dots (`.`) that separate the top-level class name from the class to load must be changed to dollar signs ($) for the name to be recognized.

Returns

A `Class` object that describes the named class.

Throws

`ClassNotFoundException`

If the class cannot be loaded because it cannot be found.

Description

This method dynamically loads a class if it has not already been loaded. The method returns a `Class` object that describes the named class.

The most common use of `forName()` is for loading classes on the fly when an application wants to use classes it wasn't built with. For example, a web browser uses this technique. When a browser needs to load an applet, the browser calls `Class.forName()` for the applet. The method loads the class if it has not already been loaded and returns the `Class` object that encapsulates the class. The browser then creates an instance of the applet by calling the `Class` object's `newInstance()` method.

When a class is loaded using a `ClassLoader` object, any classes loaded at the instigation of that class are also loaded using the same `ClassLoader` object. This method implements that security policy by trying to find a `ClassLoader` object to load the named class. The method searches the stack for the most recently invoked method associated with a class that was loaded using a `ClassLoader` object. If such a class is found, the `ClassLoader` object associated with that class is used.

Instance Methods

getClasses

`public Class[] getClasses()`

Availability

New as of JDK 1.1

Returns

An array of `Class` objects that contains the `public` classes and interfaces that are members of this class.

Description

If this `Class` object represents a reference type, this method returns an array of `Class` objects that lists all of the `public` classes and interfaces that are members of this class or interface. The list includes `public` classes and interfaces that are inherited from superclasses and that are defined by this class or interface. If there are no `public` member classes or interfaces, or if this `Class` represents a primitive type, the method returns an array of length `0`.

As of Java 1.1.1, this method always returns an array of length `0`, no matter how many `public` member classes this class or interface actually declares.

getClassLoader

`public native ClassLoader getClassLoader()`

Returns

The `ClassLoader` object used to load this class or `null` if this class was not loaded with a `ClassLoader`.

Description

This method returns the `ClassLoader` object that was used to load this class. If this class was not loaded with a `ClassLoader`, `null` is returned.

This method is useful for making sure that a class gets loaded with the same class loader as was used for loading this `Class` object.

getComponentType

`public native Class getComponentType()`

Availability

New as of JDK 1.1

Returns

A `Class` object that describes the component type of this class if it is an array type.

Description

If this `Class` object represents an array type, this method returns a `Class` object that describes the component type of the array. If this `Class` does not represent an array type, the method returns `null`.

getConstructor

`public Constructor getConstructor(Class[] parameterTypes) throws NoSuchMethodException, SecurityException`

Availability

New as of JDK 1.1

Parameters

`parameterTypes`

An array of `Class` objects that describes the parameter types, in declared order, of the constructor.

Returns

A `Constructor` object that reflects the specified `public` constructor of this class.

Throws

`NoSuchMethodException`

If the specified constructor does not exist.

`SecurityException`

If the `checkMemberAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

If this `Class` object represents a class, this method returns a `Constructor` object that reflects the specified `public` constructor of this class. The constructor is located by searching all of the constructors of the class for a `public` constructor that has exactly the same formal parameters as specified. If this `Class` does not represent a class, the method returns `null`.

getConstructors

`public Constructor[] getConstructors() throws SecurityException`

Availability

New as of JDK 1.1

Returns

An array of `Constructor` objects that reflect the `public` constructors of this class.

Throws

`SecurityException`

If the `checkMemberAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

If this `Class` object represents a class, this method returns an array of `Constructor` objects that reflect the `public` constructors of this class. If there are no `public` constructors, or if this `Class` does not represent a class, the method returns an array of length `0`.

getDeclaredClasses

`public Class[] getDeclaredClasses() throws SecurityException`

Availability

New as of JDK 1.1

Returns

An array of `Class` objects that contains all of the declared classes and interfaces that are members of this class.

Throws

`SecurityException`

If the `checkMemberAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

If this `Class` object represents a reference type, this method returns an array of `Class` objects that lists all of the classes and interfaces that are members of this class or interface. The list includes `public`, `protected`, default access, and `private` classes and interfaces that are defined by this class or interface, but it excludes classes and interfaces inherited from superclasses. If there are no such member classes or interfaces, or if this `Class` represents a primitive type, the method returns an array of length `0`.

As of Java 1.1.1, this method always returns an array of length `0`, no matter how many member classes this class or interface declares.

getDeclaredConstructor

```
 public Constructor getDeclaredConstructor(Class[] parameterTypes) throws
NoSuchMethodException, SecurityException
```

Availability

New as of JDK 1.1

Parameters

`parameterTypes`

An array of `Class` objects that describes the parameter types, in declared order, of the constructor.

Returns

A `Constructor` object that reflects the specified declared constructor of this class.

Throws

`NoSuchMethodException`

If the specified constructor does not exist.

`SecurityException`

If the `checkMemberAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

If this `Class` object represents a class, this method returns a `Constructor` object that reflects the specified declared constructor of this class. The constructor is located by searching all of the constructors of the class for a `public`, `protected`, default access, or `private` constructor that has exactly the same formal parameters as specified. If this `Class` does not represent a class, the method returns `null`.

getDeclaredConstructors

```
public Constructor[] getDeclaredConstructors() throws SecurityException
```

Availability

New as of JDK 1.1

Returns

An array of `Constructor` objects that reflect the declared constructors of this class.

Throws

`SecurityException`

If the `checkMemberAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

If this `Class` object represents a class, this method returns an array of `Constructor` objects that reflect the `public`, `protected`, default access, and `private` constructors of this class. If there are no declared constructors, or if this `Class` does not represent a class, the method returns an array of length `0`.

getDeclaredField

```
public Field getDeclaredField(String name) throws NoSuchFieldException,
SecurityException
```

Availability

New as of JDK 1.1

Parameters

name

The simple name of the field.

Returns

A Field object that reflects the specified declared field of this class.

Throws

NoSuchFieldException

If the specified field does not exist.

SecurityException

If the checkMemberAccess() method of the SecurityManager throws a SecurityException.

Description

If this Class object represents a class or interface, this method returns a Field object that reflects the specified declared field of this class. The field is located by searching all of the fields of the class (but not inherited fields) for a public, protected, default access, or private field that has the specified simple name. If this Class does not represent a class or interface, the method returns null.

getDeclaredFields

public Field[] getDeclaredFields() throws SecurityException

Availability

New as of JDK 1.1

Returns

An array of Field objects that reflect the declared fields of this class.

Throws

SecurityException

If the checkMemberAccess() method of the SecurityManager throws a SecurityException.

Description

If this Class object represents a class or interface, this method returns an array of Field objects that reflect the public, protected, default access, and private fields declared by this class, but excludes inherited fields. If there are no declared fields, or if this `Class` does not represent a class or interface, the method returns an array of length `0`.

This method does not reflect the implicit `length` field for array types. The methods of the class `Array` should be used to manipulate array types.

getDeclaredMethod

`public Method getDeclaredMethod(String name, Class[] parameterTypes) throws NoSuchMethodException, SecurityException`

Availability

New as of JDK 1.1

Parameters

`name`

The simple name of the method.

`parameterTypes`

An array of `Class` objects that describes the parameter types, in declared order, of the method.

Returns

A `Method` object that reflects the specified declared method of this class.

Throws

`NoSuchMethodException`

If the specified method does not exist.

`SecurityException`

If the `checkMemberAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

If this `Class` object represents a class or interface, this method returns a `Method` object that reflects the specified declared method of this class. The method is located by searching all of the methods of the class (but not inherited methods) for a `public`, `protected`, default access, or `private` method that has the specified simple name and exactly the same formal parameters as specified. If this `Class` does not represent a class or interface, the method returns `null`.

getDeclaredMethods

public Method[] getDeclaredMethods() throws SecurityException

Availability

New as of JDK 1.1

Returns

An array of `Method` objects that reflect the declared methods of this class.

Throws

`SecurityException`

If the `checkMemberAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

If this `Class` object represents a class or interface, this method returns an array of `Method` objects that reflect the `public`, `protected`, default access, and `private` methods declared by this class, but excludes inherited methods. If there are no declared methods, or if this `Class` does not represent a class or interface, the method returns an array of length `0`.

getDeclaringClass

`public Class getDeclaringClass()`

Availability

New as of JDK 1.1

Returns

A `Class` object that represents the declaring class if this class is a member of another class.

Description

If this `Class` object represents a class or interface that is a member of another class or interface, this method returns a `Class` object that describes the declaring class or interface. If this class or interface is not a member of another class or interface, or if it represents a primitive type, the method returns `null`.

getField

`public Field getField(String name) throws NoSuchFieldException, SecurityException`

Availability

New as of JDK 1.1

Parameters

`name`

The simple name of the field.

Returns

A `Field` object that reflects the specified `public` field of this class.

Throws

`NoSuchFieldException`

If the specified field does not exist.

`SecurityException`

If the `checkMemberAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

If this `Class` object represents a class or interface, this method returns a `Field` object that reflects the specified `public` field of this class. The field is located by searching all of the fields of the class, including any inherited fields, for a `public` field that has the specified simple name. If this `Class` does not represent a class or interface, the method returns `null`.

getFields

`public Field[] getFields() throws SecurityException`

Availability

New as of JDK 1.1

Returns

An array of `Field` objects that reflect the `public` fields of this class.

Throws

`SecurityException`

If the `checkMemberAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

If this `Class` object represents a class or interface, this method returns an array of `Field` objects that reflect the `public` fields declared by this class and any inherited `public` fields. If there are no `public` fields, or if this `Class` does not represent a class or interface, the method returns an array of length `0`.

This method does not reflect the implicit `length` field for array types. The methods of the class `Array` should be used to manipulate array types.

getInterfaces

`public native Class[] getInterfaces()`

Returns

An array of the interfaces implemented by this class or extended by this interface.

Description

If the `Class` object represents a class, this method returns an array that refers to all of the interfaces that the class implements. The order of the interfaces referred to in the array is the same as the order in the class declaration's `implements` clause. If the class does not implement any interfaces, the length of the returned array is 0.

If the object represents an interface, this method returns an array that refers to all of the interfaces that this interface extends. The interfaces occur in the order they appear in the interface declaration's `extends` clause. If the interface does not extend any interfaces, the length of the returned array is 0.

If the object represents a primitive or array type, the method returns an array of length 0.

getMethod

`public Method getMethod(String name, Class[] parameterTypes) throws NoSuchMethodException, SecurityException`

Availability

New as of JDK 1.1

Parameters

`name`

The simple name of the method.

`parameterTypes`

An array of `Class` objects that describes the parameter types, in declared order, of the method.

Returns

A `Method` object that reflects the specified `public` method of this class.

Throws

`NoSuchMethodException`

If the specified method does not exist.

`SecurityException`

If the `checkMemberAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

If this `Class` object represents a class or interface, this method returns a `Method` object that reflects the specified `public` method of this class. The method is located by searching all of the methods of the class, including any inherited methods, for a `public` method that has the specified simple name and exactly the same formal parameters as specified. If this `Class` does not represent a class or interface, the method returns `null`.

getMethods

`public Method[] getMethods() throws SecurityException`

Availability

New as of JDK 1.1

Returns

An array of `Method` objects that reflect the `public` methods of this class.

Throws

`SecurityException`

If the `checkMemberAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

If this `Class` object represents a class or interface, this method returns an array of `Method` objects that reflect the `public` methods declared by this class and any inherited `public` methods. If there are no `public` methods or if this `Class` doesn't represent a class or interface, the method returns an array of length `0`.

getModifiers

`public native int getModifiers()`

Availability

New as of JDK 1.1

Returns

An integer that represents the modifier keywords used to declare this class.

Description

If this `Class` object represents a class or interface, this method returns an integer value that represents the modifiers used to declare the class or interface. The `Modifier` class should be used to decode the returned value.

getName

`public native String getName()`

Returns

The fully qualified name of this class or interface.

Description

This method returns the fully qualified name of the type represented by this `Class` object.

If the object represents the class of an array, the method returns a `String` that contains as many left square brackets as there are dimensions in the array, followed by a code that indicates the type of element contained in the base array. Consider the following:

```
(new int [3][4][5]).getClass().getName()
```

This code returns "`[[[I`". The codes used to indicate the element type are as follows:

Code	**Type**
`[`	array
`B`	`byte`
`C`	`char`
`d`	`double`
`F`	`float`
`I`	`int`
`J`	`long`

`L` *fully_qualified_class_name*	class or interface
`S`	`short`
`Z`	`boolean`

getResource

`public URL getResource(String name)`

Availability

New as of JDK 1.1

Parameters

`name`

A resource name.

Returns

A `URL` object that is connected to the specified resource or `null` if the resource cannot be found.

Description

This method finds a resource with the given name for this `Class` object and returns a `URL` object that is connected to the resource. The rules for searching for a resource associated with a class are implemented by the `ClassLoader` for the class; this method simply calls the `getResource()` method of the `ClassLoader`. If this class does not have a `ClassLoader` (i.e., it is a system class), the method calls the `ClassLoader.getSystemResource()` method.

getResourceAsStream

`public InputStream getResourceAsStream(String name)`

Availability

New as of JDK 1.1

Parameters

`name`

A resource name.

Returns

An `InputStream` object that is connected to the specified resource or `null` if the resource cannot be found.

Description

This method finds a resource with the given name for this `Class` object and returns an `InputStream` object that is connected to the resource. The rules for searching for a resource associated with a class are implemented by the `ClassLoader` for the class; this method simply calls the `getResourceAsStream()` method of the `ClassLoader`. If this class does not have a `ClassLoader` (i.e., it is a system class), the method calls the `ClassLoader.getSystemResourceAsStream()` method.

getSigners

`public native Object[] getSigners()`

Availability

New as of JDK 1.1

Returns

An array of `Object`s that represents the signers of this class.

Description

This method returns an array of objects that represents the digital signatures for this class.

getSuperclass

`public native Class getSuperclass()`

Returns

The superclass of this class or `null` if there is no superclass.

Description

If the `Class` object represents a class other than `Object`, this method returns the `Class` object that represents its superclass. If the object represents an interface, the `Object` class, or a primitive type, the method returns `null`.

isArray

`public native boolean isArray()`

Availability

New as of JDK 1.1

Returns

`true` if this object describes an array type; otherwise `false`.

isAssignableFrom

`public native boolean isAssignableFrom(Class cls)`

Availability

New as of JDK 1.1

Parameters

`cls`

A `Class` object to be tested.

Returns

`true` if the type represented by `cls` is assignable to the type of this class: otherwise `false`.

Throws

`NullPointerException`

If `cls` is `null`.

Description

This method determines whether or not the type represented by `cls` is assignable to the type of this class. If this class represents a class, this class must be the same as `cls` or a superclass of `cls`. If this class represents an interface, this class must be the same as `cls` or a superinterface of `cls`. If this class represents a primitive type, this class must be the same as `cls`.

isInstance

`public native boolean isInstance(Object obj)`

Availability

New as of JDK 1.1

Parameters

`obj`

An `Object` to be tested.

Returns

`true` if `obj` can be cast to the reference type specified by this class; otherwise `false`.

Throws

`NullPointerException`

If `obj` is `null`.

Description

This method determines whether or not the object represented by `obj` can be cast to the type of this class object without causing a `ClassCastException`. This method is the dynamic equivalent of the `instanceof` operator.

isInterface

`public native boolean isInterface()`

Returns

`true` if this object describes an interface; otherwise `false`.

isPrimitive

`public native boolean isPrimitive()`

Availability

New as of JDK 1.1

Returns

`true` if this object describes a primitive type; otherwise `false`.

newInstance

`public native Object newInstance () throws InstantiationException, IllegalAccessException`

Returns

A reference to a new instance of this class.

Throws

`InstantiationException`

If the `Class` object represents an interface or an `abstract` class.

`IllegalAccessException`

If the class or an initializer is not accessible.

Description

This method creates a new instance of this class by performing these steps:

1. It creates a new object of the class represented by the `Class` object.
2. It calls the constructor for the class that takes no arguments.
3. It returns a reference to the initialized object.

The `newInstance()` method is useful for creating an instance of a class that has been dynamically loaded using the `forName()` method.

The reference returned by this method is usually cast to the type of object that is instantiated.

The `newInstance()` method can throw objects that are not instances of the classes it is declared to throw. If the constructor invoked by `newInstance()` throws an exception, the exception is thrown by `newInstance()` regardless of the class of the object.

toString

`public String toString()`

Returns

A `String` that contains the name of the class with either `'class'` or `'interface'` prepended as appropriate.

Overrides

`Object.toString()`

Description

This method returns a string representation of the `Class` object.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals()`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`Array`, `ClassLoader`, `ClassNotFoundException`, `Constructor`, `Field`, `IllegalAccessException`, `InputStream` `InstantiationException`, `Method`, `Modifier`, `NoSuchFieldException`, `NoSuchMethodException`, `Object`, `SecurityException`, `SecurityManager`, `URL`

PREVIOUS Character | HOME BOOK INDEX | NEXT ClassCastException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

ClassCastException

Name

ClassCastException

Synopsis

Class Name:

`java.lang.ClassCastException`

Superclass:

`java.lang.RuntimeException`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

A `ClassCastException` is thrown when there is an attempt to cast a reference to an object to an inappropriate type.

Class Summary

```
public class java.lang.ClassCastException
             extends java.lang.RuntimeException {
  // Constructors
  public ClassCastException();
  public ClassCastException(String s);
}
```

Constructors

ClassCastException

public ClassCastException()

Description

This constructor creates a `ClassCastException` with no associated detail message.

public ClassCastException(String s)

Parameters

`s`

The detail message.

Description

This constructor creates a `ClassCastException` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`

`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Exception`, `RuntimeException`, `Throwable`

ClassCircularityError

Name

ClassCircularityError

Synopsis

Class Name:

`java.lang.ClassCircularityError`

Superclass:

`java.lang.LinkageError`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

A `ClassCircularityError` is thrown when a circular reference among classes is detected during class initialization.

Class Summary

```
public class java.lang.ClassCircularityError
             extends java.lang.LinkageError {
  // Constructors
  public ClassCircularityError();
  public ClassCircularityError(String s);
}
```

Constructors

ClassCircularityError

public ClassCircularityError()

Description

This constructor creates a `ClassCircularityError` with no associated detail message.

public ClassCircularityError(String s)

Parameters

`s`

The detail message.

Description

This constructor creates a `ClassCircularityError` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`

`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Error`, `LinkageError`, `Throwable`

PREVIOUS | HOME | NEXT
ClassCastException | BOOK INDEX | ClassFormatError

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

ClassFormatError

Name

ClassFormatError

Synopsis

Class Name:

`java.lang.ClassFormatError`

Superclass:

`java.lang.LinkageError`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

A `ClassFormatError` is thrown when an error is detected in the format of a file that contains a class definition.

Class Summary

```
public class java.lang.ClassFormatError extends java.lang.LinkageError {
  // Constructors
  public ClassFormatError();
  public ClassFormatError(String s);
}
```

Constructors

ClassFormatError

public ClassFormatError()

Description

This constructor creates a `ClassFormatError` with no associated detail message.

public ClassFormatError(String s)

Parameters

`s`

The detail message.

Description

This constructor creates a `ClassFormatError` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`

`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Error`, `LinkageError`, `Throwable`

PREVIOUS HOME NEXT
ClassCircularityError BOOK INDEX ClassLoader

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

ClassLoader

Name

ClassLoader

Synopsis

Class Name:

`java.lang.ClassLoader`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The `ClassLoader` class provides a mechanism for Java to load classes over a network or from any source other than the local filesystem. The default class-loading mechanism loads classes from files found relative to directories specified by the `CLASSPATH` environment variable. This default mechanism does not use an instance of the `ClassLoader` class.

An application can implement another mechanism for loading classes by declaring a subclass of the `abstract ClassLoader` class. A subclass of `ClassLoader` must override the `loadClass()` to define a class-loading policy.

This method implements any sort of security that is necessary for the class-loading mechanism. The other methods of ClassLoader are final, so they cannot be overridden.

A ClassLoader object is typically used by calling its loadClass() method to explicitly load a top-level class, such as a subclass of Applet. The ClassLoader that loads the class becomes associated with the class; it can be obtained by calling the getClassLoader() method of the Class object that represents the class.

Once a class is loaded, it must be resolved before it can be used. Resolving a class means ensuring that all of the other classes it references are loaded. In addition, all of the classes that they reference must be loaded, and so on, until all of the needed classes have been loaded. Classes are resolved using the resolveClass() method of the ClassLoader object that loaded the initial class. This means that when a ClassLoader object is explicitly used to load a class, the same ClassLoader is used to load all of the classes that it references, directly or indirectly.

Classes loaded using a ClassLoader object may attempt to load additional classes without explicitly using a ClassLoader object. They can do this by calling the Class class' forName() method. However, in such a situation, a ClassLoader object is implicitly used. See the description of Class.forName() for more information.

Java identifies a class by a combination of its fully qualified name and the class loader that was used to load the class. If you write a subclass of ClassLoader, it should not attempt to directly load local classes. Instead, it should call findSystemClass(). A local class that is loaded directly by a ClassLoader is considered to be a different class than the same class loaded by findSystemClass(). This can lead to having two copies of the same class loaded, which can cause a number of inconsistencies. For example, the class' equals() method may decide that the same object is not equal to itself.

Class Summary

```
public abstract class java.lang.ClassLoader extends java.lang.Object {
    // Constructors
    protected ClassLoader();
    // Class Methods
    public static final URL
           getSystemResource(String name);                 // New in 1.1
    public static final InputStream
           getSystemResourceAsStream(String name);         // New in 1.1
    // Public Instance Methods
    public URL getResource(String name);                   // New in 1.1
    public InputStream getResourceAsStream(String name);    // New in 1.1
    public Class loadClass(String name);                   // New in 1.1

    // Protected Instance Methods
    protected final Class defineClass(byte data[],
              int offset, int length);              // Deprecated in 1.1
    protected final Class defineClass(String name,
              byte[] data, int offset, int length);        // New in 1.1
    protected final Class findLoadedClass(String name);     // New in 1.1
    protected final Class findSystemClass(String name);
    protected abstract Class loadClass(String name, boolean resolve);
    protected final void resolveClass(Class c);
    protected final void setSigners(Class cl,
              Object[] signers);                           // New in 1.1
}
```

Constructors

ClassLoader

`protected ClassLoader()`

Throws

`SecurityException`

If there is a `SecurityManager` object installed and its `checkCreateClassLoader()` method throws a `SecurityException` when called by this constructor.

Description

Initializes a `ClassLoader` object. Because `ClassLoader` is an `abstract` class, only subclasses of the class can access this constructor.

Class Methods

getSystemResource

`public static final URL getSystemResource(String name)`

Availability

New as of JDK 1.1

Parameters

`name`

A system resource name.

Returns

A `URL` object that is connected to the specified system resource or `null` if the resource cannot be found.

Description

This method finds a system resource with the given name and returns a `URL` object that is connected to the resource. The resource name can be any system resource.

getSystemResourceAsStream

`public static final InputStream getSystemResourceAsStream(String name)`

Availability

New as of JDK 1.1

Parameters

`name`

A system resource name.

Returns

An `InputStream` object that is connected to the specified system resource or `null` if the resource cannot be found.

Description

This method finds a system resource with the given name and returns an `InputStream` object that is connected to the resource. The resource name can be any system resource.

Public Instance Methods

getResource

`public URL getResource(String name)`

Availability

New as of JDK 1.1

Parameters

`name`

A resource name.

Returns

A `URL` object that is connected to the specified resource or `null` if the resource cannot be found.

Description

This method finds a resource with the given name and returns a `URL` object that is connected to the resource.

A resource is a file that contains data (e.g., sound, images, text) and it can be part of a package. The name of a resource is a sequence of identifiers separated by "`/`". For example, a resource might have the name *help/american/logon.html* . System resources are found on the host machine using the conventions of the host implementation. For example, the "`/`" in the resource name may be treated as a path separator, with the entire resource name treated as a relative path to be found under a directory in `CLASSPATH`.

The implementation of `getResource()` in `ClassLoader` simply returns `null`. A subclass can override this method to provide more useful functionality.

getResourceAsStream

`public InputStream getResourceAsStream(String name)`

Availability

New as of JDK 1.1

Parameters

`name`

A resource name.

Returns

An `InputStream` object that is connected to the specified resource or `null` if the resource cannot be found.

Description

This method finds a resource with the given name and returns an `InputStream` object that is connected to the resource.

A resource is a file that contains data (e.g., sound, images, text) and it can be part of a package. The name of a resource is a sequence of identifiers separated by `` `/' ``. For example, a resource might have the name *help/american/logon.html*. System resources are found on the host machine using the conventions of the host implementation. For example, the `` `/' `` in the resource name may be treated as a path separator, with the entire resource name treated as a relative path to be found under a directory in `CLASSPATH`.

The implementation of `getResourceAsStream()` in `ClassLoader` simply returns `null`. A subclass can override this method to provide more useful functionality.

loadClass

public Class loadClass(String name) throws ClassNotFoundException

Availability

New as of JDK 1.1

Parameters

`name`

The name of the class to be returned. The class name should be qualified by its package name. The lack of an explicit package name specifies that the class is part of the default package.

Returns

The `Class` object for the specified class.

Throws

`ClassNotFoundException`

If it cannot find a definition for the named class.

Description

This method loads the named class by calling `loadClass(name, true)`.

Protected Instance Methods

defineClass

`protected final Class defineClass(byte data[], int offset, int length)`

Availability

Deprecated as of JDK 1.1

Parameters

`data`

An array that contains the byte codes that define a class.

`offset`

The offset in the array of byte codes.

`length`

The number of byte codes in the array.

Returns

The newly created `Class` object.

Throws

`ClassFormatError`

If the `data` array does not constitute a valid class definition.

Description

This method creates a `Class` object from the byte codes that define the class. Before the class can be used, it must be resolved. The method is intended to be called from an implementation of the `loadClass()` method.

Note that this method is deprecated as of Java 1.1. You should use the version of `defineClass()` that takes a `name` parameter and is therefore more secure.

`protected final Class defineClass(String name, byte data[], int offset, int length)`

Availability

New as of JDK 1.1

Parameters

`name`

The expected name of the class to be defined or `null` if it is not known. The class name should be qualified by its package name. The lack of an explicit package name specifies that the class is part of the default package.

`data`

An array that contains the byte codes that define a class.

`offset`

The offset in the array of byte codes.

`length`

The number of byte codes in the array.

Returns

The newly created `Class` object.

Throws

`ClassFormatError`

If the `data` array does not constitute a valid class definition.

Description

This method creates a `Class` object from the byte codes that define the class. Before the class can be used, it must be resolved. The method is intended to be called from an implementation of the `loadClass()` method.

findLoadedClass

`protected final Class findLoadedClass(String name)`

Availability

New as of JDK 1.1

Parameters

`name`

The name of the class to be returned. The class name should be qualified by its package name. The lack of an explicit package name specifies that the class is part of the default package.

Returns

The `Class` object for the specified loaded class or `null` if the class cannot be found.

Description

This method finds the specified class that has already been loaded.

findSystemClass

`protected final Class findSystemClass(String name) throws ClassNotFoundException`

Parameters

`name`

The name of the class to be returned. The class name should be qualified by its package name. The lack of an explicit package name specifies that the class is part of the default package.

Returns

The `Class` object for the specified system class.

Throws

`ClassNotFoundException`

If the default class-loading mechanism cannot find a definition for the class.

`NoClassDefFoundError`

If the default class-loading mechanism cannot find the class.

Description

This method finds and loads a system class if it has not already been loaded. A *system class* is a class that is loaded by the default class-loading mechanism from the local filesystem. An implementation of the `loadClass()` method typically calls this method to attempt to load a class from the locations specified by the `CLASSPATH` environment variable.

loadClass

`protected abstract Class loadClass(String name, boolean resolve) throws ClassNotFoundException`

Parameters

`name`

The name of the class to be returned. The class name should be qualified by its package name. The lack of an explicit package name specifies that the class is part of the default package.

`resolve`

Specifies whether or not the class should be resolved by calling the `resolveClass()` method.

Returns

The `Class` object for the specified class.

Throws

`ClassNotFoundException`

If it cannot find a definition for the named class.

Description

An implementation of this `abstract` method loads the named class and returns its `Class` object. It is permitted and encouraged for an implementation to cache the classes it loads, rather than load one each time the method is called. An implementation of this method should do at least the following:

1. Load the byte codes that comprise the class definition into a `byte[]`.
2. Call the `defineClass()` method to create a `Class` object to represent the class definition.
3. If the `resolve` parameter is `true`, call the `resolveClass()` method to resolve the class.

If an implementation of this method caches the classes that it loads, it is recommended that it use an instance of the `java.util.Hashtable` to implement the cache.

resolveClass

`protected final void resolveClass(Class c)`

Parameters

`c`

The `Class` object for the class to be resolved.

Description

This method resolves the given `Class` object. Resolving a class means ensuring that all of the other classes that the `Class` object references are loaded. In addition, all of the classes that they reference must be loaded, and so on, until all of the needed classes have been loaded.

The `resolveClass()` method should be called by an implementation of the `loadClass()` method when the value of the `loadClass()` method's `resolve` parameter is `true`.

setSigners

`protected final void setSigners(Class cl, Object[] signers)`

Availability

New as of JDK 1.1

Parameters

`cl`

The `Class` object for the class to be signed.

`signers`

An array of `Object`s that represents the signers of this class.

Description

This method specifies the objects that represent the digital signatures for this class.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`

`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Class`, `ClassNotFoundException`, `InputStream`, `NoClassDefFoundError`, `Object`, `SecurityException`, `SecurityManager`, `URL`

ClassNotFoundException

Name

ClassNotFoundException

Synopsis

Class Name:

`java.lang.ClassNotFoundException`

Superclass:

`java.lang.Exception`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

A `ClassNotFoundException` is thrown to indicate that a class to be loaded cannot be found.

Class Summary

```
public class java.lang.ClassNotFoundException extends java.lang.Exception {
  // Constructors
  public ClassNotFoundException();
  public ClassNotFoundException(String s);
}
```

Constructors

ClassNotFoundException

public ClassNotFoundException()

Description

This constructor creates a `ClassNotFoundException` with no associated detail message.

public ClassNotFoundException(String s)

Parameters

`s`

The detail message.

Description

This constructor creates a `ClassNotFoundException` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`

`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Exception`, `Throwable`

PREVIOUS HOME NEXT
ClassLoader BOOK INDEX Cloneable

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Cloneable

Name

Cloneable

Synopsis

Interface Name:

> `java.lang.Cloneable`

Super-interface:

> None

Immediate Sub-interfaces:

> `java.text.CharacterIterator`

Implemented by:

> `java.awt.GridBagConstraints`, `java.awt.Insets`,
>
> `java.awt.image.ImageFilter`,
>
> `java.text.BreakIterator`,

java.text.Collator, java.text.DateFormat,

java.text.DateFormatSymbols,

java.text.DecimalFormatSymbols,

java.text.Format, java.text.NumberFormat,

java.util.BitSet, java.util.Calendar,

java.util.Date, java.util.Hashtable,

java.util.Locale, java.util.TimeZone,

java.util.Vector

Availability:

JDK 1.0 or later

Description

The Cloneable interface provides no functionality; it declares no methods or variables. This interface is simply provided as a way of indicating that an object can be cloned (that is, copied). A class that is declared as implementing this interface is assumed to have overridden the Object class' implementation of clone() with an implementation that can successfully clone instances of the class. The implementation of clone() that is provided by the Object class simply throws a CloneNotSupportedException.

Interface Declaration

```
public interface java.lang.Cloneable {
}
```

See Also

BitSet, BreakIterator, Calendar, CloneNoSupportedException, Collator, Date, DateFormat, DateFormatSymbols, DecimalFormatSymbols, Format, Hashtable, Locale, NumberFormat, TimeZone, Vector

PREVIOUS | HOME | NEXT
ClassNotFoundException | BOOK INDEX | CloneNotSupportedException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

CloneNotSupportedException

Name

CloneNotSupportedException

Synopsis

Class Name:

`java.lang.CloneNotSupportedException`

Superclass:

`java.lang.Exception`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

A `CloneNotSupportedException` is thrown when the `clone()` method has been called for an object that does not implement the `Cloneable` interface and thus cannot be cloned.

Class Summary

```
public class java.lang.CloneNotSupportedException
             extends java.lang.Exception {
  // Constructors
  public CloneNotSupportedException();
  public CloneNotSupportedException(String s);
}
```

Constructors

CloneNotSupportedException

public CloneNotSupportedException()

Description

This constructor creates a `CloneNotSupportedException` with no associated detail message.

public CloneNotSupportedException(String s)

Parameters

`s`

The detail message.

Description

This constructor creates a `CloneNotSupportedException` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`

`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Exception`, `Throwable`

PREVIOUS | HOME | NEXT
Cloneable | BOOK INDEX | Compiler

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Compiler

Name

Compiler

Synopsis

Class Name:

`java.lang.Compiler`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The `Compiler` class encapsulates a facility for compiling Java classes to native code. As provided by Sun, the methods of this class do not actually do anything. However, if the system property `java.compiler` has been defined and if the method `System.loadLibrary()` is able to load the library named by the property, the methods of this class use the implementations provided in the library.

The `Compiler` class has no `public` constructors, so it cannot be instantiated.

Class Summary

```
public final class java.lang.Compiler extends java.lang.Object {
    // Class Methods
    public static native Object command(Object any);
    public static native boolean compileClass(Class clazz);
    public static native boolean compileClasses(String string);
    public static native void disable();
    public static native void enable();
}
```

Class Methods

command

`public static native Object command(Object any)`

Parameters

`any`

The permissible value and its meaning is determined by the compiler library.

Returns

A value determined by the compiler library, or `null` if no compiler library is loaded.

Description

This method directs the compiler to perform an operation specified by the given argument. The available operations, if any, are determined by the compiler library.

compileClass

`public static native boolean compileClass(Class clazz)`

Parameters

`clazz`

The class to be compiled to native code.

Returns

`true` if the compilation succeeds, or `false` if the compilation fails or no compiler library is loaded.

Description

This method requests the compiler to compile the specified class to native code.

compileClasses

`public static native boolean compileClasses(String string)`

Parameters

`string`

A string that specifies the names of the classes to be compiled.

Returns

`true` if the compilation succeeds or `false` if the compilation fails or no compiler library is loaded.

Description

This method requests the compiler to compile all of the classes named in the string.

disable

```
public static native void disable()
```

Description

This method disables the compiler if one is loaded.

enable

```
public static native void enable()
```

Description

This method enables the compiler if one is loaded.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Object`, `System`

PREVIOUS HOME NEXT
CloneNotSupportedException BOOK INDEX Double

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Double

Name

Double

Synopsis

Class Name:

`java.lang.Double`

Superclass:

`java.lang.Number`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The `Double` class provides an object wrapper for a `double` value. This is useful when you need to treat a `double` value as an object. For example, there are a number of utility methods that take a reference to an `Object` as one of their arguments. You cannot specify a `double` value for one of these arguments, but you can provide a reference to a `Double` object that encapsulates the `double` value. Furthermore, as of JDK 1.1, the `Double` class is necessary to support the Reflection API and class literals.

In Java, `double` values are represented using the IEEE 754 format. The `Double` class provides constants for the three special values that are mandated by this format: `POSITIVE_INFINITY`, `NEGATIVE_INFINITY`, and `NaN` (not-a-number).

The `Double` class also provides some utility methods, such as methods for determining whether a `double` value is an infinity value or NaN, for converting `double` values to other primitive types, and for converting a `double` to a `String` and vice versa.

Class Summary

```
public final class java.lang.Double extends java.lang.Number {
    // Constants
    public final static double MAX_VALUE;
    public final static double MIN_VALUE;
    public final static double NaN;
    public final static double NEGATIVE_INFINITY;
    public final static double POSITIVE_INFINITY;
    public final static Class TYPE;                    // New in 1.1
    // Constructors
    public Double(double value);
    public Double(String s);
    // Class Methods
    public static native long doubleToLongBits(double value);
    public static boolean isInfinite(double v);
    public static boolean isNaN(double v);
    public static native double longBitsToDouble(long bits);
    public static String toString(double d);
    public static Double valueOf(String s);
    // Instance Methods
    public byte byteValue();                           // New in 1.1
    public double doubleValue();
    public boolean equals(Object obj);
    public float floatValue();
    public int hashCode();
    public int intValue();
    public boolean isInfinite();
    public boolean isNaN();
    public long longValue();
    public short shortValue();                         // New in 1.1
    public String toString();
}
```

Constants

MAX_VALUE

`public static final double MAX_VALUE = 1.79769313486231570e+308`

Description

The largest value that can be represented by a `double`.

MIN_VALUE

`public static final double MIN_VALUE = 4.94065645841246544e-324`

Description

The smallest value that can be represented by a `double`.

NaN

`public static final double NaN = 0.0 / 0.0`

Description

This variable represents the value not-a-number (NaN), which is a special value produced by `double` operations such as division of zero by zero. When NaN is one of the operands, most arithmetic operations return NaN as the result.

Most comparison operators (<, <=, ==, >=, >) return `false` when one of their arguments is NaN. The exception is `!=`, which returns `true` when one of its arguments is NaN.

NEGATIVE_INFINITY

`public static final double NEGATIVE_INFINITY = -1.0 / 0.0`

Description

> This variable represents the value negative infinity, which is produced when a `double` operation underflows or a negative `double` value is divided by zero. Negative infinity is by definition less than any other `double` value.

POSITIVE_INFINITY

`public static final double POSITIVE_INFINITY = 1.0 / 0.0`

Description

> This variable represents the value positive infinity, which is produced when a `double` operation overflows or a positive `double` value is divided by zero. Positive infinity is by definition greater than any other `double` value.

TYPE

`public static final Class TYPE`

Availability

> New as of JDK 1.1

Description

> The `Class` object that represents the type `double`. It is always true that `Double.TYPE` == `double.class`.

Constructors

Double

`public Double(double value)`

Parameters

`value`

The `double` value to be encapsulated by this object.

Description

Creates a `Double` object with the specified `double` value.

public Double(String s) throws NumberFormatException

Parameters

`s`

The string to be made into a `Double` object.

Throws

`NumberFormatException`

If the sequence of characters in the given `String` does not form a valid `double` literal.

Description

Constructs a `Double` object with the value specified by the given string. The string must contain a sequence of characters that forms a legal `double` literal.

Class Methods

doubleToLongBits

`public static native long doubleToLongBits(double value)`

Parameters

`value`

The `double` value to be converted.

Returns

The `long` value that contains the same sequence of bits as the representation of the given `double` value.

Description

This method returns the `long` value that contains the same sequence of bits as the representation of the given `double` value. The meaning of the bits in the result is defined by the IEEE 754 floating-point format: bit 63 is the sign bit, bits 62-52 are the exponent, and bits 51-0 are the mantissa.

An argument of `POSITIVE_INFINITY` produces the result `0x7ff0000000000000L`, an argument of `NEGATIVE_INFINITY` produces the result `0xfff0000000000000L`, and an argument of `NaN` produces the result `0x7ff8000000000000L`.

The value returned by this method can be converted back to the original `double` value by the `longBitsToDouble()` method.

isInfinite

```
static public boolean isInfinite(double v)
```

Parameters

`v`

The `double` value to be tested.

Returns

`true` if the specified value is equal to `POSITIVE_INFINITY` or `NEGATIVE_INFINITY`; otherwise `false`.

Description

This method determines whether or not the specified value is an infinity value.

isNaN

`public static boolean isNaN(double v)`

Parameters

`v`

The `double` value to be tested.

Returns

`true` if the specified value is equal to `NaN`; otherwise `false`.

Description

This method determines whether or not the specified value is NaN.

longBitsToDouble

`public static native double longBitsToDouble(long bits)`

Parameters

`bits`

The `long` value to be converted.

Returns

The `double` value whose representation is the same as the bits in the given `long` value.

Description

This method returns the `double` value whose representation is the same as the bits in the given `double` value. The meaning of the bits in the `long` value is defined by the IEEE 754 floating-point format: bit 63 is the sign bit, bits 62-52 are the exponent, and bits 51-0 are the mantissa. The argument `0x7f80000000000000L` produces the result `POSITIVE_INFINITY` and the argument `0xff80000000000000L` produces the result `NEGATIVE_INFINITY`. Arguments in the ranges `0x7ff0000000000001L` through `0x7fffffffffffffffL` and

`0xfff0000000000001L` through `0xffffffffffffffffL` all produce the result `NaN`.

Except for NaN values not normally used by Java, this method is the inverse of the `doubleToLongBits()` method.

toString

`public static String toString(double d)`

Parameters

`d`

The `double` value to be converted.

Returns

A string representation of the given value.

Description

This method returns a `String` object that contains a representation of the given `double` value.

The values `NaN`, `NEGATIVE_INFINITY`, `POSITIVE_INFINITY`, `-0.0`, and `+0.0` are represented by the strings `"NaN"`, `"--Infinity"`, `"Infinity"`, `"--0.0"`, and `"0.0"`, respectively.

For other values, the exact string representation depends on the value being converted. If the absolute value of `d` is greater than or equal to 10^-3 or less than or equal to 10^7, it is converted to a string with an optional minus sign (if the value is negative) followed by up to eight digits before the decimal point, a decimal point, and the necessary number of digits after the decimal point (but no trailing zero if there is more than one significant digit). There is always a minimum of one digit after the decimal point.

Otherwise, the value is converted to a string with an optional minus sign (if the value is negative), followed by a single digit, a decimal point, the necessary number of digits after the decimal point (but no trailing zero if there is more than one significant digit), and the letter `E` followed by a plus or a minus sign and a base 10 exponent of at least one digit. Again, there is always a minimum of one digit after the decimal point.

Note that the definition of this method has changed as of JDK 1.1. Prior to that release, the method provided a string representation that was equivalent to the `%g` format of the `printf` function in C.

valueOf

```
public static Double valueOf(String s) throws NumberFormatException
```

Parameters

`s`

The string to be made into a `Double` object.

Returns

The `Double` object constructed from the string.

Throws

`NumberFormatException`

If the sequence of characters in the given `String` does not form a valid `double` literal.

Description

Constructs a `Double` object with the value specified by the given string. The string must contain a sequence of characters that forms a legal `double` literal. This method ignores leading and trailing white space in the string.

Instance Methods

byteValue

```
public byte byteValue()
```

Availability

New as of JDK 1.1

Returns

> The value of this object as a `byte`.

Overrides

> `Number.byteValue()`

Description

> This method returns the truncated value of this object as a `byte`. More specifically, if the value of the object is `NaN`, the method returns 0. If the value is `POSITIVE_INFINITY`, or any other value that is too large to be represented by an `byte`, the method returns `Byte.MAX_VALUE`. If the value is `NEGATIVE_INFINITY`, or any other value that is too small to be represented by an `byte`, the method returns `Byte.MIN_VALUE`. Otherwise, the value is rounded toward zero and returned.

doubleValue

`public double doubleValue()`

Returns

> The value of this object as a `double`.

Overrides

> `Number.doubleValue()`

Description

> This method returns the value of this object as a `double`.

equals

`public boolean equals(Object obj)`

Parameters

> `obj`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `obj` is an instance of `Double` and it contains the same value as the object this method is associated with. More specifically, the method returns `true` if the `doubleToLongBits()` method returns the same result for the values of both objects.

This method produces a different result than the `==` operator when both values are `NaN`. In this case, the `==` operator produces `false`, while this method returns `true`. By the same token, the method also produces a different result when the two values are `+0.0` and `-0.0`. In this case, the `==` operator produces `true`, while this method returns `false`.

floatValue

`public float floatValue()`

Returns

The value of this object as a `float`.

Overrides

`Number.floatValue()`

Description

This method returns this object value as a `float`. Rounding may occur.

hashCode

`public int hashCode()`

Returns

A hashcode based on the `double` value of the object.

Overrides

`Object.hashCode()`

Description

This method returns a hashcode computed from the value of this object. More specifically, if `d` is the value of the object, and `bitValue` is defined as:

```
long bitValue = Double.doubleToLongBits(d)
```

then the hashcode returned by this method is computed as follows:

```
(int)(bitValue ^ (bitValue>>>32))
```

intValue

`public int intValue()`

Returns

The value of this object as an `int`.

Overrides

`Number.intValue()`

Description

This method returns the truncated value of this object as an `int`. More specifically, if the value of the object is `NaN`, the method returns 0. If the value is `POSITIVE_INFINITY`, or any other value that is too large to be represented by an `int`, the method returns `Integer.MAX_VALUE`. If the value is `NEGATIVE_INFINITY`, or any other value that is too small to be represented by an `int`, the method returns `Integer.MIN_VALUE`. Otherwise, the value is rounded toward zero and returned.

isInfinite

`public boolean isInfinite()`

Returns

> `true` if the value of this object is equal to `POSITIVE_INFINITY` or `NEGATIVE_INFINITY`; otherwise `false`.

Description

> This method determines whether or not the value of this object is an infinity value.

isNaN

`public boolean isNaN()`

Returns

> `true` if the value of this object is equal to `NaN`; otherwise `false`.

Description

> This method determines whether or not the value of this object is NaN.

longValue

`public long longValue()`

Returns

> The value of this object as a `long`.

Overrides

> `Number.longValue()`

Description

> This method returns the truncated value of this object as a `long`. More specifically, if the value of

the object is `NaN`, the method returns 0. If the value is `POSITIVE_INFINITY`, or any other value too large to be represented by a `long`, the method returns `Long.MAX_VALUE`. If the value is `NEGATIVE_INFINITY`, or any other value too small to be represented by a `long`, the method returns `Long.MIN_VALUE`. Otherwise, the value is rounded toward zero and returned.

shortValue

`public short shortValue()`

Availability

New as of JDK 1.1

Returns

The value of this object as a `short`.

Overrides

`Number.shortValue()`

Description

This method returns the truncated value of this object as a `short`. More specifically, if the value of the object is `NaN`, the method returns 0. If the value is `POSITIVE_INFINITY`, or any other value that is too large to be represented by an `short`, the method returns `Short.MAX_VALUE`. If the value is `NEGATIVE_INFINITY`, or any other value that is too small to be represented by an `short`, the method returns `Short.MIN_VALUE`. Otherwise, the value is rounded toward zero and returned.

toString

`public String toString()`

Returns

A string representation of the value of this object.

Overrides

```
Object.toString()
```

Description

This method returns a `String` object that contains a representation of the value of this object.

The values `NaN`, `NEGATIVE_INFINITY`, `POSITIVE_INFINITY`, `-0.0`, and `+0.0` are represented by the strings `"NaN"`, `"--Infinity"`, `"Infinity"`, `"--0.0"`, and `"0.0"`, respectively.

For other values, the exact string representation depends on the value being converted. If the absolute value of this object is greater than or equal to 10^-3 or less than or equal to 10^7, it is converted to a string with an optional minus sign (if the value is negative) followed by up to eight digits before the decimal point, a decimal point, and the necessary number of digits after the decimal point (but no trailing zero if there is more than one significant digit). There is always a minimum of one digit after the decimal point.

Otherwise, the value is converted to a string with an optional minus sign (if the value is negative), followed by a single digit, a decimal point, the necessary number of digits after the decimal point (but no trailing zero if there is more than one significant digit), and the letter `E` followed by a plus or a minus sign and a base 10 exponent of at least one digit. Again, there is always a minimum of one digit after the decimal point.

Note that the definition of this method has changed as of JDK 1.1. Prior to that release, the method provided a string representation that was equivalent to the `%g` format of the `printf` function in C.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`Class`, `Float`, `Number`, `NumberFormatException`, `String`

Error

Name

Error

Synopsis

Class Name:

`java.lang.Error`

Superclass:

`java.lang.Throwable`

Immediate Subclasses:

`java.awt.AWTError`, `java.lang.LinkageError`,

`java.lang.ThreadDeath`,

`java.lang.VirtualMachineError`

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The `Error` class is the superclass of all of the standard error classes that can be thrown in Java. The subclasses of `Error` are normally thrown by the class loader, the virtual machine, or other support code. Application-specific code should not normally throw any of the standard error classes.

An `Error` or one of its subclasses is typically thrown when an unpredictable run-time error, such as running out of memory, occurs. Because of the unpredictable nature of the events that cause errors to be thrown, a method does not have to declare the `Error` class or any of its subclasses in the `throws` clause of its method declaration.

A Java program should not try to handle the standard error classes. Most of these error classes represent nonrecoverable errors and as such, they cause the Java run-time system to print an error message and terminate program execution.

Class Summary

```
public class java.lang.Error extends java.lang.Throwable {
  // Constructors
  public Error();
  public Error(String s);
}
```

Constructors

Error

public Error()

Description

> This constructor creates an `Error` with no associated detail message.

public Error(String s)

Parameters

> `s`
>
> > The detail message.

Description

This constructor creates an `Error` with the specified detail message.

Inherited Methods

Method	**Inherited From**	**Method**	**Inherited From**
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`LinkageError`, `ThreadDeath`, `Throwable`, `VirtualMachineError`

PREVIOUS
Double

HOME
BOOK INDEX

NEXT
Exception

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Exception

Name

Exception

Synopsis

Class Name:

```
java.lang.Exception
```

Superclass:

```
java.lang.Throwable
```

Immediate Subclasses:

```
java.awt.AWTException,

java.awt.datatransfer.UnsupportedFlavorException,

java.io.IOException,

java.lang.ClassNotFoundException,

java.lang.CloneNotSupportedException,

java.lang.IllegalAccessException,

java.lang.InstantiationException,
```

`java.lang.InterruptedException`,

`java.lang.NoSuchMethodException`,

`java.lang.RuntimeException`,

`java.lang.reflect.InvocationTargetException`,

`java.text.FormatException`,

`java.util.TooManyListenersException`,

`java.util.zip.DataFormatException`

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The `Exception` class is the superclass of all of the standard exception classes that can be thrown in Java. The subclasses of `Exception` represent exceptional conditions a normal Java program may want to handle. Any explicitly thrown object in a Java program should be an instance of a subclass of `Exception`.

Many of the standard exceptions are also subclasses of `RuntimeException`. Run-time exceptions represent run-time conditions that can occur generally in any Java method, so a method is not required to declare that it throws any of the run-time exceptions. However, if a method can throw any of the other standard exceptions, it must declare them in its `throws` clause.

A Java program should try to handle all of the standard exception classes, since they represent routine abnormal conditions that should be anticipated and caught to prevent program termination.

Class Summary

```
public class java.lang.Exception extends java.lang.Throwable {
  // Constructors
  public Exception();
  public Exception(String s);
}
```

Constructors

Exception

public Exception()

Description

This constructor creates an `Exception` with no associated detail message.

public Exception(String s)

Parameters

`s`

The detail message.

Description

This constructor creates an `Exception` with the specified message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

ClassNotFoundException, CloneNotSupportedException, DataFormatException, FormatException, IllegalAccessException, InstantiationException, InvocationTargetException, InterruptedException, NoSuchMethodException, RuntimeException, Throwable, TooManyListenersException

PREVIOUS | HOME | NEXT
Error | BOOK INDEX | ExceptionInInitializerError

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

ExceptionInInitializerError

Name

ExceptionInInitializerError

Synopsis

Class Name:

`java.lang.ExceptionInInitializerError`

Superclass:

`java.lang.LinkageError`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

A `ExceptionInInitializerError` is thrown when an unexpected exception has been thrown in a static initializer.

Class Summary

```
public class java.lang.ExceptionInInitializer
             extends java.lang.LinkageError {
  // Constructors
  public ExceptionInInitializerError();
  public ExceptionInInitializerError(Throwable thrown);
  public ExceptionInInitializerError(String s);
  // Instance Methods
  public Throwable getException();
}
```

Constructors

ExceptionInInitializerError

public ExceptionInInitializerError()

Description

This constructor creates an `ExceptionInInitializerError` with no associated detail message.

public ExceptionInInitializerError(Throwable thrown)

Parameters

`thrown`

The exception that was thrown in the static initializer.

Description

This constructor creates an `ExceptionInInitializerError` that refers to the specified exception.

public ExceptionInInitializerError(String s)

Parameters

`s`

The detail message.

Description

This constructor creates an `ExceptionInInitializerError` with the specified detail message.

Instance Methods

getException

public Throwable getException()

Returns

The exception object that was thrown in the static initializer.

Description

This methods returns the exception that caused this error to be thrown.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Error`, `LinkageError`, `Throwable`

PREVIOUS HOME NEXT

Float

Name

Float

Synopsis

Class Name:

    java.lang.Float

Superclass:

    java.lang.Number

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The Float class provides an object wrapper for a float value. This is useful when you need to treat a float value as an object. For example, there are a number of utility methods that take a reference to an Object as one of their arguments. You cannot specify a float value for one of these arguments, but you can provide a reference to a Float object that encapsulates the float value. Furthermore, as of JDK 1.1, the Float class is necessary to support the Reflection API and class literals.

In Java, float values are represented using the IEEE 754 format. The Float class provides constants for the three special values that are mandated by this format: POSITIVE_INFINITY, NEGATIVE_INFINITY, and NaN (not-a-number).

The Float class also provides some utility methods, such as methods for determining whether a floatx value is an infinity value or NaN, for converting float values to other primitive types, and for converting a float to a String and vice versa.

Class Summary

```
public final class java.lang.Float extends java.lang.Number {
    // Constants
    public static final float MIN_VALUE;
    public static final float MAX_VALUE;
    public static final float NaN;
    public static final float NEGATIVE_INFINITY;
    public static final float POSITIVE_INFINITY;
    public final static Class TYPE;                    // New in 1.1
    // Constructors
    public Float(double value);
    public Float(float value);
    public Float(String s);
    // Class Methods
    public static native int floatToIntBits(float value);
    public static native float intBitsToFloat(int bits);
    public static boolean isInfinite(float v);
    public static boolean isNaN(float v);
    public static String toString(float f);
    public static Float valueOf(String s);
    // Instance Methods
    public byte byteValue();                           // New in 1.1
    public double doubleValue();
    public boolean equals(Object obj);
    public float floatValue();
    public int hashCode();
    public int intValue();
    public boolean isInfinite();
    public boolean isNaN();
    public long longValue();
    public short shortValue();                         // New in 1.1
    public String toString();
}
```

Constants

MAX_VALUE

`public static final float MAX_VALUE = 3.40282346638528860e+38f`

Description

> The largest value that can be represented by a `float`.

MIN_VALUE

`public static final float MIN_VALUE = 1.40129846432481707e-45f`

Description

> The smallest value that can be represented by a `float`.

NaN

`public static final float NaN = 0.0f / 0.0f`

Description

> This variable represents the value NaN, a special value produced by `float` operations such as division of zero by zero. When NaN is one of the operands, most arithmetic operations return NaN as the result. Most comparison operators (<, <=, ==, >=, >) return `false` when one of their arguments is NaN. The exception is `!=`, which returns `true` when one of its arguments is NaN.

NEGATIVE_INFINITY

`public static final float NEGATIVE_INFINITY = -1.0f / 0.0f`

Description

> This variable represents the value negative infinity, which is produced when a `float` operation underflows or a negative `float` value is divided by zero. Negative infinity is by definition less than any other `float` value.

POSITIVE_INFINITY

`public static final float POSITIVE_INFINITY = 1.0f / 0.0f`

Description

> This variable represents the value positive infinity, which is produced when a `float` operation overflows or a positive `float` value is divided by zero. Positive infinity is by definition greater than any other `float` value.

TYPE

`public static final Class TYPE`

Availability

> New as of JDK 1.1

Description

> The `Class` object that represents the type `float`. It is always true that `Float.TYPE` == `float.class`.

Constructors

Float

`public Float(double value)`

Parameters

`value`

The `double` value to be encapsulated by this object.

Description

Creates a `Float` object with the specified `double` value. The value is rounded to `float` precision.

public Float(float value)

Parameters

`value`

The `float` value to be encapsulated by this object.

Description

Creates a `Float` object with the specified `float` value.

public Float(String s) throws NumberFormatException

Parameters

`s`

The string to be made into a `Float` object.

Throws

`NumberFormatException`

If the sequence of characters in the given `String` does not form a valid `float` literal.

Description

Constructs a `Float` object with the value specified by the given string. The string must contain a sequence of characters that forms a legal `float` literal.

Class Methods

floatToIntBits

public static native int floatToIntBits(float value)

Parameters

> value
>
> > The float value to be converted.

Returns

> The int value that contains the same sequence of bits as the representation of the given float value.

Description

> This method returns the int value that contains the same sequence of bits as the representation of the given float value. The meaning of the bits in the result is defined by the IEEE 754 floating-point format: bit 31 is the sign bit, bits 30-23 are the exponent, and bits 22-0 are the mantissa. An argument of POSITIVE_INFINITY produces the result 0x7f800000, an argument of NEGATIVE_INFINITY produces the result 0xff800000, and an argument of NaN produces the result 0x7fc00000.
>
> The value returned by this method can be converted back to the original float value by the intBitsToFloat() method.

intBitsToFloat

public static native float intBitsToFloat(int bits)

Parameters

> bits
>
> > The int value to be converted.

Returns

The `float` value whose representation is the same as the bits in the given `int` value.

Description

This method returns the `float` value whose representation is the same as the bits in the given `int` value. The meaning of the bits in the `int` value is defined by the IEEE 754 floating-point format: bit 31 is the sign bit, bits 30-23 are the exponent, and bits 22-0 are the mantissa. The argument `0x7f800000` produces the result `POSITIVE_INFINITY`, and the argument `0xff800000` produces the result `NEGATIVE_INFINITY`. Arguments in the ranges `0x7f800001` through `0x7f8fffff` and `0xff800001` through `0xff8fffffL` all produce the result `NaN`.

Except for NaN values not normally used by Java, this method is the inverse of the `floatToIntBits()` method.

isInfinite

`public static boolean isInfinite(float v)`

Parameters

`v`

The `float` value to be tested.

Returns

`true` if the specified value is equal to `POSITIVE_INFINITY` or `NEGATIVE_INFINITY`; otherwise `false`.

Description

This method determines whether or not the specified value is an infinity value.

isNaN

`public static boolean isNaN(float v)`

Parameters

`v`

The `float` value to be tested.

Returns

`true` if the specified value is equal to `NaN`; otherwise `false`.

Description

This method determines whether or not the specified value is NaN.

toString

`public static String toString(float f)`

Parameters

`f`

The `float` value to be converted.

Returns

A string representation of the given value.

Description

This method returns a `String` object that contains a representation of the given `float` value.

The values `NaN`, `NEGATIVE_INFINITY`, `POSITIVE_INFINITY`, `-0.0`, and `+0.0` are represented by the strings `"NaN"`, `"--Infinity"`, `"Infinity"`, `"--0.0"`, and `"0.0"`, respectively.

For other values, the exact string representation depends on the value being converted. If the absolute value of `f` is greater than or equal to 10^-3 or less than or equal to 10^7, it is converted to a string with an optional minus sign (if the value is negative) followed by up to eight digits before the decimal point, a decimal point, and the necessary number of digits after the decimal point (but no trailing zero if there is more than one significant digit). There is always a minimum of one digit after the decimal point.

Otherwise, the value is converted to a string with an optional minus sign (if the value is negative), followed by a single digit, a decimal point, the necessary number of digits after the decimal point (but no trailing zero if there is more than one significant digit), and the letter `E` followed by a plus or a minus sign and a base 10 exponent of at least one digit. Again, there is always a minimum of one digit after the decimal point.

Note that the definition of this method has changed as of JDK 1.1. Prior to that release, the method provided a string representation that was equivalent to the `%g` format of the `printf` function in C.

valueOf

```
public static Float valueOf(String s) throws NumberFormatException
```

Parameters

`s`

The string to be made into a `Float` object.

Returns

The `Float` object constructed from the string.

Throws

`NumberFormatException`

If the sequence of characters in the given `String` does not form a valid `float` literal.

Description

Constructs a `Float` object with the value specified by the given string. The string must contain a sequence of characters that forms a legal `float` literal. This method ignores leading and trailing whitespace in the string.

Instance Methods

byteValue

`public byte byteValue()`

Availability

> New as of JDK 1.1

Returns

> The value of this object as a `byte`.

Overrides

> `Number.byteValue()`

Description

> This method returns the truncated value of this object as a `byte`. More specifically, if the value of the object is `NaN`, the method returns 0. If the value is `POSITIVE_INFINITY`, or any other value that is too large to be represented by an `byte`, the method returns `Byte.MAX_VALUE`. If the value is `NEGATIVE_INFINITY`, or any other value that is too small to be represented by an `byte`, the method returns `Byte.MIN_VALUE`. Otherwise, the value is rounded toward zero and returned.

doubleValue

`public double doubleValue()`

Returns

> The value of this object as a `double`.

Overrides

> `Number.doubleValue()`

Description

> This method returns the value of this object as a `double`.

equals

`public boolean equals(Object obj)`

Parameters

`obj`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `obj` is an instance of `Float` and it contains the same value as the object this method is associated with. More specifically, the method returns `true` if the `floatToIntBits()` method returns the same result for the values of both objects.

This method produces a different result than the `==` operator when both values are `NaN`. In this case, the `==` operator produces `false`, while this method returns `true`. By the same token, the method also produces a different result when the two values are `+0.0` and `-0.0`. In this case, the `==` operator produces `true`, while this method returns `false`.

floatValue

`public float floatValue()`

Returns

The value of this object as a `float`.

Overrides

`Number.floatValue()`

Description

This method returns the value of this object as a `float`.

hashCode

`public int hashCode()`

Returns

A hashcode based on the `float` value of the object.

Overrides

`Object.hashCode()`

Description

This method returns a hashcode computed from the value of this object. More specifically, if `f` is the value of the object, this method returns `Float.floatToIntBits(f)`.

intValue

`public int intValue()`

Returns

The value of this object as an `int`.

Overrides

`Number.intValue()`

Description

This method returns the truncated value of this object as an `int`. More specifically, if the value of the object is `NaN`, the method returns 0. If the value is `POSITIVE_INFINITY`, or any other value that is too large to be represented by an `int`, the method returns `Integer.MAX_VALUE`. If the value is `NEGATIVE_INFINITY`, or any other value that is too small to be represented by an `int`, the method returns `Integer.MIN_VALUE`. Otherwise, the value is rounded toward zero and returned.

isInfinite

`public boolean isInfinite(float v)`

Returns

`true` if the value of this object is equal to `POSITIVE_INFINITY` or `NEGATIVE_INFINITY`; otherwise `false`.

Description

This method determines whether or not the value of this object is an infinity value.

isNaN

`public boolean isNaN()`

Returns

`true` if the value of this object is equal to `NaN`; otherwise `false`.

Description

This method determines whether or not the value of this object is NaN.

longValue

`public long longValue()`

Returns

The value of this object as a `long`.

Overrides

`Number.longValue()`

Description

> This method returns the truncated value of this object as a `long`. More specifically, if the value of the object is `NaN`, the method returns 0. If the value is `POSITIVE_INFINITY`, or any other value that is too large to be represented by a `long`, the method returns `Long.MAX_VALUE`. If the value is `NEGATIVE_INFINITY`, or any other value that is too small to be represented by a `long`, the method returns `Long.MIN_VALUE`. Otherwise, the value is rounded toward zero and returned.

shortValue

`public short shortValue()`

Availability

> New as of JDK 1.1

Returns

> The value of this object as a `short`.

Overrides

> `Number.shortValue()`

Description

> This method returns the truncated value of this object as a `short`. More specifically, if the value of the object is `NaN`, the method returns 0. If the value is `POSITIVE_INFINITY`, or any other value that is too large to be represented by an `short`, the method returns `Short.MAX_VALUE`. If the value is `NEGATIVE_INFINITY`, or any other value that is too small to be represented by an `short`, the method returns `Short.MIN_VALUE`. Otherwise, the value is rounded toward zero and returned.

toString

`public String toString()`

Returns

> A string representation of the value of this object.

Overrides

`Object.toString()`

Description

This method returns a `String` object that contains a representation of the value of this object.

The values `NaN`, `NEGATIVE_INFINITY`, `POSITIVE_INFINITY`, `-0.0`, and `+0.0` are represented by the strings `"NaN"`, `"--Infinity"`, `"Infinity"`, `"--0.0"`, and `"0.0"`, respectively.

For other values, the exact string representation depends on the value being converted. If the absolute value of this object is greater than or equal to 10^-3 or less than or equal to 10^7, it is converted to a string with an optional minus sign (if the value is negative) followed by up to eight digits before the decimal point, a decimal point, and the necessary number of digits after the decimal point (but no trailing zero if there is more than one significant digit). There is always a minimum of one digit after the decimal point.

Otherwise, the value is converted to a string with an optional minus sign (if the value is negative), followed by a single digit, a decimal point, the necessary number of digits after the decimal point (but no trailing zero if there is more than one significant digit), and the letter `E` followed by a plus or a minus sign and a base 10 exponent of at least one digit. Again, there is always a minimum of one digit after the decimal point.

Note that the definition of this method has changed as of JDK 1.1. Prior to that release, the method provided a string representation that was equivalent to the `%g` format of the `printf` function in C.

Inherited Methods

Method	**Inherited From**	**Method**	**Inherited From**
`clone()`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`Class`, `Double`, `Number`, `NumberFormatException`, `String`

PREVIOUS | HOME | NEXT
ExceptionInInitializerError | BOOK INDEX | IllegalAccessError

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

IllegalAccessError

Name

IllegalAccessError

Synopsis

Class Name:

`java.lang.IllegalAccessError`

Superclass:

`java.lang.IncompatibleClassChangeError`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

An `IllegalAccessError` is thrown when a class attempts to access a field or call a method it does not have access to. Usually this error is caught by the compiler; this error can occur at run-time if the definition of a class changes after the class that references it was last compiled.

Class Summary

```
public class java.lang.IllegalAccessError
             extends java.lang.IncompatibleClassChangeError {
  // Constructors
  public IllegalAccessError();
  public IllegalAccessError(String s);
}
```

Constructors

IllegalAccessError

public IllegalAccessError()

Description

This constructor creates an `IllegalAccessError` with no associated detail message.

public IllegalAccessError(String s)

Parameters

`s`

The detail message.

Description

This constructor creates an `IllegalAccessError` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`

`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Error`, `IncompatibleClassChangeError`, `Throwable`

PREVIEWS HOME NEXT
Float BOOK INDEX IllegalAccessException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

IllegalAccessException

Name

IllegalAccessException

Synopsis

Class Name:

`java.lang.IllegalAccessException`

Superclass:

`java.lang.Exception`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

An `IllegalAccessException` is thrown when a program tries to dynamically load a class (i.e., uses the forName() method of the Class class, or the findSystemClass() or the loadClass() method of the ClassLoader class) and the currently executing method does not have access to the specified class because it is in another package and not public. This exception is also thrown when a program tries to create an instance of a class (i.e., uses the newInstance() method of the Class class) that does not have a zero-argument constructor accessible to the caller.

Class Summary

```
public class java.lang.IllegalAccessException extends java.lang.Exception {
  // Constructors
  public IllegalAccessException();
  public IllegalAccessException(String s);
}
```

Constructors

IllegalAccessException

public IllegalAccessException()

Description

This constructor creates an IllegalAccessException with no associated detail message.

public IllegalAccessException(String s)

Parameters

s

The detail message.

Description

This constructor creates an IllegalAccessException with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
clone()	Object	equals(Object)	Object

`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Class`, `ClassLoader`, `Exception`, `Throwable`

PREVIOUS HOME NEXT
IllegalAccessError BOOK INDEX IllegalArgumentException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

IllegalArgumentException

Name

IllegalArgumentException

Synopsis

Class Name:

`java.lang.IllegalArgumentException`

Superclass:

`java.lang.RuntimeException`

Immediate Subclasses:

`java.lang.IllegalThreadStateException`, `java.lang.NumberFormatException`

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

An `IllegalArgumentException` is thrown to indicate that an illegal argument has been passed to a method.

Class Summary

```
public class java.lang.IllegalArgumentException
             extends java.lang.RuntimeException {
  // Constructors
  public IllegalArgumentException();
  public IllegalArgumentException(String s);
}
```

Constructors

IllegalArgumentException

public IllegalArgumentException()

Description

This constructor creates an `IllegalArgumentException` with no associated detail message.

public IllegalArgumentException(String s)

Parameters

`s`

The detail message.

Description

This constructor creates an `IllegalArgumentException` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`

`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Exception`, `IllegalThreadStateException`, `NumberFormatException`, `RuntimeException`, `Throwable`

IllegalMonitorStateException

Name

IllegalMonitorStateException

Synopsis

Class Name:

`java.lang.IllegalMonitorStateException`

Superclass:

`java.lang.RuntimeException`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

An `IllegalMonitorStateException` is thrown when an object's `wait()`, `notify()`, or `notifyAll()` method is called from a thread that does not own the object's monitor.

Class Summary

```
public class java.lang.IllegalMonitorStateException
             extends java.lang.RuntimeException {
  // Constructors
  public IllegalMonitorStateException();
  public IllegalMonitorStateException(String s);
}
```

Constructors

IllegalMonitorStateException

public IllegalMonitorStateException()

Description

> This constructor creates an `IllegalMonitorStateException` with no associated detail message.

public IllegalMonitorStateException(String s)

Parameters

> `s`
>
> > The detail message.

Description

> This constructor creates an `IllegalMonitorStateException` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`

`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Exception`, `Object`, `RuntimeException`, `Throwable`

PREVIOUS	HOME	NEXT
IllegalArgumentException	BOOK INDEX	IllegalStateException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

IllegalStateException

Name

IllegalStateException

Synopsis

Class Name:

```
java.lang.IllegalStateException
```

Superclass:

```
java.lang.RuntimeException
```

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

An `IllegalStateException` is thrown to indicate that a method has been invoked when the run-time environment is in an inappropriate state for the requested operation.

Class Summary

```
public class java.lang.IllegalStateException
             extends java.lang.RuntimeException {
  // Constructors
  public IllegalStateException();
  public IllegalStateException(String s);
}
```

Constructors

IllegalStateException

public IllegalStateException()

Description

This constructor creates an `IllegalStateException` with no associated detail message.

public IllegalStateException(String s)

Parameters

`s`

The detail message.

Description

This constructor creates an `IllegalStateException` with the specified detail message.

Inherited Methods

Method	**Inherited From**	**Method**	**Inherited From**
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`

`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Exception`, `Object`, `RuntimeException`, `Throwable`

IllegalThreadStateException

Name

IllegalThreadStateException

Synopsis

Class Name:

`java.lang.IllegalThreadStateException`

Superclass:

`java.lang.IllegalArgumentException`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

An `IllegalThreadStateException` is thrown to indicate an attempt to perform an operation on a thread that is not legal for the thread's current state, such as attempting to resume a dead thread.

Class Summary

```
public class java.lang.IllegalThreadStateException
             extends java.lang.IllegalArgumentException {
  // Constructors
  public IllegalThreadStateException();
  public IllegalThreadStateException(String s);
}
```

Constructors

IllegalThreadStateException

public IllegalThreadStateException()

Description

This constructor creates an `IllegalThreadStateException` with no associated detail message.

public IllegalThreadStateException(String s)

Parameters

`s`

The detail message.

Description

This constructor creates an `IllegalThreadStateException` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`

`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Exception`, `IllegalArgumentException`, `RuntimeException`, `Thread`, `Throwable`

IncompatibleClassChangeError

Name

IncompatibleClassChangeError

Synopsis

Class Name:

`java.lang.IncompatibleClassChangeError`

Superclass:

`java.lang.LinkageError`

Immediate Subclasses:

`java.lang.AbstractMethodError`,

`java.lang.IllegalAccessError`,

`java.lang.InstantiationError`,

`java.lang.NoSuchFieldError`,

`java.lang.NoSuchMethodError`

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

An `IncompatibleClassChangeError` or one of its subclasses is thrown when a class refers to another class in an incompatible way. This situation occurs when the current definition of the referenced class is incompatible with the definition of the class that was found when the referring class was compiled. For example, say class `A` refers to a method in class `B`. Then, after class `A` is compiled, the method is removed from class `B`. When class `A` is loaded, the run-time system discovers that the method in class `B` no longer exists and throws an error.

Class Summary

```
public class java.lang.IncompatibleClassChangeError
             extends java.lang.LinkageError {
  // Constructors
  public IncompatibleClassChangeError();
  public IncompatibleClassChangeError(String s);
}
```

Constructors

IncompatibleClassChangeError

public IncompatibleClassChangeError()

Description

This constructor creates an `IncompatibleClassChangeError` with no associated detail message.

public IncompatibleClassChangeError(String s)

Parameters

`s`

The detail message.

Description

This constructor creates an `IncompatibleClassChangeError` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`AbstractMethodError`, `Error`, `IllegalAccessError`, `InstantiationError`, `LinkageError`, `NoSuchFieldError`, `NoSuchMethodError`, `Throwable`

PREVIOUS | HOME | NEXT
IllegalThreadStateException | BOOK INDEX | IndexOutOfBoundsException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

IndexOutOfBoundsException

Name

IndexOutOfBoundsException

Synopsis

Class Name:

`java.lang.IndexOutOfBoundsException`

Superclass:

`java.lang.RuntimeException`

Immediate Subclasses:

`java.lang.ArrayIndexOutOfBoundsException`,
`java.lang.StringIndexOutOfBoundsException`

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The appropriate subclass of `IndexOutOfBoundsException` is thrown when an array or string index is out of bounds.

Class Summary

```
public class java.lang.IndexOutOfBoundsException
             extends java.lang.RuntimeException {
  // Constructors
  public IndexOutOfBoundsException();
  public IndexOutOfBoundsException(String s);
}
```

Constructors

IndexOutOfBoundsException

public IndexOutOfBoundsException()

Description

This constructor creates an `IndexOutOfBoundsException` with no associated detail message.

public IndexOutOfBoundsException(String s)

Parameters

`s`

The detail message.

Description

This constructor creates an `IndexOutOfBoundsException` with the specified detail message.

Inherited Methods

Method	**Inherited From**	**Method**	**Inherited From**
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`

`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`ArrayIndexOutOfBoundsException`, `Exception`, `RuntimeException`, `StringIndexOutOfBoundsException`, `Throwable`

PREVIOUS HOME NEXT
IncompatibleClassChangeError BOOK INDEX Integer

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Integer

Name

Integer

Synopsis

Class Name:

> `java.lang.Integer`

Superclass:

> `java.lang.Number`

Immediate Subclasses:

> None

Interfaces Implemented:

> None

Availability:

> JDK 1.0 or later

Description

The `Integer` class provides an object wrapper for an `int` value. This is useful when you need to treat an `int` value as an object. For example, there are a number of utility methods that take a reference to an `Object` as one of their arguments. You cannot specify an `int` value for one of these arguments, but you can provide a reference to an `Integer` object that encapsulates the `int` value. Also, as of JDK 1.1, the `Integer` class is necessary to support the

Reflection API and class literals.

The `Integer` class also provides a number of utility methods for converting `int` values to other primitive types and for converting `int` values to strings and vice versa.

Class Summary

```
public final class java.lang.Integer extends java.lang.Number {
    // Constants
    public static final int MAX_VALUE;
    public static final int MIN_VALUE;
    public final static Class TYPE;                    // New in 1.1
    // Constructors
    public Integer(int value);
    public Integer(String s);
    // Class Methods
    public static Integer decode(String nm)            // New in 1.1
    public static Integer getInteger(String nm);
    public static Integer getInteger(String nm, int val);
    public static Integer getInteger(String nm, Integer val);
    public static int parseInt(String s);
    public static int parseInt(String s, int radix;
    public static String toBinaryString(long i);
    public static String toHexString(long i);
    public static String toOctalString(long i);
    public static String toString(int i);
    public static String toString(int i, int radix);
    public static Integer valueOf(String s);
    public static Integer valueOf(String s, int radix);
    // Instance Methods
    public byte byteValue();                           // New in 1.1
    public double doubleValue();
    public boolean equals(Object obj);
    public float floatValue();
    public int hashCode();
    public int intValue();
    public long longValue();
    public short shortValue();                         // New in 1.1
    public String toString();
}
```

Constants

MAX_VALUE

`public static final int MAX_VALUE = 0x7fffffff // 2147483647`

Description

The largest value that can be represented by an `int`.

MIN_VALUE

`public static final int MIN_VALUE = 0x80000000 // -2147483648`

Description

The smallest value that can be represented by an `int`.

TYPE

`public static final Class TYPE`

Availability

New as of JDK 1.1

Description

The `Class` object that represents the type `int`. It is always true that `Integer.TYPE` == `int.class`.

Constructors

Integer

`public Integer(int value)`

Parameters

`value`

The `int` value to be encapsulated by this object.

Description

Creates an `Integer` object with the specified `int` value.

`public Integer(String s) throws NumberFormatException`

Parameters

`s`

The string to be made into an `Integer` object.

Throws

`NumberFormatException`

If the sequence of characters in the given `String` does not form a valid `int` literal.

Description

Constructs an `Integer` object with the value specified by the given string. The string should consist of one or more digit characters. The digit characters can be preceded by a single `-' character. If the string contains any other characters, the constructor throws a `NumberFormatException`.

Class Methods

decode

`public static Integer decode(String nm)`

Availability

New as of JDK 1.1

Parameters

`nm`

A `String` representation of the value to be encapsulated by an `Integer` object. If the string begins with # or `0x`, it is a radix 16 representation of the value. If the string begins with `0`, it is a radix 8 representation of the value. Otherwise, it is assumed to be a radix 10 representation of the value.

Returns

An `Integer` object that encapsulates the given value.

Throws

`NumberFormatException`

If the `String` contains any nondigit characters other than a leading minus sign or the value represented by the `String` is less than `Integer.MIN_VALUE` or greater than `Integer.MAX_VALUE`.

Description

This method returns an `Integer` object that encapsulates the given value.

getInteger

`public static Integer getInteger(String nm)`

Parameters

`nm`

The name of a system property.

Returns

The value of the system property as an `Integer` object, or an `Integer` object with the value 0 if the named property does not exist or cannot be parsed.

Description

This method retrieves the value of the named system property and returns it as an `Integer` object. The method obtains the value of the system property as a `String` using `System.getProperty()`.

If the value of the property begins with `0x` or `#` and is not followed by a minus sign, the rest of the value is parsed as a hexadecimal integer. If the value begins with `0`, it's parsed as an octal integer; otherwise it's parsed as a decimal integer.

`public static Integer getInteger(String nm, int val)`

Parameters

`nm`

The name of a system property.

`val`

A default `int` value for the property.

Returns

The value of the system property as an `Integer` object, or an `Integer` object with the value `val` if the named property does not exist or cannot be parsed.

Description

This method retrieves the value of the named system property and returns it as an `Integer` object. The method obtains the value of the system property as a `String` using `System.getProperty()`.

If the value of the property begins with `0x` or `#` and is not followed by a minus sign, the rest of the value is parsed as a hexadecimal integer. If the value begins with `0`, it's parsed as an octal integer; otherwise it's parsed as a decimal integer.

`public static Integer getInteger(String nm, Integer val)`

Parameters

`nm`

The name of a system property.

`val`

A default `Integer` value for the property.

Returns

The value of the system property as an `Integer` object, or the `Integer` object `val` if the named property does not exist or cannot be parsed.

Description

This method retrieves the value of the named system property and returns it as an `Integer` object. The method obtains the value of the system property as a `String` using `System.getProperty()`.

If the value of the property begins with `0x` or `#` and is not followed by a minus sign, the rest of the value is parsed as a hexadecimal integer. If the value begins with `0`, it's parsed as an octal integer; otherwise it's as a decimal integer.

parseInt

`public static int parseInt(String s) throws NumberFormatException`

Parameters

`s`

The `String` to be converted to an `int` value.

Returns

The numeric value of the integer represented by the `String` object.

Throws

`NumberFormatException`

If the `String` does not contain a valid representation of an integer.

Description

This method returns the numeric value of the integer represented by the contents of the given `String` object. The `String` must contain only decimal digits, except that the first character may be a minus sign.

`public static int parseInt(String s, int radix) throws NumberFormatException`

Parameters

`s`

The `String` to be converted to an `int` value.

`radix`

The radix used in interpreting the characters in the `String` as digits. This value must be in the range `Character.MIN_RADIX` to `Character.MAX_RADIX`. If `radix` is in the range 2 through 10, only characters for which the `Character.isDigit()` method returns `true` are considered to be valid digits. If `radix` is in the range 11 through 36, characters in the ranges `A' through `Z' and `a' through `z' may be considered valid digits.

Returns

The numeric value of the integer represented by the `String` object in the specified radix.

Throws

`NumberFormatException`

If the `String` does not contain a valid representation of an integer, or `radix` is not in the appropriate range.

Description

This method returns the numeric value of the integer represented by the contents of the given `String` object in the specified radix. The `String` must contain only valid digits of the specified radix, except that the first character may be a minus sign. The digits are parsed in the specified radix to produce the numeric value.

toBinaryString

`public static String toBinaryString(int value)`

Parameters

`value`

The `int` value to be converted to a string.

Returns

A string that contains the binary representation of the given value.

Description

This method returns a `String` object that contains the representation of the given value as an unsigned binary number. To convert the given value to an unsigned quantity, the method simply uses the value as if it were not negative. In other words, if the given value is negative, the method adds 2^32 to it. Otherwise the value is used as it is.

The string returned by this method contains a sequence of one or more `0' and `1' characters. The method returns "0" if its argument is `0`. Otherwise, the string returned by this method begins with `1'.

toHexString

`public static String toHexString(int value)`

Parameters

`value`

The `int` value to be converted to a string.

Returns

A string that contains the hexadecimal representation of the given value.

Description

This method returns a `String` object that contains the representation of the given value as an unsigned hexadecimal number. To convert the given value to an unsigned quantity, the method simply uses the value as if it were not negative. In other words, if the given value is negative, the method adds 2^32 to it. Otherwise the value is used as it is.

The string returned by this method contains a sequence of one or more of the characters `0', `1', `2', `3', `4', `5', `6', `7', `8', `9', `a', `b', `c', `d', `e', and `f'. The method returns "0" if its argument is `0`. Otherwise, the string returned by this method does not begin with `0'.

To produce a string that contains upper- instead of lowercase letters, use the `String.toUpperCase()` method.

toOctalString

`public static String toOctalString(int value)`

Parameters

`value`

The `int` value to be converted to a string.

Returns

A string that contains the octal representation of the given value.

Description

This method returns a `String` object that contains a representation of the given value as an unsigned octal number. To convert the given value to an unsigned quantity, the method simply uses the value as if it were not negative. In other words, if the given value is negative, the method adds 2^32 to it. Otherwise the value is used as it is.

The string returned by this method contains a sequence of one or more of the characters `0', `1', `2', `3', `4', `5', `6', and `7'. The method returns "0" if its argument is `0`. Otherwise, the string returned by this method does not begin with `0'.

toString

`public static String toString(int i)`

Parameters

`i`

The `int` value to be converted to a string.

Returns

The string representation of the given value.

Description

This method returns a `String` object that contains the decimal representation of the given value.

This method returns a string that begins with `-' if the given value is negative. The rest of the string is a sequence of one or more of the characters `0', `1', `2', `3', `4', `5', `6', `7', `8', and `9'. This method returns "0" if its argument is 0. Otherwise, the string returned by this method does not begin with "0" or "-0".

public static String toString(int i, int radix)

Parameters

i

The int value to be converted to a string.

radix

The radix used in converting the value to a string. This value must be in the range Character.MIN_RADIX to Character.MAX_RADIX.

Returns

The string representation of the given value in the specified radix.

Description

This method returns a String object that contains the representation of the given value in the specified radix.

This method returns a string that begins with `-' if the given value is negative. The rest of the string is a sequence of one or more characters that represent the magnitude of the given value. The characters that can appear in the sequence are determined by the value of radix. If *N* is the value of radix, the first *N* characters on the following line can appear in the sequence:

```
0123456789abcdefghijklmnopqrstuvwxyz
```

The method does not verify that radix is in the proper range. If radix is less than Character.MIN_RADIX or greater than Character.MAX_RADIX, the value 10 is used instead of the given value.

This method returns "0" if its argument is 0. Otherwise, the string returned by this method does not begin with "0" or "-0".

valueOf

public static Integer valueOf(String s) throws NumberFormatException

Parameters

s

The string to be made into an `Integer` object.

Returns

The `Integer` object constructed from the string.

Throws

`NumberFormatException`

If the `String` does not contain a valid representation of an integer.

Description

Constructs an `Integer` object with the value specified by the given string. The string should consist of one or more digit characters. The digit characters can be preceded by a single `-' character. If the string contains any other characters, the method throws a `NumberFormatException`.

`public static Integer valueOf(String s, int radix) throws NumberFormatException`

Parameters

`s`

The string to be made into an `Integer` object.

`radix`

The radix used in converting the string to a value. This value must be in the range `Character.MIN_RADIX` to `Character.MAX_RADIX`.

Returns

The `Integer` object constructed from the string.

Throws

`NumberFormatException`

If the `String` does not contain a valid representation of an integer or `radix` is not in the appropriate range.

Description

Constructs an `Integer` object with the value specified by the given string in the specified radix. The string should consist of one or more digit characters or characters in the range `A' to `Z' or `a' to `z' that are considered digits in the given radix. The digit characters can be preceded by a single `-' character. If the string contains any other characters, the method throws a `NumberFormatException`.

Instance Methods

byteValue

`public byte byteValue()`

Availability

New as of JDK 1.1

Returns

The value of this object as a `byte`.

Overrides

`Number.byteValue()`

Description

This method returns the value of this object as a `byte`. The high order bits of the value are discarded.

doubleValue

`public double doubleValue()`

Returns

The value of this object as a `double`.

Overrides

`Number.doubleValue()`

Description

This method returns the value of this object as a `double`.

equals

`public boolean equals(Object obj)`

Parameters

`obj`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `obj` is an instance of `Integer` and it contains the same value as the object this method is associated with.

floatValue

`public float floatValue()`

Returns

The value of this object as a `float`.

Overrides

`Number.floatValue()`

Description

This method returns the value of this object as a `float`. Rounding may occur.

hashCode

`public int hashCode()`

Returns

A hashcode based on the `int` value of the object.

Overrides

`Object.hashCode()`

Description

This method returns a hashcode computed from the value of this object.

intValue

`public int intValue()`

Returns

The value of this object as an `int`.

Overrides

`Number.intValue()`

Description

This method returns the value of this object as an `int`.

longValue

`public long longValue()`

Returns

The value of this object as a `long`.

Overrides

`Number.longValue()`

Description

This method returns the value of this object as a `long`.

shortValue

`public short shortValue()`

Availability

New as of JDK 1.1

Returns

The value of this object as a `short`.

Overrides

`Number.shortValue()`

Description

This method returns the value of this object as a `short`. The high order bits of the value are discarded.

toString

`public String toString()`

Returns

The string representation of the value of this object.

Overrides

`Object.toString()`

Description

This method returns a `String` object that contains the decimal representation of the value of this object.

This method returns a string that begins with `-' if the value is negative. The rest of the string is a sequence of one or more of the characters `0', `1', `2', `3', `4', `5', `6', `7', `8', and `9'. This method returns "0" if the value of the object is `0`. Otherwise, the string returned by this method does not begin with "0" or "-0".

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`Character`, `Class`, `Long`, `Number`, `NumberFormatException`, `String`, `System`

InstantiationError

Name

InstantiationError

Synopsis

Class Name:

`java.lang.InstantiationError`

Superclass:

`java.lang.IncompatibleClassChangeError`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

An `InstantiationError` is thrown in response to an attempt to instantiate an `abstract` class or interface.

Usually this error is caught by the compiler; this error can occur at run-time if the definition of a class is changed after the class that references it was last compiled.

Class Summary

```
public class java.lang.InstantiationError
             extends java.lang.IncompatibleClassChangeError {
  // Constructors
  public InstantiationError();
  public InstantiationError(String s);
}
```

Constructors

InstantiationError

public InstantiationError()

Description

This constructor creates an `InstantiationError` with no associated detail message.

public InstantiationError(String s)

Parameters

`s`

The detail message.

Description

This constructor creates an `InstantiationError` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`

`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Error`, `IncompatibleClassChangeError`, `Throwable`

PREVIOUS Integer | HOME BOOK INDEX | NEXT InstantiationException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

InstantiationException

Name

InstantiationException

Synopsis

Class Name:

`java.lang.InstantiationException`

Superclass:

`java.lang.Exception`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

An `InstantiationException` is thrown in response to an attempt to instantiate an abstract class or an interface using the `newInstance()` method of the `Class` class.

Class Summary

```
public class java.lang.InstantiationException extends java.lang.Exception {
  // Constructors
  public InstantiationException();
  public InstantiationException(String s);
}
```

Constructors

InstantiationException

public InstantiationException()

Description

This constructor creates an `InstantiationException` with no associated detail message.

public InstantiationException(String s)

Parameters

`s`

The detail message.

Description

This constructor creates an `InstantiationException` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`

`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Class`, `Exception`, `Throwable`

InternalError

Name

InternalError

Synopsis

Class Name:

`java.lang.InternalError`

Superclass:

`java.lang.VirtualMachineError`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

An `InternalError` is thrown to signal an internal error within the virtual machine.

Class Summary

```
public class java.lang.InternalError extends java.lang.VirtualMachineError {
  // Constructors
  public InternalError();
  public InternalError(String s);
}
```

Constructors

InternalError

public InternalError()

Description

This constructor creates an `InternalError` with no associated detail message.

public InternalError(String s)

Parameters

`s`

The detail message.

Description

This constructor creates an `InternalError` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`

`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Error`, `Throwable`, `VirtualMachineError`

PREVIOUS | HOME | NEXT
InstantiationException | BOOK INDEX | InterruptedException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

InterruptedException

Name

InterruptedException

Synopsis

Class Name:

`java.lang.InterruptedException`

Superclass:

`java.lang.Exception`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

An `InterruptedException` is thrown to signal that a thread that is sleeping, waiting, or otherwise paused, has been interrupted by another thread.

Class Summary

```
public class java.lang.InterruptedException extends java.lang.Exception {
  // Constructors
  public InterruptedException();
  public InterruptedException(String s);
}
```

Constructors

InterruptedException

public InterruptedException()

Description

> This constructor creates an `InterruptedException` with no associated detail message.

public InterruptedException(String s)

Parameters

> `s`
>
> > The detail message.

Description

> This constructor creates an `InterruptedException` with the specified detail message.

Inherited Methods

Method	**Inherited From**	**Method**	**Inherited From**
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`

`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Exception`, `Thread`, `Throwable`

PREVIOUS InternalError | HOME BOOK INDEX | NEXT LinkageError

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

LinkageError

Name

LinkageError

Synopsis

Class Name:

`java.lang.LinkageError`

Superclass:

`java.lang.Error`

Immediate Subclasses:

`java.lang.ClassCircularityError`,

`java.lang.ClassFormatError`,

`java.lang.ExceptionInInitializerError`,

`java.lang.IncompatibleClassChangeError`,

`java.lang.NoClassDefFoundError`,

`java.lang.UnsatisfiedLinkError`,

`java.lang.VerifyError`

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The appropriate subclass of `LinkageError` is thrown when there is a problem resolving a reference to a class. Reasons for this may include a difficulty in finding the definition of the class or an incompatibility between the current definition and the expected definition of the class.

Class Summary

```
public class java.lang.LinkageError extends java.lang.Error {
  // Constructors
  public LinkageError();
  public LinkageError(String s);
}
```

Constructors

LinkageError

public LinkageError()

Description

This constructor creates a `LinkageError` with no associated detail message.

public LinkageError(String s)

Parameters

`s`

The detail message.

Description

This constructor create a `LinkageError` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`ClassCircularityError`, `ClassFormatError`, `Error`, `ExceptionInInitializerError`, `IncompatibleClassChangeError`, `NoClassDefFoundError`, `Throwable`, `UnsatisfiedLinkError`, `VerifyError`

PREVIOUS InterruptedException | HOME | BOOK INDEX | NEXT Long

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Long

Name

Long

Synopsis

Class Name:

`java.lang.Long`

Superclass:

`java.lang.Number`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The `Long` class provides an object wrapper for a `long` value. This is useful when you need to treat a `long` value as an object. For example, there are a number of utility methods that take a reference to an `Object` as one of their arguments. You cannot specify a `long` value for one of these arguments, but you can provide a reference to a `Long` object that encapsulates the `long` value. Furthermore, as of JDK 1.1, the `Long` class is necessary to support the Reflection API and class literals.

The `Long` class also provides a number of utility methods for converting `long` values to other primitive types and for converting `long` values to strings and vice versa.

Class Summary

```
public final class java.lang.Long extends java.lang.Number {
    // Constants
    public static final long MIN_VALUE;
    public static final long MAX_VALUE;
    public final static Class TYPE;                   // New in 1.1
    // Constructors
    public Long(long value);
    public Long(String s);
    // Class Methods
    public static Long getLong(String nm);
    public static Long getLong(String nm, long val);
    public static Long getLong(String nm, Long val);
    public static long parseLong(String s);
    public static long parseLong(String s, int radix);
    public static String toBinaryString(long i);
    public static String toHexString(long i);
    public static String toOctalString(long i);
    public static String toString(long i);
    public static String toString(long i, int radix);
    public static Long valueOf(String s);
    public static Long valueOf(String s, int radix);
    // Instance Methods
    public byte byteValue();                          // New in 1.1
    public double doubleValue();
    public boolean equals(Object obj);
    public float floatValue();
    public int hashCode();
    public int intValue();
    public long longValue();
    public short shortValue();                        // New in 1.1
    public String toString();
}
```

Constants

MAX_VALUE

`public static final long MAX_VALUE = 0x7fffffffffffffffL`

Description

The largest value that can be represented by a `long`.

MIN_VALUE

`public static final long MIN_VALUE = 0x8000000000000000L`

Description

The smallest value that can be represented by a `long`.

TYPE

`public static final Class TYPE`

Availability

New as of JDK 1.1

Description

The `Class` object that represents the type `long`. It is always true that `Long.TYPE` == `long.class`.

Constructors

Long

`public Long(long value)`

Parameters

`value`

The `long` value to be encapsulated by this object.

Description

Creates a `Long` object with the specified `long` value.

`public Long(String s) throws NumberFormatException`

Parameters

`s`

The string to be made into a `Long` object.

Throws

`NumberFormatException`

If the sequence of characters in the given `String` does not form a valid `long` literal.

Description

Constructs a `Long` object with the value specified by the given string. The string should consist of one or more digit characters. The digit characters can be preceded by a single `-' character. If the string contains any other characters, the constructor throws a `NumberFormatException`.

Class Methods

getLong

`public static Integer getLong(String nm)`

Parameters

`nm`

The name of a system property.

Returns

The value of the system property as a `Long` object or a `Long` object with the value 0 if the named property does not exist or cannot be parsed.

Description

This method retrieves the value of the named system property and returns it as a `Long` object. The method obtains the value of the system property as a `String` using `System.getProperty()`.

If the value of the property begins with `0x` or `#` and is not followed by a minus sign, the rest of the value is parsed as a hexadecimal integer. If the value begins with `0`, it's parsed as an octal integer; otherwise it's parsed as a decimal integer.

`public static Long getLong(String nm, long val)`

Parameters

`nm`

The name of a system property.

`val`

A default value for the property.

Returns

The value of the system property as a `Long` object or a `Long` object with the value `val` if the named property does not exist or cannot be parsed.

Description

This method retrieves the value of the named system property and returns it as a `Long` object. The method obtains the value of the system property as a `String` using `System.getProperty()`.

If the value of the property begins with `0x` or # and is not followed by a minus sign, the rest of the value is parsed as a hexadecimal integer. If the value begins with `0`, it's parsed as an octal integer; otherwise it's parsed as a decimal integer.

`public static Long getLong(String nm, Long val)`

Parameters

`nm`

The name of a system property.

`val`

A default value for the property.

Returns

The value of the system property as a `Long` object, or the `Long` object `val` if the named property does not exist or cannot be parsed.

Description

This method retrieves the value of the named system property and returns it as a `Long` object. The method obtains the value of the system property as a `String` using `System.getProperty()`.

If the value of the property begins with `0x` or # and is not followed by a minus sign, the rest of the value is parsed as a hexadecimal integer. If the value begins with `0`, it's parsed as an octal integer; otherwise it's parsed as a decimal integer.

parseLong

`public static long parseLong(String s) throws NumberFormatException`

Parameters

`s`

The `String` to be converted to a `long` value.

Returns

The numeric value of the `long` represented by the `String` object.

Throws

`NumberFormatException`

If the `String` does not contain a valid representation of a `long` value.

Description

This method returns the numeric value of the `long` represented by the contents of the given `String` object. The `String` must contain only decimal digits, except that the first character may be a minus sign.

`public static long parseLong(String s, int radix) throws NumberFormatException`

Parameters

`s`

The `String` to be converted to a `long` value.

`radix`

The radix used in interpreting the characters in the `String` as digits. It must be in the range `Character.MIN_RADIX` to `Character.MAX_RADIX`. If `radix` is in the range 2 through 10, only characters for which the `Character.isDigit()` method returns `true` are considered valid digits. If `radix` is in the range 11 through 36, characters in the ranges `A' through `Z' and `a' through `z' may be considered valid digits.

Returns

The numeric value of the `long` represented by the `String` object in the specified radix.

Throws

`NumberFormatException`

If the `String` does not contain a valid representation of a `long` or `radix` is not in the appropriate range.

Description

This method returns the numeric value of the `long` represented by the contents of the given `String` object in the specified radix. The `String` must contain only valid digits of the specified radix, except that the first character may be a minus sign. The digits are parsed in the specified radix to produce the numeric value.

toBinaryString

`public static String toBinaryString(long value)`

Parameters

`value`

The `long` value to be converted to a string.

Returns

A string that contains the binary representation of the given value.

Description

This method returns a `String` object that contains the representation of the given value as an unsigned binary number. To convert the given value to an unsigned quantity, the method simply uses the value as if it were not negative. In other words, if the given value is negative, the method adds 2^64 to it. Otherwise the value is used as it is.

The string returned by this method contains a sequence of one or more `0' and `1' characters. The method returns "0" if its argument is `0`. Otherwise, the string returned by this method begins with `1'.

toHexString

`public static String toHexString(long value)`

Parameters

`value`

The `long` value to be converted to a string.

Returns

A string that contains the hexadecimal representation of the given value.

Description

This method returns a `String` object that contains the representation of the given value as an unsigned hexadecimal number. To convert the given value to an unsigned quantity, the method simply uses the value as if it were not negative. In other words, if the given value is negative, the method adds 2^64 to it. Otherwise the value is used as it is.

The string returned by this method contains a sequence of one or more of the characters `0', `1', `2', `3', `4', `5', `6', `7', `8', `9', `a', `b', `c', `d', `e', and `f'. The method returns "0" if its argument is `0`. Otherwise, the string returned by this method does not begin with `0'.

To produce a string that contains upper- instead of lowercase letters, use the `String.toUpperCase()` method.

toOctalString

`public static String toOctalString(long value)`

Parameters

`value`

The `long` value to be converted to a string.

Returns

A string that contains the octal representation of the given value.

Description

This method returns a `String` object that contains a representation of the given value as an unsigned octal number. To convert the given value to an unsigned quantity, the method simply uses the value as if it were not negative. In other words, if the given value is negative, the method adds 2^64 to it. Otherwise the value is used as it is.

The string returned by this method contains a sequence of one or more of the characters `0', `1', `2', `3', `4', `5', `6', and `7'. The method returns "0" if its argument is `0`. Otherwise, the string returned by this method does not begin with `0'.

toString

`public static String toString(long i)`

Parameters

`i`

The `long` value to be converted to a string.

Returns

The string representation of the given value.

Description

This method returns a `String` object that contains the decimal representation of the given value.

This method returns a string that begins with `` `-' `` if the given value is negative. The rest of the string is a sequence of one or more of the characters `0', `1', `2', `3', `4', `5', `6', `7', `8', and `9'. This method returns "0" if its argument is `0`. Otherwise, the string returned by this method does not begin with "0" or "`-0`".

`public static String toString(long i, int radix)`

Parameters

`i`

The `long` value to be converted to a string.

`radix`

The radix used in converting the value to a string. This value must be in the range `Character.MIN_RADIX` to `Character.MAX_RADIX`.

Returns

The string representation of the given value in the specified radix.

Description

This method returns a `String` object that contains the representation of the given value in the specified radix.

This method returns a string that begins with`` `-' `` if the given value is negative. The rest of the string is a sequence of one or more characters that represent the magnitude of the given value. The characters that can appear in the sequence are determined by the value of `radix`. If *N* is the value of `radix`, the first *N* characters on the following line can appear in the sequence:

`0123456789abcdefghijklmnopqrstuvwxyz`

The method does not verify that `radix` is in the proper range. If `radix` is less than `Character.MIN_RADIX` or greater than `Character.MAX_RADIX`, the value 10 is used instead of the given value.

This method returns "0" if its argument is `0`. Otherwise, the string returned by this method does not begin with "0" or "`-`0".

valueOf

`public static Long valueOf(String s) throws NumberFormatException`

Parameters

`s`

The string to be made into a `Long` object.

Returns

The `Long` object constructed from the string.

Throws

`NumberFormatException`

If the `String` does not contain a valid representation of a `long`.

Description

Constructs a `Long` object with the value specified by the given string. The string should consist of one or more digit characters. The digit characters can be preceded by a single `-` character. If the string contains any other characters, the method throws a `NumberFormatException`.

`public static Long valueOf(String s, int radix) throws NumberFormatException`

Parameters

`s`

The string to be made into a `Long` object.

`radix`

The radix used in converting the string to a value. This value must be in the range `Character.MIN_RADIX` to `Character.MAX_RADIX`.

Returns

The `Long` object constructed from the string.

Throws

`NumberFormatException`

If the `String` does not contain a valid representation of a `long`.

Description

Constructs a `Long` object with the value specified by the given string in the specified radix. The string should consist of one or more digit characters or characters in the range `A' to `Z' or `a' to `z' that are considered digits in the given radix. The digit characters can be preceded by a single `-' character. If the string contains any other characters, the method throws a `NumberFormatException`.

The method does not verify that `radix` is in the proper range. If `radix` is less than `Character.MIN_RADIX` or greater than `Character.MAX_RADIX`, the value 10 is used instead of the given value.

Instance Methods

byteValue

`public byte byteValue()`

Availability

New as of JDK 1.1

Returns

The value of this object as a `byte`.

Overrides

`Number.byteValue()`

Description

This method returns the value of this object as a `byte`. The high order bits of the value are discarded.

doubleValue

`public double doubleValue()`

Returns

The value of this object as a `double`.

Overrides

`Number.doubleValue()`

Description

This method returns the value of this object as a `double`. Rounding may occur.

equals

`public boolean equals(Object obj)`

Parameters

`obj`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `obj` is an instance of `Long` and it contains the same value as the object this method is associated with.

floatValue

`public float floatValue()`

Returns

The value of this object as a `float`.

Overrides

`Number.floatValue()`

Description

This method returns the value of this object as a `float`. Rounding may occur.

hashCode

`public int hashCode()`

Returns

A hashcode based on the `long` value of the object.

Overrides

`Object.hashCode()`

Description

This method returns a hashcode computed from the value of this object. More specifically, the result is the exclusive OR of the two halves of the `long` value represented by the object. If `value` is the value of the object, the method returns a result equivalent to the following expression:

```
(int)(value^(value>>>32))
```

intValue

`public int intValue()`

Returns

The value of this object as an `int`.

Overrides

`Number.intValue()`

Description

This method returns the value of this object as an `int`. The high-order bits of the value are discarded.

longValue

`public long longValue()`

Returns

The value of this object as a `long`.

Overrides

`Number.longValue()`

Description

This method returns the value of this object as a `long`.

shortValue

`public short shortValue()`

Availability

New as of JDK 1.1

Returns

The value of this object as a `short`.

Overrides

`Number.shortValue()`

Description

This method returns the value of this object as a `short`. The high-order bits of the value are discarded.

toString

`public String toString()`

Returns

The string representation of the value of this object.

Overrides

`Object.toString()`

Description

This method returns a `String` object that contains the decimal representation of the value of this object.

This method returns a string that begins with `-' if the value is negative. The rest of the string is a sequence of one or more of the characters `0', `1', `2', `3', `4', `5', `6', `7', `8', and `9'. This method returns "0" if the value of the object is 0. Otherwise, the string returned by this method does not begin with "0" or "-0".

Inherited Methods

Method	Inherited From	Method	Inherited From
clone()	Object	finalize()	Object
getClass()	Object	notify()	Object
notifyAll()	Object	wait()	Object
wait(long)	Object	wait(long, int)	Object

See Also

Character, Class, Integer, Number, NumberFormatException, String, System

PREVIOUS HOME NEXT
LinkageError BOOK INDEX Math

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

PREVIOUS

NEXT

Math

Name

Math

Synopsis

Class Name:

> `java.lang.Math`

Superclass:

> `java.lang.Object`

Immediate Subclasses:

> None

Interfaces Implemented:

> None

Availability:

> JDK 1.0 or later

Description

The `Math` class contains constants for the mathematical values pi and *e*. The class also defines methods that compute various mathematical functions, such as trigonometric and exponential functions. All of these constants and methods are `static`. In other words, it is not necessary to create an instance of the `Math` class in order to use its constants and methods. In fact, the `Math` class does not define any `public` constructors, so it cannot be instantiated.

To ensure that the methods in this class return consistent results under different implementations of Java, all of the methods use the algorithms from the well-known Freely-Distributable Math Library package, *fdlibm*. This package is part of the network library *netlib*. The library can be obtained through the URL *http://netlib.att.com*. The algorithms used in this class are from the version of *fdlibm* dated January 4, 1995. *fdlibm* provides more than one definition for some functions. In those cases, the "IEEE 754 core function" version is used.

Class Summary

```
public final class java.lang.Math extends java.lang.Object {
    // Constants
    public static final double E;
    public static final double PI;
    // Class Methods
    public static int abs(int a);
    public static long abs(long a);
    public static float abs(float a);
    public static double abs(double a);
    public static native double acos(double a);
    public static native double asin(double a);
    public static native double atan(double a);
    public static native double atan2(double a, double b);
    public static native double ceil(double a);
    public static native double cos(double a);
    public static native double exp(double a);
    public static native double floor(double a);
    public static native double IEEEremainder(double f1, double f2);
    public static native double log(double a);
    public static int max(int a, int b);
    public static long max(long a, long b);
    public static float max(float a, float b);
    public static double max(double a, double b);
    public static int min(int a, int b);
    public static long min(long a, long b);
    public static float min(float a, float b);
    public static double min(double a, double b);
    public static native double pow(double a, double b);
    public static synchronized double random();
    public static native double rint(double a);
    public static int round(float a);
    public static long round(double a);
    public static native double sin(double a);
    public static native double sqrt(double a);
    public static native double tan(double a);
}
```

Constants

E

`public static final double E = 2.7182818284590452354`

Description

> The value of this constant is *e*, the base for natural logarithms.

PI

`public static final double PI = 3.14159265358979323846`

Description

> The value for this constant is pi.

Class Methods

abs

`public static double abs(double a)`

Parameters

`a`

A `double` value.

Returns

The absolute value of its argument.

Description

This method returns the absolute value of its argument.

If the argument to this method is negative or positive zero, the method should return positive zero. If the argument is positive or negative infinity, the method returns positive infinity. If the argument is NaN, the method returns NaN.

`public static float abs(float a)`

Parameters

`a`

A `float` value.

Returns

The absolute value of its argument.

Description

This method returns the absolute value of its argument.

If the argument to this method is negative or positive zero, the method should return positive zero. If the argument is positive or negative infinity, the method returns positive infinity. If the argument is NaN, the method returns NaN.

`public static int abs(int a)`

Parameters

`a`

An `int` value.

Returns

The absolute value of its argument.

Description

This method returns the absolute value of its argument.

If the argument is `Integer.MIN_VALUE`, the method actually returns `Integer.MIN_VALUE` because the true absolute value of `Integer.MIN_VALUE` is one greater than the largest positive value that can be represented by an `int`.

`public static long abs(long a)`

Parameters

`a`

A `long` value.

Returns

The absolute value of its argument.

Description

This method returns the absolute value of its argument.

If the argument is `Long.MIN_VALUE`, the method actually returns `Long.MIN_VALUE` because the true absolute value of `Long.MIN_VALUE` is one greater than the largest positive value represented by a `long`.

acos

`public static native double acos(double a)`

Parameters

`a`

A `double` value greater than or equal to `-1.0` and less than or equal to `1.0`.

Returns

The arc cosine measured in radians; the result is greater than or equal to `0.0` and less than or equal to pi.

Description

This method returns the arc cosine of the given value.

If the value is NaN or its absolute value is greater than `1.0`, the method returns NaN.

asin

`public static native double asin(double a)`

Parameters

`a`

A `double` value greater than or equal to `-1.0` and less than or equal to `1.0`.

Returns

The arc sine measured in radians; the result is greater than or equal to -pi/2 and less than or equal to pi/2.

Description

This method returns the arc sine of the given value.

If the value is NaN or its absolute value is greater than `1.0`, the method returns NaN. If the value is positive zero, the method returns positive zero. If the value is negative zero, the method returns negative zero.

atan

`public static native double atan(double a)`

Parameters

`a`

A `double` value greater than or equal to `-1.0` and less than or equal to `1.0`.

Returns

The arc tangent measured in radians; the result is greater than or equal to -pi/2 and less than or equal to pi/2.

Description

This method returns the principle value of the arc tangent of the given value.

If the value is NaN, the method returns NaN. If the value is positive zero, the method returns positive zero. If the value is negative zero, the method returns negative zero.

atan2

`public static native double atan2(double a, double b)`

Parameters

`a`

A `double` value.

`b`

A `double` value.

Returns

The theta component of the polar coordinate (*r*,theta) that corresponds to the cartesian coordinate (`a`,`b`); the result is measured in radians and is greater than or equal to -pi and less than or equal to pi.

Description

This method returns the theta component of the polar coordinate (*r*,theta) that corresponds to the cartesian coordinate (`a`,`b`). It computes theta as the principle value of the arc tangent of `b`/`a`, using the signs of both arguments to determine the quadrant (and sign) of the return value.

If either argument is NaN, the method returns NaN.

If the first argument is positive zero and the second argument is positive, then the method returns positive zero. If the first argument is positive zero and the second argument is negative, then the method returns the `double` value closest to pi.

If the first argument is negative zero and the second argument is positive, the method returns negative zero. If the first argument is negative zero and the second argument is negative, the method returns the `double` value closest to -pi.

If the first argument is positive and finite and the second argument is positive infinity, the method returns positive zero. If the first argument is positive and finite and the second argument is negative infinity, the method returns the `double` value closest to pi.

If the first argument is negative and finite and the second argument is positive infinity, the method returns negative zero. If the first argument is negative and finite and the second argument is negative infinity, the method returns the `double` value closest to -pi.

If the first argument is positive and the second argument is positive zero or negative zero, the method returns the `double` value closest to pi/2. If the first argument is negative and the second argument is positive or negative zero, the method returns the `double` value closest to -pi/2.

If the first argument is positive infinity and the second argument is finite, the method returns the `double` value closest to pi/2. If the first argument is negative infinity and the second argument is finite, the method returns the `double` value closest to -pi/2.

If both arguments are positive infinity, the method returns the `double` value closest to pi/4. If the first argument is positive infinity and the second argument is negative infinity, the method returns the `double` value closest to 3pi/4. If the first argument is negative infinity and the second argument is positive infinity, the method returns the `double` value closest to -pi/4. If both arguments are negative infinity, the method returns the `double` value closest to -3pi/4.

ceil

```
public static native double ceil(double a)
```

Parameters

`a`

A `double` value.

Returns

The smallest integer greater than or equal to the given value.

Description

This method performs the ceiling operation. It returns the smallest integer that is greater than or equal to its argument.

If the argument is NaN, an infinity value, or a zero value, the method returns that same value. If the argument is less than zero but greater than `-1.0`, the method returns negative zero.

cos

`public static native double cos(double a)`

Parameters

`a`

A `double` value that's an angle measured in radians.

Returns

The cosine of the given angle.

Description

This method returns the cosine of the given angle measured in radians.

If the angle is NaN or an infinity value, the method returns NaN.

exp

`public static native double exp(double a)`

Parameters

`a`

A `double` value.

Returns

`e^a`

Description

This method returns the exponential function of `a`. In other words, *e* is raised to the value specified by the parameter `a`, where *e* is the base of the natural logarithms.

If the value is NaN, the method returns NaN. If the value is positive infinity, the method returns positive infinity. If the value is negative infinity, the method returns positive zero.

floor

`public static native double floor(double a)`

Parameters

`a`

A `double` value.

Returns

The greatest integer less than or equal to the given value.

Description

This method performs the floor operation. It returns the largest integer that is less than or equal to its argument.

If the argument is NaN, an infinity value, or a zero value, the method returns that same value.

IEEEremainder

`public static native double IEEEremainder(double a, double b)`

Parameters

`a`

A `double` value.

`b`

A `double` value.

Returns

The remainder of `a` divided by `b` as defined by the IEEE 754 standard.

Description

This method returns the remainder of `a` divided by `b` as defined by the IEEE 754 standard. This operation involves first determining the mathematical quotient of `a`/`b` rounded to the nearest integer. If the quotient is equally close to two integers, it is rounded to the even integer. The method then returns `a-(b x` *`Q`*`)`, where *`Q`* is the rounded quotient.

If either argument is NaN, the method returns NaN. If the first argument is positive or negative infinity and the second argument is positive or negative zero, the method also returns NaN. If the first argument is a finite value and the second argument is positive or negative infinity, the method returns its first argument.

log

`public static native double log(double a)`

Parameters

`a`

A `double` value that is greater than `0.0`.

Returns

The natural logarithm of `a`.

Description

This method returns the natural logarithm (base *e*) of its argument.

In particular, if the argument is positive infinity, the method returns positive infinity. If the argument is positive or negative zero, the method returns negative infinity. If the argument is less than zero, the method returns NaN. If the argument is NaN, the method returns NaN.

max

`public static double max(double a, double b)`

Parameters

`a`

A `double` value.

`b`

A `double` value.

Returns

The greater of `a` and `b`.

Description

This method returns the greater of its two arguments. In other words, it returns the one that is closer to positive infinity.

If one argument is positive zero and the other is negative zero, the method returns positive zero. If either argument is NaN, the method returns NaN.

`public static float max(float a, float b)`

Parameters

`a`

A `float` value.

`b`

A `float` value.

Returns

The greater of `a` and `b`.

Description

This method returns the greater of its two arguments. In other words, it returns the one that is closer to positive infinity.

If one argument is positive zero and the other is negative zero, the method returns positive zero. If either argument is NaN, the method returns NaN.

`public static int max(int a, int b)`

Parameters

`a`

An `int` value.

`b`

An `int` value.

Returns

The greater of `a` and `b`.

Description

This method returns the greater of its two arguments. In other words, it returns the one that is closer to `Integer.MAX_VALUE`.

`public static long max(long a, long b)`

Parameters

`a`

A `long` value.

`b`

A `long` value.

Returns

The greater of `a` and `b`.

Description

This method returns the greater of its two arguments. In other words, it returns the one that is closer to `Long.MAX_VALUE`.

min

`public static double min(double a, double b)`

Parameters

`a`

A `double` value.

`b`

A `double` value.

Returns

The lesser of `a` and `b`.

Description

This method returns the lesser of its two arguments. In other words, it returns the one that is closer to negative infinity.

If one argument is positive zero and the other is negative zero, the method returns negative zero. If either argument is NaN, the method returns NaN.

`public static float min(float a, float b)`

Parameters

`a`

A `float` value.

`b`

A `float` value.

Returns

The lesser of `a` and `b`.

Description

This method returns the lesser of its two arguments. In other words, it returns the one that is closer to negative infinity.

If one argument is positive zero and the other is negative zero, the method returns negative zero. If either argument is NaN, the method returns NaN.

`public static int min(int a, int b)`

Parameters

`a`

An `int` value.

`b`

An `int` value.

Returns

The lesser of `a` and `b`.

Description

This method returns the lesser of its two arguments. In other words, it returns the one that is closer to `Integer.MIN_VALUE`.

`public static long min(long a, long b)`

Parameters

`a`

A `long` value.

`b`

A `long` value.

Returns

The lesser of `a` and `b`.

Description

This method returns the lesser of its two arguments. In other words, it returns the one that is closer to `Long.MIN_VALUE`.

pow

`public static native double pow(double a, double b)`

Parameters

`a`

A `double` value.

`b`

A `double` value.

Returns

`a^b`

Description

This method computes the value of raising `a` to the power of `b`.

If the second argument is positive or negative zero, the method returns `1.0`. If the second argument is `1.0`, the method returns its first argument. If the second argument is NaN, the method returns NaN. If the first argument is NaN and the second argument is nonzero, the method returns NaN.

If the first argument is positive zero and the second argument is greater than zero, the method returns positive zero. If the first argument is positive zero and the second argument is less than zero, the method returns positive infinity.

If the first argument is positive infinity and the second argument is less than zero, the method returns positive zero. If the first argument is positive infinity and the second argument is greater than zero, the method returns positive infinity.

If the absolute value of the first argument is greater than `1` and the second argument is positive infinity, the method returns positive infinity. If the absolute value of the first argument is greater than `1` and the second argument is negative infinity, the method returns positive zero. If the absolute value of the first argument is less than `1` and the second argument is negative infinity, the method returns positive infinity. If the absolute value of the first argument is less than `1` and the second argument is positive infinity, the method returns positive zero. If the absolute value of the first argument is `1` and the second argument is positive or negative infinity, the method returns NaN.

If the first argument is negative zero and the second argument is greater than zero but not a finite odd integer, the method returns positive zero. If the first argument is negative zero and the second argument is a positive finite odd integer, the method returns negative zero. If the first argument is negative zero and the second argument is less than zero but not a finite odd integer, the method returns positive infinity. If the first argument is negative zero and the second argument is a negative finite odd integer, the method returns negative infinity.

If the first argument is negative infinity and the second argument is less than zero but not a finite odd integer, the method returns positive zero. If the first argument is negative infinity and the second argument is a negative finite odd integer, the method returns negative zero. If the first argument is negative infinity and the second argument is greater than zero but not a finite odd integer, the method returns positive infinity. If the first argument is negative infinity and the second argument is a positive finite odd integer, the method returns negative infinity.

If the first argument is less than zero and the second argument is a finite even integer, the method returns the result of the absolute value of the first argument raised to the power of the second argument. If the first argument is less than zero and the second argument is a finite odd integer, the method returns the negative of the result of the absolute value of the first argument raised to the power of the second argument. If the first argument is finite and less than zero and the second argument is finite and not an integer, the method returns NaN.

If both arguments are integer values, the method returns the first argument raised to the power of the second argument.

random

`public static synchronized double random()`

Returns

A random number between `0.0` and `1.0`.

Description

This method returns a random number greater than or equal to `0.0` and less than `1.0`. The implementation of this method uses the `java.util.Random` class. You may prefer to use the `Random` class directly, in order to gain more control over the distribution, type, and repeatability of the random numbers you are generating.

rint

`public static native double rint(double a)`

Parameters

`a`

A `double` value.

Returns

The value of its argument rounded to the nearest integer.

Description

This method returns its argument rounded to the nearest integer; the result is returned as a `double` value. If the argument is equidistant from two integers (e.g., `1.5`), the method returns the even integer.

If the argument is an infinity value, a zero value, or NaN, the method returns that same value.

round

```
public static long round(double a)
```

Parameters

`a`

A `double` value.

Returns

The value of its argument rounded to the nearest `long`.

Description

This method returns its `double` argument rounded to the nearest integral value and converted to a `long`. If the argument is equidistant from two integers, the method returns the greater of the two integers.

If the argument is positive infinity or any other value greater than `Long.MAX_VALUE`, the method returns `Long.MAX_VALUE`. If the argument is negative infinity or any other value less than `Long.MIN_VALUE`, the method returns `Long.MIN_VALUE`. If the argument is NaN, the method returns `0`.

`public static int round(float a)`

Parameters

`a`

A `float` value.

Returns

The value of its argument rounded to the nearest `int`.

Description

This method returns its `float` argument rounded to the nearest integral value and converted to an `int`. If the argument is equidistant from two integers, the method returns the greater of the two integers.

If the argument is positive infinity or any other value greater than the `Integer.MAX_VALUE`, the method returns `Integer.MAX_VALUE`. If the argument is negative infinity or any other value less than `Integer.MIN_VALUE`, the method returns `Integer.MIN_VALUE`. If the argument is NaN, the method returns `0`.

sin

`public static native double sin(double a)`

Parameters

`a`

A `double` value that's an angle measured in radians.

Returns

The sine of the given angle.

Description

This method returns the sine of the given angle measured in radians.

If the angle is NaN or an infinity value, the method returns NaN. If the angle is positive zero, the method returns positive zero. If the angle is negative zero, the method returns negative zero.

sqrt

`public static native double sqrt(double a)`

Parameters

`a`

A `double` value.

Returns

The square root of its argument.

Description

This method returns the square root of its argument.

If the argument is negative or NaN, the method returns NaN. If the argument is positive infinity, the method returns positive infinity. If the argument is positive or negative zero, the method returns that same value.

tan

`public static native double tan(double a)`

Parameters

`a`

A `double` value that is an angle measured in radians.

Returns

The tangent of the given angle.

Description

This method returns the tangent of the given angle measured in radians.

If the angle is NaN or an infinity value, the method returns NaN. If the angle is positive zero, the method returns positive zero. If the angle is negative zero, the method returns negative zero.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Double`, `Float`, `Integer`, `Long`, `Object`

PREVIOUS Long | HOME | BOOK INDEX | NEXT NegativeArraySizeException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

NegativeArraySizeException

Name

NegativeArraySizeException

Synopsis

Class Name:

    java.lang.NegativeArraySizeException

Superclass:

    java.lang.RuntimeException

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

A `NegativeArraySizeException` is thrown in response to an attempt to create an array with a negative size.

Class Summary

```
public class java.lang.NegativeArraySizeException
             extends java.lang.RuntimeException {
  // Constructors
  public NegativeArraySizeException();
  public NegativeArraySizeException(String s);
}
```

Constructors

NegativeArraySizeException

public NegativeArraySizeException()

Description

This constructor creates a `NegativeArraySizeException` with no associated detail message.

public NegativeArraySizeException(String s)

Parameters

`s`

The detail message.

Description

This constructor creates a `NegativeArraySizeException` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`

`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Exception`, `RuntimeException`, `Throwable`

PREVIOUS | HOME | NEXT
Math | BOOK INDEX | NoClassDefFoundError

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

NoClassDefFoundError

Name

NoClassDefFoundError

Synopsis

Class Name:

`java.lang.NoClassDefFoundError`

Superclass:

`java.lang.LinkageError`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

A `NoClassDefFoundError` is thrown when the definition of a class cannot be found.

Class Summary

```
public class java.lang.NoClassDefFoundError extends java.lang.LinkageError {
  // Constructors
  public NoClassDefFoundError();
  public NoClassDefFoundError(String s);
}
```

Constructors

NoClassDefFoundError

public NoClassDefFoundError()

Description

This constructor creates a `NoClassDefFoundError` with no associated detail message.

public NoClassDefFoundError(String s)

Parameters

`s`

The detail message.

Description

This constructor creates a `NoClassDefFoundError` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`

`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Error`, `LinkageError`, `Throwable`

PREVIOUS | HOME | NEXT

NegativeArraySizeException | BOOK INDEX | NoSuchFieldError

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

NoSuchFieldError

Name

NoSuchFieldError

Synopsis

Class Name:

```
java.lang.NoSuchFieldError
```

Superclass:

```
java.lang.IncompatibleClassChangeError
```

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

A `NoSuchFieldError` is thrown in response to an attempt to reference an instance or class variable that is not defined in the current definition of a class. Usually this error is caught by the compiler; it can occur at run-time if the definition of a class is changed after the class that references it was last compiled.

Class Summary

```
public class java.lang.NoSuchFieldError
             extends java.lang.IncompatibleClassChangeError {
  // Constructors
  public NoSuchFieldError();
  public NoSuchFieldError(String s);
}
```

Constructors

NoSuchFieldError

public NoSuchFieldError()

Description

This constructor creates a `NoSuchFieldError` with no associated detail message.

public NoSuchFieldError(String s)

Parameters

`s`

The detail message.

Description

This constructor creates a `NoSuchFieldError` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`

`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Error`, `IncompatibleClassChangeError`, `Throwable`

PREVIOUS HOME NEXT
NoClassDefFoundError BOOK INDEX NoSuchFieldException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

NoSuchFieldException

Name

NoSuchFieldException

Synopsis

Class Name:

    java.lang.NoSuchFieldException

Superclass:

    java.lang.Exception

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

A `NoSuchFieldException` is thrown when a specified variable cannot be found.

Class Summary

```
public class java.lang.NoSuchFieldException extends java.lang.Exception {
  // Constructors
  public NoSuchFieldException();
  public NoSuchFieldException(String s);
}
```

Constructors

NoSuchFieldException

public NoSuchFieldException()

Description

This constructor creates a `NoSuchFieldException` with no associated detail message.

public NoSuchFieldException(String s)

Parameters

`s`

The detail message.

Description

This constructor creates a `NoSuchFieldException` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`

`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Exception`, `Throwable`

PREVIOUS
NoSuchFieldError

HOME
BOOK INDEX

NEXT
NoSuchMethodError

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

NoSuchMethodError

Name

NoSuchMethodError

Synopsis

Class Name:

    java.lang.NoSuchMethodError

Superclass:

    java.lang.IncompatibleClassChangeError

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

A `NoSuchMethodError` is thrown in response to an attempt to reference a method that is not defined in the current definition of a class. Usually this error is caught by the compiler; it can occur at run-time if the definition of a class is changed after the class that references it was last compiled.

Class Summary

```
public class java.lang.NoSuchMethodError
             extends java.lang.IncompatibleClassChangeError {
  // Constructors
  public NoSuchMethodError();
  public NoSuchMethodError(String s);
}
```

Constructors

NoSuchMethodError

public NoSuchMethodError()

Description

This constructor creates a `NoSuchMethodError` with no associated detail message.

public NoSuchMethodError(String s)

Parameters

`s`

The detail message.

Description

This constructor creates a `NoSuchMethodError` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`

`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Error`, `IncompatibleClassChangeError`, `Throwable`

PREVIOUS | HOME | NEXT
NoSuchFieldException | BOOK INDEX | NoSuchMethodException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

NoSuchMethodException

Name

NoSuchMethodException

Synopsis

Class Name:

`java.lang.NoSuchMethodException`

Superclass:

`java.lang.Exception`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

A `NoSuchMethodException` is thrown when a specified method cannot be found.

Class Summary

```
public class java.lang.NoSuchMethodException extends java.lang.Exception {
  // Constructors
  public NoSuchMethodException();
  public NoSuchMethodException(String s);
}
```

Constructors

NoSuchMethodException

public NoSuchMethodException()

Description

This constructor creates a `NoSuchMethodException` with no associated detail message.

public NoSuchMethodException(String s)

Parameters

`s`

The detail message.

Description

This constructor creates a `NoSuchMethodException` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`

`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Exception`, `Throwable`

PREVIOUS

NEXT

NullPointerException

Name

NullPointerException

Synopsis

Class Name:

java.lang.NullPointerException

Superclass:

java.lang.RuntimeException

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

A `NullPointerException` is thrown when there is an attempt to access an object through a `null` object reference. This can occur when there is an attempt to access an instance variable or call a method through a `null` object or when there is an attempt to subscript an array with a `null` object.

Class Summary

```
public class java.lang.NullPointerException
             extends java.lang.RuntimeException {
  // Constructors
  public NullPointerException();
  public NullPointerException(String s);
}
```

Constructors

NullPointerException

public NullPointerException()

Description

This constructor creates a `NullPointerException` with no associated detail message.

public NullPointerException(String s)

Parameters

`s`

The detail message.

Description

This constructor creates a `NullPointerException` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`

getClass()	Object	getLocalizedMessage()	Throwable
getMessage()	Throwable	hashCode()	Object
notify()	Object	notifyAll()	Object
printStackTrace()	Throwable	printStackTrace(PrintStream)	Throwable
printStackTrace(PrintWriter)	Throwable	toString()	Object
wait()	Object	wait(long)	Object
wait(long, int)	Object		

See Also

Exception, RuntimeException, Throwable

Number

Name

Number

Synopsis

Class Name:

`java.lang.Number`

Superclass:

`java.lang.Object`

Immediate Subclasses:

`java.lang.Byte`, `java.lang.Double`,

`java.lang.Float`, `java.lang.Integer`,

`java.lang.Long`, `java.lang.Short`,

`java.math.BigDecimal`,

`java.math.BigInteger`

Interfaces Implemented:

java.io.Serializable

Availability:

JDK 1.0 or later

Description

The Number class is an abstract class that serves as the superclass for all of the classes that provide object wrappers for primitive numeric values: byte, short, int, long, float, and double. Wrapping a primitive value is useful when you need to treat such a value as an object. For example, there are a number of utility methods that take a reference to an Object as one of their arguments. You cannot specify a primitive value for one of these arguments, but you can provide a reference to an object that encapsulates the primitive value. Furthermore, as of JDK 1.1, these wrapper classes are necessary to support the Reflection API and class literals.

The Number class defines six methods that must be implemented by its subclasses: byteValue(), shortValue(), intValue(), longValue(), floatValue(), and doubleValue(). This means that a Number object can be fetched as an byte, short, int, long, float, or double value, without regard for its actual class.

Class Summary

```
public abstract class java.lang.Number extends java.lang.Number
                                       implements java.io.Serializable {
    // Instance Methods
    public abstract byte byteValue();                  // New in 1.1
    public abstract double doubleValue();
    public abstract float floatValue();
    public abstract int intValue();
    public abstract long longValue();
    public abstract short shortValue();                // New in 1.1
}
```

Instance Methods

byteValue

public abstract byte byteValue()

Availability

New as of JDK 1.1

Returns

The value of this object as a `byte`.

Description

This method returns the value of this object as a `byte`. If the data type of the value is not `byte`, rounding may occur.

doubleValue

`public abstract double doubleValue()`

Returns

The value of this object as a `double`.

Description

This method returns the value of this object as a `double`. If the data type of the value is not `double`, rounding may occur.

floatValue

`public abstract float floatValue()`

Returns

The value of this object as a `float`.

Description

This method returns the value of this object as a `float`. If the data type of the value is not `float`, rounding may occur.

intValue

`public abstract int intValue()`

Returns

The value of this object as an `int`.

Description

This method returns the value of this object as an `int`. If the type of value is not an `int`, rounding may occur.

longValue

`public abstract long longValue()`

Returns

The value of this object as a `long`.

Description

This method returns the value of this object as a `long`. If the type of value is not a `long`, rounding may occur.

shortValue

`public abstract short shortValue()`

Availability

New as of JDK 1.1

Returns

The value of this object as a `short`.

Description

This method returns the value of this object as a `short`. If the type of value is not a `short`, rounding may occur.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Byte`, `Double`, `Float`, `Integer`, `Long`, `Object`, `Short`

PREVIOUS HOME NEXT

NullPointerException BOOK INDEX NumberFormatException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

NumberFormatException

Name

NumberFormatException

Synopsis

Class Name:

`java.lang.NumberFormatException`

Superclass:

`java.lang.IllegalArgumentException`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

A `NumberFormatException` is thrown to indicate that an attempt to parse numeric information in a string has failed.

Class Summary

```
public class java.lang.NumberFormatException
             extends java.lang.IllegalArgumentException {
  // Constructors
  public NumberFormatException();
  public NumberFormatException(String s);
}
```

Constructors

NumberFormatException

public NumberFormatException()

Description

This constructor creates a `NumberFormatException` with no associated detail message.

public NumberFormatException(String s)

Parameters

`s`

The detail message.

Description

This constructor creates a `NumberFormatException` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`

`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Exception`, `IllegalArgumentException`, `RuntimeException`, `Throwable`

PREVIOUS HOME NEXT
Number BOOK INDEX Object

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Object

Name

Object

Synopsis

Class Name:

`java.lang.Object`

Superclass:

None

Immediate Subclasses:

Too many to be listed here

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The `Object` class is the ultimate superclass of all other classes in Java. Because every other class is a subclass of `Object`, all of the methods accessible from `Object` are inherited by every other class. In other words, all objects in Java, including arrays, have access to implementations of the methods in `Object`.

The methods of `Object` provide some basic object functionality. The `equals()` method compares two objects for equality, while the `hashCode()` method returns a hashcode for an object. The `getClass()` method returns the `Class` object associated with an object. The `wait()`, `notify()`, and `notifyAll()` methods support thread synchronization for an object. The `toString()` method provides a string representation of an object.

Some of these methods should be overridden by subclasses of `Object`. For example, every class should provide its own implementation of the `toString()` method, so that it can provide an appropriate string representation.

Although it is possible to create an instance of the `Object` class, this is rarely done because it is more useful to create specialized objects. However, it is often useful to declare a variable that contains a reference to an `Object` because such a variable can contain a reference to an object of any other class.

Class Summary

```
public class java.lang.Object {
    // Constructors
    public Object();
    // Public Instance Methods
    public boolean equals(Object obj);
    public final native Class getClass();
    public native int hashCode();
    public final native void notify();
    public final native void notifyAll();
    public String toString();
    public final native void wait();
    public final native void wait(long millis);
    public final native void wait(long millis, int nanos);
    // Protected Instance Methods
    protected native Object clone();
    protected void finalize() throws Throwable;
}
```

Constructors

Object

`public Object()`

Description

> Creates an instance of the `Object` class.

Public Instance Methods

equals

`public boolean equals(Object obj)`

Parameters

`obj`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Description

The `equals()` method of `Object` returns `true` if the `obj` parameter refers to the same object as the object this method is associated with. This is equivalent to using the == operator to compare two objects.

Some classes, such as `String`, override the `equals()` method to provide a comparison based on the contents of the two objects, rather than on the strict equality of the references. Any subclass can override the `equals()` method to implement an appropriate comparison, as long as the overriding method satisfies the following rules for an equivalence relation:

- The method is *reflexive* : given a reference `x`, `x.equals(x)` returns `true`.
- The method is *symmetric* : given references `x` and `y`, `x.equals(y)` returns `true` if and only if `y.equals(x)` returns `true`.
- The method is *transitive* : given references `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` returns `true`.
- The method is *consistent* : given references `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided that no information contained by the objects referenced by `x` or `y` changes.
- A comparison with `null` returns `false`: given a reference `x` that is non-`null`, `x.equals(null)` returns `false`.

getClass

`public final native Class getClass()`

Returns

A reference to the `Class` object that describes the class of this object.

Description

The `getClass()` method of `Object` returns the `Class` object that describes the class of this object. This method is `final`, so it cannot be overridden by subclasses.

hashCode

`public native int hashCode()`

Returns

A relatively unique value that should be the same for all objects that are considered equal.

Description

The `hashCode()` method of `Object` calculates a hashcode value for this object. The method returns an integer value that should be relatively unique to the object. If the `equals()` method for the object bases its result on the contents of the object, the `hashcode()` method should also base its result on the contents. The `hashCode()` method is provided for the benefit of hashtables, which store and retrieve elements using key values called *hashcodes*. The internal placement of a particular piece of data is determined by its hashcode; hashtables are designed to use hashcodes to provide efficient retrieval.

The `java.util.Hashtable` class provides an implementation of a hashtable that stores values of type `Object`. Each object is stored in the hashtable based on the hash code of its key object. It is important that each object have the most unique hash code possible. If two objects have the same hash code but they are not equal (as determined by `equals()`), a `Hashtable` that stores these two objects may need to spend additional time searching when it is trying to retrieve objects. The implementation of `hashCode()` in `Object` tries to make sure that every object has a distinct hash code by basing its result on the internal representation of a reference to the object.

Some classes, such as `String`, override the `hashCode()` method to produce values based on the contents of individual objects, instead of the objects themselves. In other words, two `String` objects that contain the exact same strings have the same hash code. If `String` did not override the `hashCode()` method inherited from `Object`, these two `String` objects would have different hash code values and it would be impossible to use `String` objects as keys for hashtables.

Any subclass can override the `hashCode()` method to implement an appropriate way of producing hash code values, as long as the overriding method satisfies the following rules:

- If the `hashCode()` method is called on the same object more than once during the execution of a Java application, it must consistently return the same integer value. The integer does not, however, need to be consistent between Java applications, or from one execution of an application to another execution of the same application.
- If two objects compare as equal according to their `equals()` methods, calls to the `hashCode()` methods for the objects must produce the same result.
- If two objects compare as not equal according to their `equals()` methods, calls to the `hashCode()` methods for the two objects are not required to produce distinct results. However, implementations of `hashCode()` that produce distinct results for unequal objects may improve the performance of hashtables.

In general, if a subclass overrides the `equals()` method of `Object`, it should also override the `hashCode()` method.

notify

`public final native void notify()`

Throws

`IllegalMonitorStateException`

If the method is called from a thread that does not hold this object's lock.

Description

The `notify()` method wakes up a thread that is waiting to return from a call to this object's `wait()` method. The awakened thread can resume executing as soon as it regains this object's lock. If more than one thread is waiting, the `notify()` method arbitrarily awakens just one of the threads.

The `notify()` method can be called only by a thread that is the current owner of this object's lock. A thread holds the lock on this object while it is executing a `synchronized` instance method of the object or executing the body of a `synchronized` statement that synchronizes on the object. A thread can also hold the lock for a `Class` object if it is executing a `synchronized static` method of that class.

This method is `final`, so it cannot be overridden by subclasses.

notifyAll

`public final native void notifyAll()`

Throws

`IllegalMonitorStateException`

If the method is called from a thread that does not hold this object's lock.

Description

The `notifyAll()` method wakes up all the threads that are waiting to return from a call to this object's `wait()` method. Each awakened thread can resume executing as soon as it regains this object's lock.

The `notifyAll()` method can be called only by a thread that is the current owner of this object's lock. A thread holds the lock on this object while it is executing a `synchronized` instance method of the object or executing the body of a `synchronized` statement that synchronizes on the object. A thread can also hold the lock for a `Class` object if it is executing a `synchronized static` method of that class.

This method is `final`, so it cannot be overridden by subclasses.

toString

`public String toString()`

Returns

The string representation of this object.

Description

The `toString()` method of `Object` returns a generic string representation of this object. The method returns a `String` that consists of the object's class name, an "at" sign, and the unsigned hexadecimal representation of the value returned by the object's `hashCode()` method.

Many classes override the `toString()` method to provide a string representation that is specific to that type of object. Any subclass can override the `toString()` method; the overriding method should simply return a `String` that represents the contents of the object with which it is associated.

wait

`public final native void wait() throws InterruptedException`

Throws

`IllegalMonitorStateException`

If the method is called from a thread that does not hold this object's lock.

`InterruptedException`

If another thread interrupted this thread.

Description

The `wait()` method causes a thread to wait until it is notified by another thread to stop waiting. When `wait()` is called, the thread releases its lock on this object and waits until another thread notifies it to wake up through a call to either `notify()` or `notifyAll()`. After the thread is awakened, it has to regain the lock on this object before it can resume executing.

The `wait()` method can be called only by a thread that is the current owner of this object's lock. A thread holds the lock on this object while it is executing a `synchronized` instance method of the object or executing the body of a `synchronized` statement that synchronizes on the object. A thread can also hold the lock for a `Class` object if it is executing a `synchronized static` method of that class.

This method is `final`, so it cannot be overridden by subclasses.

`public final native void wait(long timeout) throws InterruptedException`

Parameters

`timeout`

The maximum number of milliseconds to wait.

Throws

`IllegalMonitorStateException`

If the method is called from a thread that does not hold this object's lock.

`InterruptedException`

If another thread interrupted this thread.

Description

The `wait()` method causes a thread to wait until it is notified by another thread to stop waiting or until the specified amount of time has elapsed, whichever comes first. When `wait()` is called, the thread releases its lock on this object and waits until another thread notifies it to wake up through a call to either `notify()` or `notifyAll()`. If the thread is not notified within the specified `timeout` period, it is automatically awakened when that amount of time has elapsed. If `timeout` is zero, the thread waits indefinitely, just as if `wait()` had been called without a `timeout` argument. After the thread is awakened, it has to regain the lock on this object before it can resume executing.

The `wait()` method can be called only by a thread that is the current owner of this object's lock. A thread holds the lock on this object while it is executing a `synchronized` instance method of the object or executing the body of a `synchronized` statement that synchronizes on the object. A thread can also hold the lock for a `Class` object if it is executing a `synchronized` `static` method of that class.

This method is `final`, so it cannot be overridden by subclasses.

`public final native void wait(long timeout, int nanos) throws InterruptedException`

Parameters

`timeout`

The maximum number of milliseconds to wait.

`nanos`

An additional number of nanoseconds to wait.

Throws

`IllegalMonitorStateException`

If the method is called from a thread that does not hold this object's lock.

`InterruptedException`

If another thread interrupted this thread.

Description

The `wait()` method causes a thread to wait until it is notified by another thread to stop waiting or until the specified amount of time has elapsed, whichever comes first. When `wait()` is called, the thread releases its lock on this object and waits until another thread notifies it to wake up through a call to either `notify()` or `notifyAll()`. If the thread is not notified within the specified time period, it is automatically awakened when that amount of time has elapsed. If `timeout` and `nanos` are zero, the thread waits indefinitely, just as if `wait()` had been called without any arguments. After the thread is awakened, it has to regain the lock on this object before it can resume executing.

The `wait()` method can be called only by a thread that is the current owner of this object's lock. A thread holds the lock on this object while it is executing a `synchronized` instance method of the object or executing the body of a `synchronized` statement that synchronizes on the object. A thread can also hold the lock for a `Class` object if it is executing a `synchronized` `static` method of that class.

Note that Sun's reference implementation of Java does not attempt to implement the precision implied by this method. Instead, it rounds to the nearest millisecond (unless `timeout` is `0`, in which case it rounds up to 1 millisecond) and calls `wait(long)`.

This method is `final`, so it cannot be overridden by subclasses.

Protected Instance Methods

clone

`protected native Object clone() throws CloneNotSupportedException`

Returns

A clone of this object.

Throws

`OutOfMemoryError`

If there is not enough memory to create the new object.

`CloneNotSupportedException`

If the object is of a class that does not support `clone()`.

Description

A *clone* of an object is another object of the same type that has all of its instance variables set to the same values as the object being cloned. In other words, a clone is an exact copy of the original object.

The `clone()` method of `Object` creates a new object that is a clone of this object. No constructor is used in creating the clone. The `clone()` method only clones an object if the class of that object indicates that its instances can be cloned. A class indicates that its objects can be cloned by implementing the `Cloneable` interface.

Although array objects do not implement the `Cloneable` interface, the `clone()` method works for arrays. The clone of an array is an array that has the same number of elements as the original array, and each element in the clone array has the same value as the corresponding element in the original array. Note that if an array element contains an object reference, the clone array contains a reference to the same object, not a copy of the object.

A subclass of `Object` can override the `clone()` method in `Object` to provide any additional functionality that is needed. For example, if an object contains references to other objects, the `clone()` method should recursively call the `clone()` methods of all the referenced objects. An overriding `clone()` method can throw a `CloneNotSupportedException` to indicate that particular objects cannot be cloned.

finalize

`protected void finalize() throws Throwable`

Throws

`Throwable`

For any reason that suits an overriding implementation of this method.

Description

The `finalize()` method is called by the garbage collector when it decides that an object can never be referenced again. The method gives an object a chance to perform any cleanup operations that are necessary before it is destroyed by the garbage collector.

The `finalize()` method of `Object` does nothing. A subclass overrides the `finalize()` method to perform any necessary cleanup operations. The overriding method should call `super.finalize()` as the very last thing it does, so that any `finalize()` method in the superclass is called.

When the garbage collector calls an object's `finalize()` method, the garbage collector does not immediately destroy the object because the `finalize()` method might do something that results in a reference to the object. Thus the garbage collector waits to destroy the object until it can again prove it is safe to do so. The next time the garbage collector decides it is safe to destroy the object, it does so without calling `finalize()` again. In other words, the garbage collector never calls the `finalize()` method more than once for a particular object.

A `finalize()` method can throw any kind of exception. An exception causes the `finalize()` method to stop running. The garbage collector then catches and ignores the exception, so it has no further effect on a program.

See Also

`CloneNotSupportedException`, `IllegalMonitorStateException`, `InterruptedException`, `OutOfMemoryError`, `Throwable`

PREVIOUS HOME NEXT

OutOfMemoryError

Name

OutOfMemoryError

Synopsis

Class Name:

`java.lang.OutOfMemoryError`

Superclass:

`java.lang.VirtualMachineError`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

An `OutOfMemoryError` is thrown when an attempt to allocate memory fails.

Class Summary

```
public class java.lang.OutOfMemoryError
             extends java.lang.VirtualMachineError {
  // Constructors
  public OutOfMemoryError();
  public OutOfMemoryError(String s);
}
```

Constructors

OutOfMemoryError

public OutOfMemoryError()

Description

This constructor creates an `OutOfMemoryError` with no associated detail message.

public OutOfMemoryError(String s)

Parameters

`s`

The detail message.

Description

This constructor creates an `OutOfMemoryError` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`

`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Error`, `Throwable`, `VirtualMachineError`

PREVIOUS
Object

HOME
BOOK INDEX

NEXT
Process

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Process

Name

Process

Synopsis

Class Name:

`java.lang.Process`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None that are provided on all platforms. However, a platform-specific version of Java should include at least one operating-system-specific subclass.

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The `Process` class describes processes that are started by the `exec()` method in the `Runtime` class. A `Process` object controls a process and gets information about it.

The `Process` class is an `abstract` class; therefore, it cannot be instantiated. The actual `Process` objects created by the `exec()` method belong to operating-system-specific subclasses of `Process` that implement the `Process` methods in platform-dependent ways.

Note that losing all references to a `Process` object, thereby making it garbage collectable, does not result in the underlying `Process` object dying. It merely means that there is no longer a Java object to control the process. The process itself continues to run asynchronously. In addition, no guarantees are made as to whether a controlled process will be able to continue after its parent process dies.

Class Summary

```
public abstract class java.lang.Process extends java.lang.Object {
    // Constructors
    public Process();
    // Instance Methods
    public abstract void destroy();
    public abstract int exitValue();
    public abstract InputStream getErrorStream();
    public abstract InputStream getInputStream();
    public abstract OutputStream getOutputStream();
    public abstract int waitFor();
}
```

Constructors

Process

`public Process()`

Description

> Creates a `Process` object.

Instance Methods

destroy

`abstract public void destroy()`

Description

> This method kills the process controlled by this object.

exitValue

`abstract public int exitValue()`

Returns

> The exit value of the process controlled by this object.

Throws

> `IllegalThreadStateException`
>
> > If the process is still running and the exit value is not yet available.

Description

> This method returns the exit value of the process that this object is controlling.
>
> The `waitFor()` method is a similar method that waits for the controlled process to terminate and then returns its exit value.

getErrorStream

`abstract public InputStream getErrorStream()`

Returns

> An `InputStream` object connected to the error stream of the process.

Description

This method returns an `InputStream` object that can read from the error stream of the process.

Although it is suggested that this `InputStream` not be buffered, the Java specification does not forbid such an implementation. In other words, although error output from programs is traditionally unbuffered, there is no guarantee that it won't be buffered. This means that error output written by the process may not be received immediately.

getInputStream

```
abstract public InputStream getInputStream()
```

Returns

An `InputStream` object that is connected to the standard output stream of the process.

Description

This method returns an `InputStream` object that can read from the standard output stream of the process.

This `InputStream` is likely to be buffered, which means that output written by the process may not be received immediately.

getOutputStream

```
abstract public OutputStream getOutputStream()
```

Returns

An `OutputStream` object that is connected to the standard input stream of the process.

Description

This method returns an `OutputStream` object that can write to the standard input stream of the process.

This `OutputStream` is likely to be buffered, which means that input sent to the process may not be received until the buffer fills up or a new line or carriage-return character is sent.

waitFor

`abstract public int waitFor()`

Returns

> The exit value of the process controlled by this object.

Throws

> `InterruptedException`
>
> > If another thread interrupts this thread while it is waiting for the process to exit.

Description

> This method returns the exit value of the process that this object is controlling. If the process is still running, the method waits until the process terminates and its exit value is available.
>
> The `exitValue()` method is a similar method that does not wait for the controlled process to terminate.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`InterruptedException`, `Object`, `Runtime`

PREVIOUS HOME NEXT
OutOfMemoryError BOOK INDEX Runnable

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Runnable

Name

Runnable

Synopsis

Interface Name:

`java.lang.Runnable`

Super-interface:

None

Immediate Sub-interfaces:

None

Implemented By:

`java.lang.Thread`

Availability:

JDK 1.0 or later

Description

The `Runnable` interface declares the `run()` method that is required for use with the `Thread` class. Any class that implements the `Runnable` interface must define a `run()` method. This method is the top-level code that is run by a thread.

Interface Declaration

```
public interface java.lang.Runnable {
    // Methods
    public abstract void run();
}
```

Methods

run

`public abstract void run()`

Description

> When a `Thread` object starts running a thread, it associates executable code with the thread by calling a `Runnable` object's `run()` method. The subsequent behavior of the thread is controlled by the `run()` method. Thus, a class that wants to perform certain operations in a separate thread should implement the `Runnable` interface and define an appropriate `run()` method. When the `run()` method called by a `Thread` object returns or throws an exception, the thread dies.

See Also

`Thread`, `ThreadGroup`

PREVIOUS Process | HOME BOOK INDEX | NEXT Runtime

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Runtime

Name

Runtime

Synopsis

Class Name:

`java.lang.Runtime`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The `Runtime` class provides access to various information about the environment in which a program is running. The Java run-time environment creates a single instance of this class that is associated with a program. The

Runtime class does not have any public constructors, so a program cannot create its own instances of the class. A program must call the getRuntime() method to get a reference to the current Runtime object.

Information about operating system features is accessible through the System class.

Class Summary

```
public class java.lang.Runtime extends java.lang.Object {
    // Class Methods
    public static Runtime getRuntime();
    public static void runFinalizersOnExit(boolean value);  // New in 1.1
    // Instance Methods
    public Process exec(String command);
    public Process exec(String command, String envp[]);
    public Process exec(String cmdarray[]);
    public Process exec(String cmdarray[], String envp[]);
    public void exit(int status);
    public native long freeMemory();
    public native void gc();
    public InputStream
           getLocalizedInputStream(InputStream in);      // Deprecated in 1.1
    public OutputStream
           getLocalizedOutputStream(OutputStream out);   // Deprecated in 1.1
    public synchronized void load(String filename);
    public synchronized void loadLibrary(String libname);
    public native void runFinalization();
    public native long totalMemory();
    public native void traceInstructions(boolean on);
    public native void traceMethodCalls(boolean on);
}
```

Class Methods

getRuntime

public static Runtime getRuntime()

Returns

A reference to the current Runtime object.

Description

This method returns a reference to the current Runtime object. Because the other methods of the

`Runtime` class are not `static`, a program must call this method first in order to get a reference to a `Runtime` object that can be used in calling the other methods.

runFinalizersOnExit

`public static void runFinalizersOnExit(boolean value)`

Availability

New as of JDK 1.1

Parameters

`value`

A `boolean` value that specifies whether or not finalization occurs on exit.

Throws

`SecurityException`

If the `checkExit()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method specifies whether or not the `finalize()` methods of all objects that have `finalize()` methods are run before the Java virtual machine exits. By default, the finalizers are not run on exit.

Instance Methods

exec

`public Process exec(String command) throws IOException`

Parameters

`command`

A string that contains the name of an external command and any arguments to be passed to it.

Returns

A `Process` object that controls the process started by this method.

Throws

`IOException`

If there is a problem finding or accessing the specified external command.

`SecurityException`

If the `checkExec()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method starts a new process to execute the given external command. The standard input, standard output, and standard error streams from the process are redirected to `OutputStream` and `InputStream` objects that are accessible through the `Process` object returned by this method.

Calling this method is equivalent to:

```
exec(command, null)
```

`public Process exec(String command, String[] envp) throws IOException`

Parameters

`command`

A string that contains the name of an external command and any arguments to be passed to it.

`envp`

An array of strings that specifies the values for the environment variables of the new process. Each `String` in the array should be of the form *variableName =value*. If `envp` is `null`, the values of the environment variables in the current process are copied to the new process.

Returns

A `Process` object that controls the process started by this method.

Throws

`IOException`

If there is a problem finding or accessing the specified external command.

`SecurityException`

If the `checkExec()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method starts a new process to execute the given external command. The standard input, standard output, and standard error streams from the process are redirected to `OutputStream` and `InputStream` objects that are accessible through the `Process` object returned by this method.

The method parses the `command` string into words that are separated by whitespace. It creates a `String` object for each word and places word `String` objects into an array. If that array is called `commandArray`, calling this method is equivalent to:

```
exec(commmandArray, envp)
```

`public Process exec(String[] commandArray) throws IOException`

Parameters

`commandArray`

An array of strings that contains separate strings for the name of an external command and any arguments to be passed to it. The first string in the array must be the command name.

Returns

A `Process` object that controls the process started by this method.

Throws

`IOException`

If there is a problem finding or accessing the specified external command.

`SecurityException`

If the `checkExec()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method starts a new process to execute the given external command. The standard input, standard output, and standard error streams from the process are redirected to `OutputStream` and `InputStream` objects that are accessible through the `Process` object returned by this method.

Calling this method is equivalent to:

```
exec(commandArray, null)
```

`public Process exec(String[] commandArray, String[] envp) throws IOException`

Parameters

`commandArray`

An array of strings that contains separate strings for the name of an external command and any arguments to be passed to it. The first string in the array must be the command name.

`envp`

An array of strings that specifies the values for the environment variables of the new process. Each `String` in the array should be of the form *variableName =value*. If `envp` is `null`, the values of the environment variables in the current process are copied to the new process.

Returns

A `Process` object that controls the process started by this method.

Throws

`IOException`

If there is a problem finding or accessing the specified external command.

`SecurityException`

If the `checkExec()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method starts a new process to execute the given external command. The standard input, standard output, and standard error streams from the process are redirected to `OutputStream` and `InputStream` objects that are accessible through the `Process` object returned by this method.

exit

`public void exit(int status)`

Parameters

`status`

The exit status code to use.

Throws

`SecurityException`

If the `checkExit()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method causes the Java virtual machine to exit with the given status code. By convention, a nonzero status code indicates abnormal termination. This method never returns.

freeMemory

`public native long freeMemory()`

Returns

An estimate of the number of free bytes in system memory.

Description

This method returns an estimate of the number of free bytes in system memory. The value returned by this method is always less than the value returned by `totalMemory()`. Additional memory may be freed by calling the `gc()` method.

gc

`public native void gc()`

Description

This method causes the Java virtual machine to run the garbage collector in the current thread.

The garbage collector finds objects that will never be used again because there are no live references to them. After it finds these objects, the garbage collector frees the storage occupied by these objects.

The garbage collector is normally run continuously in a thread with the lowest possible priority, so that it works intermittently to reclaim storage. The `gc()` method allows a program to invoke the garbage collector explicitly when necessary.

getLocalizedInputStream

`public InputStream getLocalizedInputStream(InputStream in)`

Availability

Deprecated as of JDK 1.1

Parameters

`in`

An `InputStream` object that is to be localized.

Returns

The localized `InputStream`.

Description

This method returns an `InputStream` object that converts characters from the local character set to Unicode. For example, if the `InputStream` uses an 8-bit character set with values less than 128 representing Cyrillic letters, this method maps those characters to the corresponding Unicode characters in the range `'\u0400'` to `'\u04FF'`.

This method is deprecated as of JDK 1.1. You should instead use the new `InputStreamReader` and `BufferedReader` classes to convert characters from the local character set to Unicode.

getLocalizedOutputStream

`public OutputStream getLocalizedOutputStream(OutputStream out)`

Availability

Deprecated as of JDK 1.1

Parameters

`out`

An `OutputStream` object that is to be localized.

Returns

The localized `OutputStream`.

Description

This method returns an `OutputStream` object that converts characters from Unicode to the local character set. For example, if the local character set is an 8-bit character set with values less than 128 representing Cyrillic letters, this method maps Unicode characters in the range `'\u0400'` to `'\u04FF'` to the appropriate characters in the local character set.

This method is deprecated as of JDK 1.1. You should instead use the new `OutputStreamWriter` and `BufferedWriter` classes to convert characters from Unicode to the local character set.

load

`public synchronized void load(String filename)`

Parameters

`filename`

A string that specifies the complete path of the file to be loaded.

Throws

`SecurityException`

If the `checkLink()` method of the `SecurityManager` throws a `SecurityException`.

`UnsatisfiedLinkError`

If the method is unsuccessful in loading the specified dynamically linked library.

Description

This method loads the specified dynamically linked library.

It is often more convenient to call the `load()` method of the `System` class because it does not require getting a `Runtime` object.

loadLibrary

`public synchronized void loadLibrary(String libname)`

Parameters

`libname`

A string that specifies the name of a dynamically linked library.

Throws

`SecurityException`

If the `checkLink()` method of the `SecurityManager` throws a `SecurityException`.

`UnsatisfiedLinkError`

If the method is unsuccessful in loading the specified dynamically linked library.

Description

This method loads the specified dynamically linked library. It looks for the specified library in a platform-specific way.

It is often more convenient to call the `loadLibrary()` method of the `System` class because it does not require getting a `Runtime` object.

runFinalization

`public native void runFinalization()`

Description

This method causes the Java virtual machine to run the `finalize()` methods of any objects in the finalization queue in the current thread.

When the garbage collector discovers that there are no references to an object, it checks to see if the object has a `finalize()` method that has never been called. If the object has such a `finalize()` method, the object is placed in the finalization queue. While there is a reference to the object in the finalization queue, the object is no longer considered garbage-collectable.

Normally, the objects in the finalization queue are handled by a separate finalization thread that runs continuously at a very low priority. The finalization thread removes an object from the queue and calls its `finalize()` method. As long as the `finalize()` method does not generate a reference to the object, the object again becomes available for garbage collection.

Because the finalization thread runs at a very low priority, there may be a long delay from the time that an object is put on the finalization queue until the time that its `finalize()` method is called. The `runFinalization()` method allows a program to run the `finalize()` methods explicitly. This can be useful when there is a shortage of some resource that is released by a `finalize()` method.

totalMemory

`public native long totalMemory()`

Returns

The total number of bytes in system memory.

Description

This method returns the total number of bytes in system memory in the Java virtual machine. The total includes the number of bytes of memory being used by allocated objects, as well as the number of free bytes available for allocating additional objects. An estimate of the number of free bytes in system memory is available through the `freeMemory()` method.

traceInstructions

`public native void traceInstructions(boolean on)`

Parameters

`on`

A `boolean` value that specifies if instructions are to be traced. `true` if instructions are to be traced; otherwise `false`.

Description

This method controls whether or not the Java virtual machine outputs a detailed trace of each instruction that is executed. The `boolean` parameter causes tracing to be turned on or off. The tracing of instructions is only possible in a Java virtual machine that was compiled with the tracing option enabled. Production releases of the Java virtual machine are generally not compiled with tracing enabled.

traceMethodCalls

`public native void traceMethodCalls(boolean on)`

Parameters

`on`

A `boolean` value that specifies if method calls are to be traced. `true` if instructions are to be traced; otherwise `false`.

Description

This method controls whether or not the Java virtual machine outputs a detailed trace of each method that is invoked. The `boolean` parameter causes tracing to be turned on or off. The tracing of instructions is only possible in a Java virtual machine that was compiled with the tracing option enabled. Production releases of the Java virtual machine are generally not compiled with tracing enabled.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`IOException`, `Object`, `Process`, `SecurityException`, `SecurityManager`, `System`, `UnsatisfiedLinkError`

PREVIOUS
Runnable

HOME
BOOK INDEX

NEXT
RuntimeException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

RuntimeException

Name

RuntimeException

Synopsis

Class Name:

`java.lang.RuntimeException`

Superclass:

`java.lang.Exception`

Immediate Subclasses:

`java.lang.ArithmeticException`,

`java.lang.ArrayStoreException`,

`java.lang.ClassCastException`,

`java.lang.IllegalArgumentException`,

`java.lang.IllegalMonitorStateException`,

`java.lang.IllegalStateException`,

`java.lang.IndexOutOfBoundsException`,

java.lang.NegativeArraySizeException,

java.lang.NullPointerException,

java.lang.SecurityException,

java.util.EmptyStackException,

java.util.MissingResourceException,

java.util.NoSuchElementException

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The RuntimeException class is the superclass of the standard run-time exceptions that can be thrown in Java. The appropriate subclass of RuntimeException is thrown in response to a run-time error detected at the virtual machine level. A run-time exception represents a run-time condition that can occur generally in any Java method, so a method is not required to declare that it throws any of the run-time exceptions.

A Java program should try to handle all of the standard run-time exception classes, since they represent routine abnormal conditions that should be anticipated and caught to prevent program termination.

Class Summary

```
public class java.lang.RuntimeException extends java.lang.Exception {
  // Constructors
  public RuntimeException();
  public RuntimeException(String s);
}
```

Constructors

RuntimeException

public RuntimeException()

Description

This constructor creates a `RuntimeException` with no associated detail message.

public RuntimeException(String s)

Parameters

`s`

The detail message.

Description

This constructor creates a `RuntimeException` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`ArithmeticException`, `ArrayStoreException`, `ClassCastException`, `EmptyStackException`, `IllegalArgumentException`, `IllegalMonitorStateException`, `IllegalStateException`, `IndexOutOfBoundsException`, `MissingResourceException`, `NegativeArraySizeException`, `NoSuchElementException`, `NullPointerException`, `SecurityException`, `Throwable`

PREVIOUS HOME NEXT
Runtime BOOK INDEX SecurityException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

SecurityException

Name

SecurityException

Synopsis

Class Name:

`java.lang.SecurityException`

Superclass:

`java.lang.RuntimeException`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

A `SecurityException` is thrown in response to an attempt to perform an operation that violates the security policy implemented by the installed `SecurityManager` object.

Class Summary

```
public class java.lang.SecurityException
             extends java.lang.RuntimeException {
  // Constructors
  public SecurityException();
  public SecurityException(String s);
}
```

Constructors

SecurityException

public SecurityException()

Description

> This constructor creates a `SecurityException` with no associated detail message.

public SecurityException(String s)

Parameters

> `s`
>
> > The detail message.

Description

> This constructor creates a `SecurityException` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`

`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Exception`, `RuntimeException`, `SecurityManager`, `Throwable`

PREVIOUS HOME NEXT

RuntimeException BOOK INDEX SecurityManager

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

SecurityManager

Name

SecurityManager

Synopsis

Class Name:

```
java.lang.SecurityManager
```

Superclass:

```
java.lang.Object
```

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The SecurityManager class provides a way of implementing a comprehensive security policy for a Java program. As of this writing, SecurityManager objects are used primarily by Web browsers to establish security policies for applets. However, the use of a SecurityManager object is appropriate in any situation where a hosting environment wants to limit the actions of hosted programs.

The SecurityManager class contains methods that are called by methods in other classes to ask for permission to do something that can affect the security of the system. These permission methods all have names that begin with check. If a check method does not permit an action, it throws a SecurityException or returns a value that indicates the lack of permission. The SecurityManager class provides default implementations of all of the check methods. These default implementations are the most restrictive possible implementations; they simply deny permission to do anything that can affect the security of the system.

The SecurityManager class is an abstract class. A hosting environment should define a subclass of SecurityManager that implements an appropriate security policy. To give the subclass of SecurityManager control over security, the hosting environment creates an instance of the class and installs it by passing it to the setSecurityManager() method of the System class. Once a SecurityManager object is installed, it cannot be changed. If the setSecurityManager() method is called any additional times, it throws a SecurityException.

The methods in other classes that want to ask the SecurityManager for permission to do something are able to access the SecurityManager object by calling the getSecurityManager() method of the System class. This method returns the SecurityManager object, or null to indicate that there is no SecurityManager installed.

Class Summary

```
public abstract class java.lang.SecurityManager extends java.lang.Object {
    // Constructors
    protected SecurityManager();
    // Variables
    protected boolean inCheck;
    // Instance Methods
    public void checkAccept(String host, int port);
    public void checkAccess(Thread t);
    public void checkAccess(ThreadGroup g);
    public void checkAwtEventQueueAccess();                  // New in 1.1
    public void checkConnect(String host, int port);
    public void checkConnect(String host, int port, Object context);
    public void checkCreateClassLoader();
    public void checkDelete(String file);
    public void checkExec(String cmd);
    public void checkExit(int status);
    public void checkLink(String libname);
    public void checkListen(int port);
    public void checkMemberAccess(Class clazz, int which);   // New in 1.1
    public void checkMulticast(InetAddress maddr);           // New in 1.1
    public void checkMulticast(InetAddress maddr, byte ttl); // New in 1.1
    public void checkPackageAccess();
    public void checkPackageDefinition();
    public void checkPrintJobAccess();                       // New in 1.1
    public void checkPropertiesAccess();
    public void checkPropertyAccess(String key);
    public void checkRead(int fd);
    public void checkRead(String file);
    public void checkRead(String file, Object context);
    public void checkSecurityAccess(String action);          // New in 1.1
    public void checkSetFactory();
    public void checkSystemClipboardAccess();                // New in 1.1
    public boolean checkTopLevelWindow();
    public void checkWrite(int fd);
    public void checkWrite(String file);
    public boolean getInCheck();
    public Object getSecurityContext();
    public ThreadGroup getThreadGroup();                     // New in 1.1
    // Protected Instance Methods
    protected int classDepth(String name);
    protected int classLoaderDepth();
    protected ClassLoader currentClassLoader();
    protected Class currentLoadedClass();                    // New in 1.1
    protected Class[] getClassContext();
    protected boolean inClass(String name);
    protected boolean inClassLoader();
}
```

Variables

inCheck

`protected boolean inCheck = false`

Description

> This variable indicates whether or not a security check is in progress. A subclass of `SecurityManager` should set this variable to `true` while a security check is in progress.
>
> This variable can be useful for security checks that require access to resources that a hosted program may not be permitted to access. For example, a security policy might be based on the contents of a

permissions file. This means that the various `check` methods need to read information from a file to decide what to do. Even though a hosted program may not be allowed to read files, the `check` methods can allow such reads when `inCheck` is `true` to support this style of security policy.

Constructors

SecurityManager

`protected SecurityManager()`

Throws

`SecurityException`

If a `SecurityManager` object already exists. In other words, if `System.getSecurityManager()` returns a value other than `null`.

Description

Creates a new `SecurityManager` object. This constructor cannot be called if there is already a current `SecurityManager` installed for the program.

Public Instance Methods

checkAccept

`public void checkAccept(String host, int port)`

Parameters

`host`

The name of the host machine.

`port`

A port number.

Throws

`SecurityException`

If the caller does not have permission to accept the connection.

Description

This method decides whether or not to allow a connection from the given host on the given port to be accepted. An implementation of the method should throw a `SecurityException` to deny permission to accept the connection. The method is called by the `accept()` method of the `java.net.ServerSocket` class.

The `checkAccept()` method of `SecurityManager` always throws a `SecurityException`.

checkAccess

`public void checkAccess(Thread g)`

Parameters

`g`

A reference to a `Thread` object.

Throws

`SecurityException`

If the current thread does not have permission to modify the specified thread.

Description

This method decides whether or not to allow the current thread to modify the specified `Thread`. An implementation of the method should throw a `SecurityException` to deny permission to modify the thread. Methods of the `Thread` class that call this method include `stop()`, `suspend()`, `resume()`, `setPriority()`, `setName()`, and `setDaemon()`.

The `checkAccess()` method of `SecurityManager` always throws a `SecurityException`.

`public void checkAccess(ThreadGroup g)`

Parameters

`g`

A reference to a `ThreadGroup` object.

Throws

`SecurityException`

If the current thread does not have permission to modify the specified thread group.

Description

This method decides whether or not to allow the current thread to modify the specified `ThreadGroup`. An implementation of the method should throw a `SecurityException` to deny permission to modify the thread group. Methods of the `ThreadGroup` class that call this method include `setDaemon()`, `setMaxPriority()`, `stop()`, `suspend()`, `resume()`, and `destroy()`.

The `checkAccess()` method of `SecurityManager` always throws a `SecurityException`.

checkAwtEventQueueAccess

`public void checkAwtEventQueueAccess()`

Availability

New as of JDK 1.1

Throws

`SecurityException`

If the caller does not have permission to access the AWT event queue.

Description

This method decides whether or not to allow access to the AWT event queue. An implementation of the method should throw a `SecurityException` to deny permission to access the event queue. The method is called by the `getSystemEventQueue()` method of the `java.awt.Toolkit` class.

The `checkAwtEventQueueAccess()` method of `SecurityManager` always throws a `SecurityException`.

checkConnect

`public void checkConnect(String host, int port)`

Parameters

`host`

The name of the host.

`port`

A port number. A value of `-1` indicates an attempt to determine the IP address of given hostname.

Throws

`SecurityException`

If the caller does not have permission to open the socket connection.

Description

This method decides whether or not to allow a socket connection to the given host on the given port to be opened. An implementation of the method should throw a `SecurityException` to deny permission to open the connection. The method is called by the constructors of the `java.net.Socket` class, the `send()` and `receive()` methods of the `java.net.DatagramSocket` class, and the `getByName()` and `getAllByName()` methods of the `java.net.InetAddress` class.

The `checkConnect()` method of `SecurityManager` always throws a `SecurityException`.

`public void checkConnect(String host, int port, Object context)`

Parameters

`host`

The name of the host.

`port`

A port number. A value of `-1` indicates an attempt to determine the IP address of given host name.

`context`

A security context object returned by this object's `getSecurityContext()` method.

Throws

`SecurityException`

If the specified security context does not have permission to open the socket connection.

Description

This method decides whether or not to allow a socket connection to the given host on the given port to be opened for the specified security context. An implementation of the method should throw a `SecurityException` to deny permission to open the connection.

The `checkConnect()` method of `SecurityManager` always throws a `SecurityException`.

checkCreateClassLoader

`public void checkCreateClassLoader()`

Throws

`SecurityException`

If the caller does not have permission to create a `ClassLoader` object.

Description

This method decides whether or not to allow a `ClassLoader` object to be created. An implementation of the method should throw a `SecurityException` to deny permission to create a `ClassLoader`. The method is called by the constructor of the `ClassLoader` class.

The `checkCreateClassLoader()` method of `SecurityManager` always throws a `SecurityException`.

checkDelete

`public void checkDelete(String file)`

Parameters

`file`

The name of a file.

Throws

`SecurityException`

If the caller does not have permission to delete the specified file.

Description

This method decides whether or not to allow a file to be deleted. An implementation of the method should throw a `SecurityException` to deny permission to delete the specified file. The method is called by the `delete()` method of the `java.io.File` class.

The `checkDelete()` method of `SecurityManager` always throws a `SecurityException`.

checkExec

`public void checkExec(String cmd)`

Parameters

`cmd`

The name of an external command.

Throws

`SecurityException`

If the caller does not have permission to execute the specified command.

Description

This method decides whether or not to allow an external command to be executed. An implementation of the method should throw a `SecurityException` to deny permission to execute the specified command. The method is called by the `exec()` methods of the `Runtime` and `System` classes.

The `checkExec()` method of `SecurityManager` always throws a `SecurityException`.

checkExit

`public void checkExit(int status)`

Parameters

`status`

An exit status code.

Throws

`SecurityException`

If the caller does not have permission to exit the Java virtual machine with the given status code.

Description

This method decides whether or not to allow the Java virtual machine to exit with the given status code. An implementation of the method should throw a `SecurityException` to deny permission to exit with the specified status code. The method is called by the `exit()` methods of the `Runtime` and `System` classes.

The `checkExit()` method of `SecurityManager` always throws a `SecurityException`.

checkLink

`public void checkLink(String lib)`

Parameters

`lib`

The name of a library.

Throws

`SecurityException`

If the caller does not have permission to load the specified library.

Description

This method decides whether to allow the specified library to be loaded. An implementation of the method should throw a `SecurityException` to deny permission to load the specified library. The method is called by the `load()` and `loadLibrary()` methods of the `Runtime` and `System` classes.

The `checkLink()` method of `SecurityManager` always throws a `SecurityException`.

checkListen

`public void checkListen(int port)`

Parameters

`port`

A port number.

Throws

`SecurityException`

If the caller does not have permission to listen on the specified port.

Description

This method decides whether or not to allow the caller to listen on the specified port. An implementation of the method should throw a `SecurityException` to deny permission to listen on the specified port. The method is called by the constructors of the `java.net.ServerSocket` class and by the constructor of the `java.net.DatagramSocket` class that takes one argument.

The `checkListen()` method of `SecurityManager` always throws a `SecurityException`.

checkMemberAccess

`public void checkMemberAccess(Class clazz, int which)`

Availability

New as of JDK 1.1

Parameters

`clazz`

A `Class` object.

`which`

The value `Member.PUBLIC` for the set of all `public` members including inherited members or the value `Member.DECLARED` for the set of all declared members of the specified class or interface.

Throws

`SecurityException`

If the caller does not have permission to access the members of the specified class or interface.

Description

This method decides whether or not to allow access to the members of the specified `Class` object. An implementation of the method should throw a `SecurityException` to deny permission to access the members. Methods of the `Class` class that call this method include `getField()`, `getFields()`, `getDeclaredField()`, `getDeclaredFields()`, `getMethod()`, `getMethods()`, `getDeclaredMethod()`, `getDeclaredMethods()`, `getConstructor()`, `getConstructors()`, `getDeclaredConstructor()`, `getDeclaredConstructors()`, and `getDeclaredClasses()`.

The `checkMemberAccess()` method of `SecurityManager` always throws a `SecurityException`.

checkMulticast

`public void checkMulticast(InetAddress maddr)`

Availability

New as of JDK 1.1

Parameters

`maddr`

An `InetAddress` object that represents a multicast address.

Throws

`SecurityException`

If the current thread does not have permission to use the specified multicast address.

Description

This method decides whether or not to allow the current thread to use the specified multicast `InetAddress`. An implementation of the method should throw a `SecurityException` to deny permission to use the multicast address. The method is called by the `send()` method of `java.net.DatagramSocket` if the packet is being sent to a multicast address. The method is also called by the `joinGroup()` and `leaveGroup()` methods of `java.net.MulticastSocket`.

The `checkMulticast()` method of `SecurityManager` always throws a `SecurityException`.

`public void checkMulticast(InetAddress maddr, byte ttl)`

Availability

New as of JDK 1.1

Parameters

`maddr`

An `InetAddress` object that represents a multicast address.

`ttl`

The time-to-live (TTL) value.

Throws

`SecurityException`

If the current thread does not have permission to use the specified multicast address and TTL value.

Description

This method decides whether or not to allow the current thread to use the specified multicast

`InetAddress` and TTL value. An implementation of the method should throw a `SecurityException` to deny permission to use the multicast address. The method is called by the `send()` method of `java.net.MulticastSocket`.

The `checkMulticast()` method of `SecurityManager` always throws a `SecurityException`.

checkPackageAccess

`public void checkPackageAccess(String pkg)`

Parameters

`pkg`

The name of a package.

Throws

`SecurityException`

If the caller does not have permission to access the specified package.

Description

This method decides whether or not to allow the specified package to be accessed. An implementation of the method should throw a `SecurityException` to deny permission to access the specified package. The method is intended to be called by implementations of the `loadClass()` method in subclasses of the `ClassLoader` class.

The `checkPackageAccess()` method of `SecurityManager` always throws a `SecurityException`.

checkPackageDefinition

`public void checkPackageDefinition(String pkg)`

Parameters

`pkg`

The name of a package.

Throws

`SecurityException`

If the caller does not have permission to define classes in the specified package.

Description

This method decides whether or not to allow the caller to define classes in the specified package. An implementation of the method should throw a `SecurityException` to deny permission to create classes in the specified package. The method is intended to be called by implementations of the `loadClass()` method in subclasses of the `ClassLoader` class.

The `checkPackageDefinition()` method of `SecurityManager` always throws a `SecurityException`.

checkPrintJobAccess

`public void checkPrintJobAccess()`

Availability

New as of JDK 1.1

Throws

`SecurityException`

If the caller does not have permission to initiate a print job request.

Description

This method decides whether or not to allow the caller to initiate a print job request. An implementation of the method should throw a `SecurityException` to deny permission to initiate the request.

The `checkPrintJobAccess()` method of `SecurityManager` always throws a `SecurityException`.

checkPropertiesAccess

`public void checkPropertiesAccess()`

Throws

`SecurityException`

If the caller does not have permission to access the system properties.

Description

This method decides whether or not to allow the caller to access and modify the system properties. An implementation of the method should throw a `SecurityException` to deny permission to access and modify the properties. Methods of the `System` class that call this method include `getProperties()` and `setProperties()`.

The `checkPropertiesAccess()` method of `SecurityManager` always throws a `SecurityException`.

checkPropertyAccess

```
public void checkPropertyAccess(String key)
```

Parameters

`key`

The name of an individual system property.

Throws

`SecurityException`

If the caller does not have permission to access the specified system property.

Description

This method decides whether or not to allow the caller to access the specified system property. An implementation of the method should throw a `SecurityException` to deny permission to access the property. The method is called by the `getProperty()` method of the `System` class.

The `checkPropertyAccess()` method of `SecurityManager` always throws a `SecurityException`.

checkRead

`public void checkRead(FileDescriptor fd)`

Parameters

`fd`

A reference to a `FileDescriptor` object.

Throws

`SecurityException`

If the caller does not have permission to read from the given file descriptor.

Description

This method decides whether or not to allow the caller to read from the specified file descriptor. An implementation of the method should throw a `SecurityException` to deny permission to read from the file descriptor. The method is called by the constructor of the `java.io.FileInputStream` class that takes a `FileDescriptor` argument.

The `checkRead()` method of `SecurityManager` always throws a `SecurityException`.

`public void checkRead(String file)`

Parameters

`file`

The name of a file.

Throws

`SecurityException`

If the caller does not have permission to read from the named file.

Description

This method decides whether or not to allow the caller to read from the named file. An implementation of the method should throw a `SecurityException` to deny permission to read from the file. The method is called by constructors of the `java.io.FileInputStream` and `java.io.RandomAccessFile` classes, as well as by the `canRead()`, `exists()`, `isDirectory()`, `isFile()`, `lastModified()`, `length()`, and `list()` methods of the `java.io.File` class.

The `checkRead()` method of `SecurityManager` always throws a `SecurityException`.

`public void checkRead(String file, Object context)`

Parameters

`file`

The name of a file.

`context`

A security context object returned by this object's `getSecurityContext()` method.

Throws

`SecurityException`

If the specified security context does not have permission to read from the named file.

Description

This method decides whether or not to allow the specified security context to read from the named file. An implementation of the method should throw a `SecurityException` to deny permission to read from the file.

The `checkRead()` method of `SecurityManager` always throws a `SecurityException`.

checkSecurityAccess

`public void checkSecurityAccess(String action)`

Availability

New as of JDK 1.1

Parameters

`action`

A string that specifies a security action.

Throws

`SecurityException`

If the caller does not have permission to perform the specified security action.

Description

This method decides whether to allow the caller to perform the specified security action. An implementation of the method should throw a `SecurityException` to deny permission to perform the action. The method is called by many of the methods in the `java.security.Identity` and `java.security.Security` classes.

The `checkSecurityAccess()` method of `SecurityManager` always throws a `SecurityException`.

checkSetFactory

`public void checkSetFactory()`

Throws

`SecurityException`

If the caller does not have permission to set the factory class to be used by another class.

Description

This method decides whether to allow the caller to set the factory class to be used by another class. An implementation of the method should throw a `SecurityException` to deny permission to set the factory class. The method is called by the `setSocketFactory()` method of the `java.net.ServerSocket` class, the `setSocketImplFactory()` method of the `java.net.Socket` class, the `setURLStreamHandlerFactory()` method of the `java.net.URL` class, and the `setContentHandlerFactory()` method of the `java.net.URLConnection` class.

The `checkSetFactory()` method of `SecurityManager` always throws a `SecurityException`.

checkSystemClipboardAccess

`public void checkSystemClipboardAccess()`

Availability

New as of JDK 1.1

Throws

`SecurityException`

If the caller does not have permission to access the system clipboard.

Description

This method decides whether or not to allow the caller to access the system clipboard. An implementation of the method should throw a `SecurityException` to deny permission to access the system clipboard.

The `checkSystemClipboardAccess()` method of `SecurityManager` always throws a `SecurityException`.

checkTopLevelWindow

`public boolean checkTopLevelWindow(Object window)`

Parameters

`window`

A window object.

Returns

`true` if the caller is trusted to put up the specified top-level window; otherwise `false`.

Description

This method decides whether or not to trust the caller to put up the specified top-level window. An implementation of the method should return `false` to indicate that the caller is not trusted. In this case, the hosting environment can still decide to display the window, but the window should include a visual indication that it is not trusted. If the caller is trusted, the method should return `true`, and the window can be displayed without any special indication.

The `checkTopLevelWindow()` method of `SecurityManager` always returns `false`.

checkWrite

`public void checkWrite(FileDescriptor fd)`

Parameters

`fd`

A `FileDescriptor` object.

Throws

`SecurityException`

If the caller does not have permission to write to the given file descriptor.

Description

This method decides whether or not to allow the caller to write to the specified file descriptor. An implementation of the method should throw a `SecurityException` to deny permission to write to the file descriptor. The method is called by the constructor of the `java.io.FileOutputStream` class that takes a `FileDescriptor` argument.

The `checkWrite()` method of `SecurityManager` always throws a `SecurityException`.

`public void checkWrite(String file)`

Parameters

`file`

The name of a file.

Throws

`SecurityException`

If the caller does not have permission to read from the named file.

Description

This method decides whether or not to allow the caller to write to the named file. An implementation of the method should throw a `SecurityException` to deny permission to write to the file. The method is called by constructors of the `java.io.FileOutputStream` and `java.io.RandomAccessFile` classes, as well as by the `canWrite()`, `mkdir()`, and `renameTo()` methods of the `java.io.File` class.

The `checkWrite()` method of `SecurityManager` always throws a `SecurityException`.

getInCheck

`public boolean getInCheck()`

Returns

`true` if a security check is in progress; otherwise `false`.

Description

This method returns the value of the `SecurityManager` object's `inCheck` variable, which is `true` if a security check is in progress and `false` otherwise.

getSecurityContext

`public Object getSecurityContext()`

Returns

An implementation-dependent object that contains enough information about the current execution environment to perform security checks at a later time.

Description

This method is meant to create an object that encapsulates information about the current execution environment. The resulting security context object is used by specific versions of the `checkConnect()` and `checkRead()` methods. The intent is that such a security context object can be used by a trusted method to determine whether or not another, untrusted method can perform a particular operation.

The `getSecurityContext()` method of `SecurityManager` simply returns `null`. This method should be overridden to return an appropriate security context object for the security policy that is being implemented.

getThreadGroup

`public ThreadGroup getThreadGroup()`

Availability

New as of JDK 1.1

Returns

A `ThreadGroup` in which to place any threads that are created when this method is called.

Description

This method returns the appropriate parent `ThreadGroup` for any threads that are created when the method is called. The `getThreadGroup()` method of `SecurityManager` simply returns the `ThreadGroup` of the current thread. This method should be overridden to return an appropriate `ThreadGroup`.

Protected Instance Methods

classDepth

`protected native int classDepth(String name)`

Parameters

`name`

The fully qualified name of a class.

Returns

The number of pending method invocations from the top of the stack to a call to a method of the given class; `-1` if no stack frame in the current thread is associated with a call to a method in the given class.

Description

This method returns the number of pending method invocations between this method invocation and an invocation of a method associated with the named class.

classLoaderDepth

`protected native int classLoaderDepth()`

Returns

> The number of pending method invocations from the top of the stack to a call to a method that is associated with a class that was loaded by a `ClassLoader` object; `-1` if no stack frame in the current thread is associated with a call to such a method.

Description

> This method returns the number of pending method invocations between this method invocation and an invocation of a method associated with a class that was loaded by a `ClassLoader` object.

currentClassLoader

`protected native ClassLoader currentClassLoader()`

Returns

> The most recent `ClassLoader` object executing on the stack.

Description

> This method finds the most recent pending invocation of a method associated with a class that was loaded by a `ClassLoader` object. The method then returns the `ClassLoader` object that loaded that class.

currentLoadedClass

`protected Class currentLoadedClass()`

Availability

> New as of JDK 1.1

Returns

> The most recent `Class` object loaded by a `ClassLoader`.

Description

This method finds the most recent pending invocation of a method associated with a class that was loaded by a `ClassLoader` object. The method then returns the `Class` object for that class.

getClassContext

`protected native Class[] getClassContext()`

Returns

An array of `Class` objects that represents the current execution stack.

Description

This method returns an array of `Class` objects that represents the current execution stack. The length of the array is the number of pending method calls on the current thread's stack, not including the call to `getClassContext()`. Each element of the array references a `Class` object that describes the class associated with the corresponding method call. The first element of the array corresponds to the most recently called method, the second element is that method's caller, and so on.

inClass

`protected boolean inClass(String name)`

Parameters

`name`

The fully qualified name of a class.

Returns

`true` if there is a pending method invocation on the stack for a method of the given class; otherwise `false`.

Description

This method determines whether or not there is a pending method invocation that is associated with the named class.

inClassLoader

`protected boolean inClassLoader()`

Returns

> `true` if there is a pending method invocation on the stack for a method of a class that was loaded by a `ClassLoader` object; otherwise `false`.

Description

> This method determines whether or not there is a pending method invocation that is associated with a class that was loaded by a `ClassLoader` object. The method returns `true` only if the `currentClassLoader()` method does not return `null`.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Class`, `ClassLoader`, `DatagramSocket`, `File`, `FileDescriptor`, `FileInputStream`, `FileOutputStream`, `InetAddress`, `MulticastSocket`, `Object`, `RandomAccessFile`, `Runtime`, `SecurityException`, `ServerSocket`, `Socket`, `System`, `Thread`, `ThreadGroup`, `Toolkit`, `URL`, `URLConnection`

PREVIOUS
SecurityException

HOME
BOOK INDEX

NEXT
Short

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Short

Name

Short

Synopsis

Class Name:

`java.lang.Short`

Superclass:

`java.lang.Number`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `Short` class provides an object wrapper for a `short` value. This is useful when you need to treat a `short` value as an object. For example, there are a number of utility methods that take a reference to an `Object` as one of their arguments. You cannot specify a `short` value for one of these arguments, but you can provide a reference to a `Short` object that encapsulates the `byte` value. Furthermore, the `Short` class is necessary as of JDK 1.1 to support the

Reflection API and class literals.

The `Short` class also provides a number of utility methods for converting `short` values to other primitive types and for converting `short` values to strings and vice-versa.

Class Summary

```
public final class java.lang.Short extends java.lang.Number {
  // Constants
  public static final short MAX_VALUE;
  public static final short MIN_VALUE;
  public static final Class TYPE;
  // Constructors
  public Short(short value);
  public Short(String s);
  // Class Methods
  public static Short decode(String nm);
  public static short parseShort(String s);
  public static short parseShort(String s, int radix);
  public static String toString(short s);
  public static Short valueOf(String s, int radix);
  public static Short valueOf(String s);
  // Instance Methods
  public byte byteValue();
  public double doubleValue();
  public boolean equals(Object obj);
  public float floatValue();
  public int hashCode();
  public int intValue();
  public long longValue();
  public short shortValue();
  public String toString();
}
```

Constants

MAX_VALUE

public static final short MAX_VALUE= 32767

The largest value that can be represented by a `short`.

MIN_VALUE

public static final byte MIN_VALUE= -32768

The smallest value that can be represented by a `short`.

TYPE

public static final Class TYPE

The `Class` object that represents the primitive type `short`. It is always true that `Short.TYPE` == `short.class`.

Constructors

Short

public Short(short value)

Parameters

> `value`
>
> > The `short` value to be encapsulated by this object.

Description

> Creates a `Short` object with the specified `short` value.

public Short(String s) throws NumberFormatException

Parameters

> `s`
>
> > The string to be made into a `Short` object.

Throws

> `NumberFormatException`
>
> > If the sequence of characters in the given `String` does not form a valid `short` literal.

Description

> Constructs a `Short` object with the value specified by the given string. The string should consist of one or more digit characters. The digit characters can be preceded by a single `` `-' `` character. If the string contains any other characters, the constructor throws a `NumberFormatException`.

Class Methods

decode

public static Short decode(String nm) throws NumberFormatException

Parameters

`nm`

A `String` representation of the value to be encapsulated by a `Short` object. If the string begins with # or `0x`, it is a radix 16 representation of the value. If the string begins with `0`, it is a radix 8 representation of the value. Otherwise, it is assumed to be a radix 10 representation of the value.

Returns

A `Short` object that encapsulates the given value.

Throws

`NumberFormatException`

If the `String` contains any non-digit characters other than a leading minus sign or the value represented by the `String` is less than `Short.MIN_VALUE` or greater than `Short.MAX_VALUE`.

Description

This method returns a `Short` object that encapsulates the given value.

parseByte

```
public static short parseShort(String s) throws NumberFormatException
```

Parameters

`s`

The `String` to be converted to a `short` value.

Returns

The numeric value of the `short` represented by the `String` object.

Throws

`NumberFormatException`

If the `String` does not contain a valid representation of a `short` or the value represented by the `String` is less than `Short.MIN_VALUE` or greater than `Short.MAX_VALUE`.

Description

This method returns the numeric value of the `short` represented by the contents of the given `String` object. The `String` must contain only decimal digits, except that the first character may be a minus sign.

`public static short parseShort(String s, int radix) throws NumberFormatException`

Parameters

`s`

The `String` to be converted to a short value.

`radix`

The radix used in interpreting the characters in the `String` as digits. This value must be in the range `Character.MIN_RADIX` to `Character.MAX_RADIX`. If radix is in the range 2 through 10, only characters for which the `Character.isDigit()` method returns `true` are considered to be valid digits. If `radix` is in the range 11 through 36, characters in the ranges `A' through `Z' and `a' through `z' are considered valid digits.

Returns

The numeric value of the `short` represented by the `String` object in the specified radix.

Throws

`NumberFormatException`

If the `String` does not contain a valid representation of a `short`, radix is not in the appropriate range, or the value represented by the `String` is less than `Short.MIN_VALUE` or greater than `Short.MAX_VALUE`.

Description

This method returns the numeric value of the `short` represented by the contents of the given `String` object in the specified radix. The `String` must contain only valid digits of the specified radix, except that the first character may be a minus sign. The digits are parsed in the specified radix to produce the numeric value.

toString

public String toString(short s)

Parameters

`s`

The `short` value to be converted to a string.

Returns

The `string` representation of the given value.

Description

This method returns a `String` object that contains the decimal representation of the given value.

This method returns a string that begins with `` `- ``' if the given value is negative. The rest of the string is a sequence of one or more of the characters `0', `1', `2', `3', `4', `5', `6', `7', `8', and `9'. This method returns "0" if its argument is `0`. Otherwise, the string returned by this method does not begin with "0" or "-0".

valueOf

public static Short valueOf(String s) throws NumberFormatException

Parameters

`s`

The string to be made into a `Short` object.

Returns

The `Short` object constructed from the string.

Throws

`NumberFormatException`

If the `String` does not contain a valid representation of a `short` or the value represented by the `String` is less than `Short.MIN_VALUE` or greater than `Short.MAX_VALUE`.

Description

Constructs a `Short` object with the value specified by the given string. The string should consist of one or more digit characters. The digit characters can be preceded by a single `` `- ``'. If the string contains any other characters, the method throws a `NumberFormatException`.

`public static Short valueOf(String s, int radix) throws NumberFormatException`

Parameters

`s`

The string to be made into a `Short` object.

`radix`

The radix used in converting the string to a value. This value must be in the range `Character.MIN_RADIX` to `Character.MAX_RADIX`.

Returns

The `Short` object constructed from the string.

Throws

`NumberFormatException`

If the `String` does not contain a valid representation of a `short`, `radix` is not in the appropriate range, or the value represented by the `String` is less than `Short.MIN_VALUE` or greater than `Short.MAX_VALUE`.

Description

Constructs a `Short` object with the value specified by the given string in the specified radix. The string should consist of one or more digit characters or characters in the range `A' to `Z' or `a' to `z' that are considered digits in the given radix. The digit characters can be preceded by a single `-' character. If the string contains any other characters, the method throws a `NumberFormatException`.

Instance Methods

byteValue

public byte byteValue()

Returns

The value of this object as a `byte`. The high order bits of the value are discarded.

Overrides

`Number.byteValue()`

Description

This method returns the value of this object as a `byte`.

doubleValue

public double doubleValue()

Returns

The value of this object as a `double`.

Overrides

`Number.doubleValue()`

Description

This method returns the value of this object as a `double`.

equals

public boolean equals(Object obj)

Parameters

`obj`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `obj` is an instance of `Short` and it contains the same value as the object this method is associated with.

floatValue

public float floatValue()

Returns

The value of this object as a `float`.

Overrides

`Number.floatValue()`

Description

This method returns the value of this object as a `float`.

hashCode

public int hashCode()

Returns

A hashcode based on the `short` value of the object.

Overrides

`Object.hashCode()`

Description

This method returns a hash code computed from the value of this object.

intValue

public int intValue()

Returns

The value of this object as an `int`.

Overrides

`Number.intValue()`

Description

This method returns the value of this object as an `int`.

longValue

public long longValue()

Returns

The value of this object as a `long`.

Overrides

`Number.longValue()`

Description

This method returns the value of this object as a `long`.

shortValue

public short shortValue()

Returns

The value of this object as a `short`.

Overrides

`Number.shortValue()`

Description

This method returns the value of this object as a `short`.

toString

public String toString()

Returns

The string representation of the value of this object.

Overrides

`Object.toString()`

Description

This method returns a `String` object that contains the decimal representation of the value of this object.

This method returns a string that begins with `` `- ``' if the value is negative. The rest of the string is a sequence of one or more of the characters `0', `1', `2', `3', `4', `5', `6', `7', `8', and `9'. This method returns "0" if the value of the object is `0`. Otherwise, the string returned by this method does not begin with "0" or "-0".

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`Byte`, `Character`, `Class`, `Double`, `Float`, `Integer`, `Long`, `Number`, `String`

PREVIOUS HOME NEXT
SecurityManager BOOK INDEX StackOverflowError

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

StackOverflowError

Name

StackOverflowError

Synopsis

Class Name:

`java.lang.StackOverflowError`

Superclass:

`java.lang.VirtualMachineError`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

A `StackOverflowError` is thrown when a stack overflow error occurs within the virtual machine.

Class Summary

```
public class java.lang.StackOverflowError
             extends java.lang.VirtualMachineError {
  // Constructors
  public StackOverflowError();
  public StackOverflowError(String s);
}
```

Constructors

StackOverflowError

public StackOverflowError()

Description

This constructor creates a `StackOverflowError` with no associated detail message.

public StackOverflowError(String s)<

Parameters

`s`

The detail message.

Description

This constructor creates a `StackOverflowError` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`

`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Error`, `Throwable`, `VirtualMachineError`

PREVIOUS	HOME	NEXT
Short	BOOK INDEX	String

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

String

Name

String

Synopsis

Class Name:

`java.lang.String`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

`java.io.Serializable`

Availability:

JDK 1.0 or later

Description

The `String` class represents sequences of characters. Once a `String` object is created, it is immutable. In other words, the sequence of characters that a `String` represents cannot be changed after it is created. The `StringBuffer` class, on the other hand, represents a sequence of characters that can be changed. `StringBuffer` objects are used to perform computations on `String` objects.

The `String` class includes a number of utility methods, such as methods for fetching individual characters or ranges of contiguous characters, for translating characters to upper- or lowercase, for searching strings, and for parsing numeric values in strings.

`String` literals are compiled into `String` objects. Where a `String` literal appears in an expression, the compiled code contains a `String` object. If `s` is declared as `String`, the following two expressions are identical:

```
s.equals("ABC")
"ABC".equals(s)
```

The string concatenation operator implicitly creates `String` objects.

Class Summary

```
public final class java.lang.String extends java.lang.Object {
    // Constructors
    public String();
    public String(byte[] bytes);                           // New in 1.1
    public String(byte[] bytes, String enc);               // New in 1.1
    public String(byte[] bytes, int offset, int length);  // New in 1.1
    public String(byte[] bytes, int offset,
                  int length, String enc);                 // New in 1.1
    public String(byte[] lowbytes, int hibyte);        // Deprecated in 1.1
    public String(byte[] lowbytes, int hibyte,
                  int offset, int count);              // Deprecated in 1.1
    public String(char[] value);
    public String(char[] value, int offset, int;
    public String(String value);
    public String(StringBuffer buffer);
    // Class Methods
    public static String copyValueOf(char data[]);
    public static String copyValueOf(char data[], int offset, int count);
    public static String valueOf(boolean b);
    public static String valueOf(char c);
    public static String valueOf(char[] data);
    public static String valueOf(char[] data, int offset, int count);
    public static String valueOf(double d);
    public static String valueOf(float f);
    public static String valueOf(int i);
    public static String valueOf(long l);
    public static String valueOf(Object obj);
    // Instance Methods
    public char charAt(int index);
    public int compareTo(String anotherString);
    public String concat(String str);
    public boolean endsWith(String suffix);
    public boolean equals(Object anObject);
    public boolean equalsIgnoreCase(String anotherString);
    public byte[] getBytes();                             // New in 1.1
    public byte[] getBytes(String enc);                   // New in 1.1
    public void getBytes(int srcBegin, int srcEnd,
                         byte[] dst, int dstBegin);     // Deprecated in 1.1
    public void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin);
    public int hashCode();
    public int indexOf(int ch);
    public int indexOf(int ch, int fromIndex);
    public int indexOf(String str);
    public int indexOf(String str, int fromIndex);
    public native String intern();
    public int lastIndexOf(int ch);
    public int lastIndexOf(int ch, int fromIndex);
    public int lastIndexOf(String str);
    public int lastIndexOf(String str, int fromIndex;
    public int length();
    public boolean regionMatches(boolean ignoreCase, int toffset,
                                 String other, int ooffset, int len);
    public boolean regionMatches(int toffset, String other,
                                 int ooffset, int len);
    public String replace(char oldChar, char newChar);
    public boolean startsWith(String prefix);
    public boolean startsWith(String prefix, int toffset);
    public String substring(int beginIndex);
    public String substring(int beginIndex, int endIndex);
    public char[] toCharArray();
    public String toLowerCase();
    public String toLowerCase(Locale locale);             // New in 1.1
    public String toString();
    public String toUpperCase();
    public String toUpperCase(Locale locale);             // New in 1.1
    public String trim();
}
```

Constructors

String

`public String()`

Description

> Creates a new `String` object that represents the empty string (i.e., a string with zero characters).

`public String(byte[] bytes)`

Availability

New as of JDK 1.1

Parameters

`bytes`

An array of `byte` values.

Description

Creates a new `String` object that represents the sequence of characters stored in the given `bytes` array. The bytes in the array are converted to characters using the system's default character encoding scheme.

`public String(byte[] bytes, String enc)`

Availability

New as of JDK 1.1

Parameters

`bytes`

An array of `byte` values.

`enc`

The name of an encoding scheme.

Throws

`UnsupportedEncodingException`

If `enc` is not a supported encoding scheme.

Description

Creates a new `String` object that represents the sequence of characters stored in the given `bytes` array. The bytes in the array are converted to characters using the specified character encoding scheme.

`public String(byte[] bytes, int offset, int length)`

Availability

New as of JDK 1.1

Parameters

`bytes`

An array of `byte` values.

`offset`

An offset into the array.

`length`

The number of bytes to be included.

Throws

`StringIndexOutOfBoundsException`

If `offset` or `length` indexes an element that is outside the bounds of the `bytes` array.

Description

Creates a new `String` object that represents the sequence of characters stored in the specified portion of the given `bytes` array. The bytes in the array are converted to characters using the system's default character encoding scheme.

`public String(byte[] bytes, int offset, int length, String enc)`

Availability

New as of JDK 1.1

Parameters

`bytes`

An array of `byte` values.

`offset`

An offset into the array.

`length`

The number of bytes to be included.

`enc`

The name of an encoding scheme.

Throws

`StringIndexOutOfBoundsException`

If `offset` or `length` indexes an element that is outside the bounds of the `bytes` array.

`UnsupportedEncodingException`

If `enc` is not a supported encoding scheme.

Description

Creates a new `String` object that represents the sequence of characters stored in the specified portion of the given `bytes` array. The bytes in the array are converted to characters using the specified character encoding scheme.

`public String(byte[] lowbytes, int hibyte)`

Availability

Deprecated as of JDK 1.1

Parameters

`lowbytes`

An array of `byte` values.

`hibyte`

The value to be put in the high-order byte of each 16-bit character.

Description

Creates a new `String` object that represents the sequence of characters stored in the given `lowbytes` array. The type of the array elements is `byte`, which is an 8-bit data type, so each element must be converted to a `char`, which is a 16-bit data type. The value of the `hibyte` argument is used to provide the value of the high-order byte when the `byte` values in the array are converted to `char` values.

More specifically, for each element `i` in the array `lowbytes`, the character at position `i` in the created `String` object is:

```
((hibyte & 0xff)<<8) | lowbytes[i]
```

This method is deprecated as of JDK 1.1 because it does not convert bytes into characters properly. You should instead use one of the constructors that takes a specific character encoding argument or that uses the default encoding.

`public String(byte[] lowbytes, int hibyte, int offset, int count)`

Availability

Deprecated as of JDK 1.1

Parameters

`lowbytes`

An array of `byte` values.

`hibyte`

The value to be put in the high-order byte of each 16-bit character.

`offset`

An offset into the array.

`count`

The number of bytes from the array to be included in the string.

Throws

`StringIndexOutOfBoundsException`

If `offset` or `count` indexes an element that is outside the bounds of the `lowbytes` array.

Description

Creates a new `String` object that represents the sequence of characters stored in the specified portion of the `lowbytes` array. That is, the portion of the array that starts at `offset` elements from the beginning of the array and is `count` elements long.

The type of the array elements is `byte`, which is an 8-bit data type, so each element must be converted to a `char`, which is a 16-bit data type. The value of the `hibyte` argument is used to provide the value of the high-order byte when the `byte` values in the array are converted to `char` values.

More specifically, for each element `i` in the array `lowbytes` that is included in the `String` object, the character at position `i` in the created `String` is:

```
 ((hibyte & 0xff)<<8) | lowbytes[I]
```

This method is deprecated as of JDK 1.1 because it does not convert bytes into characters properly. You should instead use one of the constructors that takes a specific character encoding argument or that uses the default encoding.

`public String(char[] value)`

Parameters

`value`

An array of `char` values.

Description

Creates a new `String` object that represents the sequence of characters stored in the given array.

`public String(char[] value, int offset, int count)`

Parameters

`value`

An array of `char` values.

`offset`

An offset into the array.

`count`

The number of characters from the array to be included in the string.

Throws

`StringIndexOutOfBoundsException`

If `offset` or `count` indexes an element that is outside the bounds of the `value` array.

Description

Creates a new `String` object that represents the sequence of characters stored in the specified portion of the given array. That is, the portion of the given array that starts at `offset` elements from the beginning of the array and is `count` elements long.

`public String(String value)`

Parameters

`value`

A `String` object.

Description

Creates a new `String` object that represents the same sequence of characters as the given `String` object.

`public String(StringBuffer value)`

Parameters

`value`

A `StringBuffer` object.

Description

Creates a new `String` object that represents the same sequence of characters as the given object.

Class Methods

copyValueOf

`public static String copyValueOf(char data[])`

Parameters

`data`

An array of `char` values.

Returns

A new `String` object that represents the sequence of characters stored in the given array.

Description

This method returns a new `String` object that represents the character sequence contained in the given array. The `String` object produced by this method is guaranteed not to refer to the given array, but instead to use a copy. Because the `String` object uses a copy of the array, subsequent changes to the array do not change the contents of this `String` object.

This method is now obsolete. The same result can be obtained using the `valueOf()` method that takes an array of `char` values.

`public static String copyValueOf(char data[], int offset, int count)`

Parameters

`data`

An array of `char` values.

`offset`

An offset into the array.

`count`

The number of characters from the array to be included in the string.

Returns

A new `String` object that represents the sequence of characters stored in the specified portion of the given array.

Throws

`StringIndexOutOfBoundsException`

If `offset` or `count` indexes an element that is outside the bounds of the `data` array.

Description

This method returns a new `String` object that represents the character sequence contained in the specified portion of the given array. That is, the portion of the given array that starts at `offset` elements from the beginning of the array and is `count` elements long. The `String` object produced by this method is guaranteed not to refer to the given array, but instead to use a copy. Because the `String` object uses a copy of the array, subsequent changes to the array do not change the contents of this `String` object.

This method is obsolete. The same result can be obtained by using the `valueOf()` method that takes an array of `char` values, an offset, and a count.

valueOf

`public static String valueOf(boolean b)`

Parameters

`b`

A `boolean` value.

Returns

A new `String` object that contains `'true'` if `b` is `true` or `'false'` if `b` is `false`.

Description

This method returns a string representation of a `boolean` value. In other words, it returns `'true'` if `b` is `true` or `'false'` if `b` is `false`.

`public static String valueOf(char c)`

Parameters

`c`

A `char` value.

Returns

A new `String` object that contains just the given character.

Description

This method returns a string representation of a `char` value. In other words, it returns a `String` object that contains just the given character.

`public static String valueOf(char[] data)`

Parameters

`data`

An array of `char` values.

Returns

A new `String` object that contains the sequence of characters stored in the given array.

Description

This method returns a string representation of an array of `char` values. In other words, it returns a `String` object that contains the sequence of characters stored in the given array.

`public static String valueOf(char[] data, int offset, int count)`

Parameters

`data`

An array of `char` values.

`offset`

An offset into the array.

`count`

The number of characters from the array to be included in the string.

Returns

A new `String` object that contains the sequence of characters stored in the specified portion of the given array.

Throws

`StringIndexOutOfBoundsException`

If `offset` or `count` indexes an element that is outside the bounds of the `data` array.

Description

This method returns a string representation of the specified portion of an array of char values. In other words, it returns a `String` object that contains the sequence of characters in the given array that starts `offset` elements from the beginning of the array and is `count` elements long.

`public static String valueOf(double d)`

Parameters

`d`

A `double` value.

Returns

A new `String` object that contains a string representation of the given `double` value.

Description

This method returns a string representation of a `double` value. In other words, it returns the `String` object returned by `Double.toString(d)`.

`public static String valueOf(float f)`

Parameters

`f`

A `float` value.

Returns

A new `String` object that contains a string representation of the given `float` value.

Description

This method returns a string representation of a `float` value. In other words, it returns the `String` object returned by `Float.toString(f)`.

`public static String valueOf(int i)`

Parameters

`i`

An `int` value.

Returns

A new `String` object that contains a string representation of the given `int` value.

Description

This method returns a string representation of an `int` value. In other words, it returns the `String` object returned by `Integer.toString(i)`.

`public static String valueOf(long l)`

Parameters

`l`

A `long` value.

Returns

A new `String` object that contains a string representation of the given `long` value.

Description

This method returns a string representation of a `long` value. In other words, it returns the `String` object returned by `Long.toString(l)`.

`public static String valueOf (Object obj)`

Parameters

`obj`

A reference to an object.

Returns

A new `String` that contains a string representation of the given object.

Description

This method returns a string representation of the given object. If `obj` is `null`, the method returns`'null'`. Otherwise, the method returns the `String` object returned by the `toString()` method of the object.

Instance Methods

charAt

`public char charAt(int index)`

Parameters

`index`

An index into the string.

Returns

The character at the specified position in this string.

Throws

`StringIndexOutOfBoundsException`

If `index` is less than zero or greater than or equal to the length of the string.

Description

This method returns the character at the specified position in the `String` object; the first character in the string is at position `0`.

compareTo

`public int compareTo(String anotherString)`

Parameters

`anotherString`

The `String` object to be compared.

Returns

A positive value if this string is greater than `anotherString`, `0` if the two strings are the same, or a negative value if this string is less than `anotherString`.

Description

This method lexicographically compares this `String` object to `anotherString`.

Here is how the comparison works: the two `String` objects are compared character-by-character, starting at index `0` and continuing until a position is found in which the two strings contain different characters or until all of the characters in the shorter string have been compared. If the characters at `k` are different, the method returns:

```
this.charAt(k)-anotherString.charAt(k)
```

Otherwise, the comparison is based on the lengths of the strings and the method returns:

```
this.length()-anotherString.length()
```

concat

`public String concat(String str)`

Parameters

`str`

The `String` object to be concatenated.

Returns

A new `String` object that contains the character sequences of this string and `str` concatenated together.

Description

This method returns a new `String` object that concatenates the characters from the argument string `str` onto the characters from this `String` object. Although this is a good way to concatenate two strings, concatenating more than two strings can be done more efficiently using a `StringBuffer` object.

endsWith

`public boolean endsWith(String suffix)`

Parameters

`suffix`

The `String` object suffix to be tested.

Returns

`true` if this string ends with the sequence of characters specified by `suffix`; otherwise `false`.

Description

This method determines whether or not this `String` object ends with the specified `suffix`.

equals

`public boolean equals(Object anObject)`

Parameters

`anObject`

The `Object` to be compared.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `anObject` is an instance of `String` and it contains the same sequence of characters as this `String` object.

Note the difference between this method and the `==` operator, which only returns `true` if both of its arguments are references to the same object.

equalsIgnoreCase

`public boolean equalsIgnoreCase(String anotherString)`

Parameters

`anotherString`

The `String` object to be compared.

Returns

`true` if the strings are equal, ignoring case; otherwise `false`.

Description

This method determines whether or not this `String` object contains the same sequence of characters, ignoring case, as `anotherString`. More specifically, corresponding characters in the two strings are considered equal if any of the following conditions are true:

- The two characters compare as equal using the `==` operator.
- The `Character.toUppercase()` method returns the same result for both characters.
- The `Character.toLowercase()` method returns the same result for both characters.

getBytes

`public byte[] getBytes()`

Availability

New as of JDK 1.1

Returns

A `byte` array that contains the characters of this `String`.

Description

This method converts the characters in this `String` object to an array of `byte` values. The characters in the string are converted to bytes using the system's default character encoding scheme.

`public byte[] getBytes(String enc)`

Availability

New as of JDK 1.1

Parameters

`enc`

The name of an encoding scheme.

Returns

A `byte` array that contains the characters of this `String`.

Throws

`UnsupportedEncodingException`

If `enc` is not a supported encoding scheme.

Description

This method converts the characters in this `String` object to an array of `byte` values. The characters in the string are converted to bytes using the specified character encoding scheme.

`public void getBytes(int srcBegin, int srcEnd, byte[] dst, int dstBegin)`

Availability

Deprecated as of JDK 1.1

Parameters

`srcBegin`

The index of the first character to be copied.

`srcEnd`

The index after the last character to be copied.

`dst`

The destination `byte` array.

`dstBegin`

An offset into the destination array.

Throws

`StringIndexOutOfBoundsException`

If `srcBegin`, `srcEnd`, or `dstBegin` is out of range.

Description

This method copies the low-order byte of each character in the specified range of this `String` object to the given array of `byte` values. More specifically, the first character to be copied is at index `srcBegin`; the last character to be copied is at index `srcEnd-1`. The low-order bytes of these characters are copied into `dst`, starting at index `dstBegin` and ending at index:

```
dstBegin + (srcEnd-srcBegin) - 1
```

This method is deprecated as of JDK 1.1 because it does not convert characters into bytes properly. You should instead use the `getBytes()` method that takes a specific character encoding argument or the one that uses the default encoding.

getChars

```
public void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)
```

Parameters

`srcBegin`

The index of the first character to be copied.

`srcEnd`

The index after the last character to be copied.

`dst`

The destination `char` array.

`dstBegin`

An offset into the destination array.

Throws

`StringIndexOutOfBoundsException`

If `srcBegin`, `srcEnd`, or `dstBegin` is out of range.

Description

This method copies each character in the specified range of this `String` object to the given array of `char` values. More specifically, the first character to be copied is at index `srcBegin`; the last character to be copied is at index `srcEnd-1`. These characters are copied into `dst`, starting at index `dstBegin` and ending at index:

```
dstBegin + (srcEnd-srcBegin) - 1
```

hashCode

`public int hashCode()`

Returns

A hashcode based on the sequence of characters in this string.

Overrides

`Object.hashCode()`

Description

This method returns a hashcode based on the sequence of characters this `String` object represents.

More specifically, one of two algorithms is used to compute a hash code for the string, depending on its length. If *n* is the length of the string and *S_i* is the character at position *i* in the string, then if *n* = 15 the method returns:

Mathematical Equation

If *n* > 15, the method returns:

Mathematical Equation

indexOf

`public int indexOf(int ch)`

Parameters

`ch`

A `char` value.

Returns

The index of the first occurrence of the given character in this string or `-1` if the character does not occur.

Description

This method returns the index of the first occurrence of the given character in this `String` object. If there is no such occurrence, the method returns the value `-1`.

`public int indexOf(int ch, int fromIndex)`

Parameters

`ch`

A `char` value.

`fromIndex`

The index where the search is to begin.

Returns

The index of the first occurrence of the given character in this string after `fromIndex` or `-1` if the character does not occur.

Description

This method returns the index of the first occurrence of the given character in this `String` object after ignoring the first `fromIndex` characters. If there is no such occurrence, the method returns the value `-1`.

`public int indexOf(String str)`

Parameters

`str`

A `String` object.

Returns

The index of the first occurrence of `str` in this string or `-1` if the substring does not occur.

Description

This method returns the index of the first character of the first occurrence of the substring `str` in this `String` object. If there is no such occurrence, the method returns the value `-1`.

`public int indexOf(String str, int fromIndex)`

Parameters

`str`

A `String` object.

`fromIndex`

The index where the search is to begin.

Returns

The index of the first occurrence of `str` in this string after `fromIndex` or `-1` if the substring does not occur.

Description

This method returns the index of the first character of the first occurrence of the substring `str` in this `String` object after ignoring the first `fromIndex` characters. If there is no such occurrence, the method returns the value `-1`.

intern

`public native String intern()`

Returns

A `String` object that is guaranteed to be the same object for every `String` that contains the same character sequence.

Description

This method returns a canonical representation for this `String` object. The returned `String` object is guaranteed to be the same `String` object for every `String` object that contains the same character sequence. In other words, if:

```
s1.equals(s2)
```

then:

```
s1.intern() == s2.intern()
```

The `intern`() method is used by the Java environment to ensure that `String` literals and constant-value `String` expressions that contain the same sequence of characters are all represented by a single `String` object.

lastIndexOf

`public int lastIndexOf(int ch)`

Parameters

`ch`

A `char` value.

Returns

The index of the last occurrence of the given character in this string or `-1` if the character does not occur.

Description

This method returns the index of the last occurrence of the given character in this `String` object. If there is no such occurrence, the method returns the value `-1`.

`public int lastIndexOf(int ch, int fromIndex)`

Parameters

`ch`

A `char` value.

`fromIndex`

The index where the search is to begin.

Returns

The index of the last occurrence of the given character in this string after `fromIndex` or `-1` if the character does not occur.

Description

This method returns the index of the last occurrence of the given character in this `String` object after ignoring the first `fromIndex` characters. If there is no such occurrence, the method returns the value `-1`.

`public int lastIndexOf(String str)`

Parameters

`str`

A `String` object.

Returns

The index of the last occurrence of `str` in this string or `-1` if the substring does not occur.

Description

This method returns the index of the first character of the last occurrence of the substring `str` in this `String` object. If there is no such occurrence, the method returns the value `-1`.

`public int lastIndexOf(String str, int fromIndex)`

Parameters

`str`

A `String` object.

`fromIndex`

The index where the search is to begin.

Returns

The index of the last occurrence of `str` in this string after `fromIndex` or `-1` if the substring does not occur.

Description

This method returns the index of the first character of the last occurrence of the substring `str` in this `String` object after ignoring the first `fromIndex` characters. If there is no such occurrence, the method returns the value `-1`.

length

`public int length()`

Returns

The length of the character sequence represented by this string.

Description

This method returns the number of characters in the character sequence represented by this `String` object.

regionMatches

`public boolean regionMatches(int toffset, String other, int ooffset, int len)`

Parameters

`toffset`

The index of the first character in this string.

`other`

The `String` object to be used in the comparison.

`ooffset`

The index of the first character in `other`.

`len`

The length of the sub-sequences to be compared.

Returns

`true` if the sub-sequences are identical; otherwise `false`.

Description

This method determines whether or not the specified sub-sequences in this `String` object and `other` are identical. The method returns false if `toffset` is negative, if `ooffset` is negative, if `toffset+len` is greater than the length of this string, or if `ooffset+len` is greater than the length of `other`. Otherwise, the method returns `true` if for all nonnegative integers `k` less than `len` it is true that:

```
this.charAt(toffset+k) == other.charAt(ooffset+k)
```

`public boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)`

Parameters

`ignoreCase`

A `boolean` value that indicates whether case should be ignored.

`toffset`

The index of the first character in this string.

`other`

The `String` object to be used in the comparison.

`ooffset`

The index of the first character in `other`.

`len`

The length of the sub-sequences to be compared.

Returns

`true` if the sub-sequences are identical; otherwise `false`. The `ignoreCase` argument controls whether or not case is ignored in the comparison.

Description

This method determines whether or not the specified sub-sequences in this `String` object and `other` are identical. The method returns false if `toffset` is negative, if `ooffset` is negative, if `toffset+len` is greater than the length of this string, or if `ooffset+len` is greater than the length of `other`. Otherwise, if `ignoreCase` is `false`, the method returns `true` if for all nonnegative integers `k` less than `len` it is true that:

```
this.charAt(toffset+k) == other.charAt(ooffset+k)
```

If `ignoreCase` is `true`, the method returns `true` if for all nonnegative integers `k` less than `len` it is true that:

```
Character.toLowerCase(this.charAt(toffset+k))
== Character.toLowerCase(other.charAt(ooffset+k))
```

or:

```
Character.toUpperCase(this.charAt(toffset+k))
== Character.toUpperCase(other.charAt(ooffset+k))
```

replace

`public String replace(char oldChar, char newChar)`

Parameters

`oldChar`

The character to be replaced.

`newChar`

The replacement character.

Returns

A new `String` object that results from replacing every occurrence of `oldChar` in the string with `newChar`.

Description

This method returns a new `String` object that results from replacing every occurrence of `oldChar` in this `String` object with `newChar`. If there are no occurrences of `oldChar`, the method simply returns this `String` object.

startsWith

`public boolean startsWith(String prefix)`

Parameters

`prefix`

The `String` object prefix to be tested.

Returns

`true` if this string begins with the sequence of characters specified by `prefix`; otherwise `false`.

Description

This method determines whether or not this `String` object begins with the specified `prefix`.

`public boolean startsWith(String prefix, int toffset)`

Parameters

`prefix`

The `String` object prefix to be tested.

`toffset`

The index where the search is to begin.

Returns

`true` if this string contains the sequence of characters specified by `prefix` starting at the index `toffset`; otherwise `false`.

Description

This method determines whether or not this `String` object contains the specified `prefix` at the index specified by `toffset`.

substring

`public String substring(int beginIndex)`

Parameters

`beginIndex`

The index of the first character in the substring.

Returns

A new `String` object that contains the sub-sequence of this string that starts at `beginIndex` and extends to the end of the string.

Throws

`StringIndexOutOfBoundsException`

If `beginIndex` is less than zero or greater than or equal to the length of the string.

Description

This method returns a new `String` object that represents a sub-sequence of this `String` object. The sub-sequence consists of the characters starting at `beginIndex` and extending through the end of this `String` object.

`public String substring(int beginIndex, int endIndex)`

Parameters

`beginIndex`

The index of the first character in the substring.

`endIndex`

The index after the last character in the substring.

Returns

A new `String` object that contains the sub-sequence of this string that starts at `beginIndex` and extends to the character at `endindex-1`.

Throws

`StringIndexOutOfBoundsException`

If `beginIndex` or `endIndex` is less than zero or greater than or equal to the length of the string.

Description

This method returns a new `String` object that represents a sub-sequence of this `String` object. The sub-sequence consists of the characters starting at `beginIndex` and extending through `endIndex-1` of this `String` object.

toCharArray

`public char[] toCharArray()`

Returns

A new `char` array that contains the same sequence of characters as this string.

Description

This method returns a new `char` array that contains the same sequence of characters as this `String`object. The length of the array is the same as the length of this `String` object.

toLowerCase

`public String toLowerCase()`

Returns

A new `String` object that contains the characters of this string converted to lowercase.

Description

This method returns a new `String` that represents a character sequence of the same length as this `String` object, but with each character replaced by its lowercase equivalent if it has one. If no character in the string has a lowercase equivalent, the method returns this `String` object.

`public String toLowerCase(Locale locale)`

Availability

New as of JDK 1.1

Parameters

`locale`

The `Locale` to use.

Returns

A new `String` object that contains the characters of this string converted to lowercase using the rules of the specified locale.

Description

This method returns a new `String` that represents a character sequence of the same length as this `String` object, but with each character replaced by its lowercase equivalent if it has one according to the rules of the specified locale. If no character in the string has a lowercase equivalent, the method returns this `String` object.

toString

`public String toString()`

Returns

This `String` object.

Overrides

`Object.toString()`

Description

This method returns this `String` object.

toUpperCase

`public String toUpperCase()`

Returns

A new `String` object that contains the characters of this string converted to uppercase.

Description

This method returns a new `String` that represents a character sequence of the same length as this `String` object, but with each character replaced by its uppercase equivalent if it has one. If no character in the string has an uppercase equivalent, the method returns this `String` object.

`public String toUpperCase(Locale locale)`

Availability

New as of JDK 1.1

Parameters

`locale`

The `Locale` to use.

Returns

A new `String` object that contains the characters of this string converted to uppercase using the rules of the specified locale.

Description

This method returns a new `String` that represents a character sequence of the same length as this `String` object, but with each character replaced by its uppercase equivalent if it has one according to the rules of the specified locale. If no character in the string has an uppercase equivalent, the method returns this `String` object.

trim

```
public String trim()
```

Returns

> A new `String` object that represents the same character sequence as this string, but with leading and trailing whitespace and control characters removed.

Description

> If the first and last character in this `String` object are greater than `'\u0020'` (the space character), the method returns this `String` object. Otherwise, the method returns a new `String` object that contains the same character sequence as this `String` object, but with leading and trailing characters that are less than `'\u0020''` removed.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`Class`, `Character`, `Double`, `Float`, `Integer`, `Locale`, `Long`, `Object`, `StringBuffer`, `StringIndexOutOfBoundsException`, `UnsupportedEncodingException`

PREVIOUS
StackOverflowError

HOME
BOOK INDEX

NEXT
StringBuffer

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

StringBuffer

Name

StringBuffer

Synopsis

Class Name:

`java.lang.StringBuffer`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

`java.io.Serializable`

Availability:

JDK 1.0 or later

Description

The `StringBuffer` class represents a variable-length sequence of characters. `StringBuffer` objects are used in computations that involve creating new `String` objects. The `StringBuffer` class provides a number of utility methods for working with `StringBuffer` objects, including `append()` and `insert()` methods that add characters to a `StringBuffer` and methods that fetch the contents of `StringBuffer` objects.

When a `StringBuffer` object is created, the constructor determines the initial contents and capacity of the `StringBuffer`. The capacity of a `StringBuffer` is the number of characters that its internal data structure can hold. This is distinct from the length of the contents of a `StringBuffer`, which is the number of characters that are actually stored in the `StringBuffer` object. The capacity of a `StringBuffer` can vary. When a `StringBuffer` object is asked to hold more characters than its current capacity allows, the `StringBuffer` enlarges its internal data structure. However, it is more costly in terms of execution time and memory when a `StringBuffer` has to repeatedly increase its capacity than when a `StringBuffer` object is created with sufficient capacity.

Because the intended use of `StringBuffer` objects involves modifying their contents, all methods of the `StringBuffer` class that modify `StringBuffer` objects are `synchronized`. This means that is it safe for multiple threads to try to modify a `StringBuffer` object at the same time.

`StringBuffer` objects are used implicitly by the string concatenation operator. Consider the following code:

```
String s, s1, s2;
s = s1 + s2;
```

To compute the string concatenation, the Java compiler generates code like:

```
s = new StringBuffer().append(s1).append(s2).toString();
```

Class Summary

```
public class java.lang.StringBuffer extends java.lang.Object {
    // Constructors
    public StringBuffer();
    public StringBuffer(int length);
    public StringBuffer(String str);
    // Instance Methods
    public StringBuffer append(boolean b);
    public synchronized StringBuffer append(char c);
    public synchronized StringBuffer append(char[] str);
    public synchronized StringBuffer append(char[] str, int offset, int len);
    public StringBuffer append(double d);
    public StringBuffer append(float f);
    public StringBuffer append(int i);
    public StringBuffer append(long l);
    public synchronized StringBuffer append(Object obj);
    public synchronized StringBuffer append(String str);
    public int capacity();
    public synchronized char charAt(int index);
    public synchronized void ensureCapacity(int minimumCapacity);
    public synchronized void getChars(int srcBegin, int srcEnd,
                             char[] dst, int dstBegin);
    public StringBuffer insert(int offset, boolean b);
    public synchronized StringBuffer insert(int offset, char c);
    public synchronized StringBuffer insert(int offset, char[] str);
    public StringBuffer insert(int offset, double d);
    public StringBuffer insert(int offset, float f);
    public StringBuffer insert(int offset, int i);
    public StringBuffer insert(int offset, long l);
    public synchronized StringBuffer insert(int offset, Object obj);
    public synchronized StringBuffer insert(int offset, String str);
    public int length();
    public synchronized StringBuffer reverse();
    public synchronized void setCharAt(int index, char ch);
    public synchronized void setLength(int newLength);
    public String toString();
}
```

Constructors

StringBuffer

`public StringBuffer()`

Description

Creates a `StringBuffer` object that does not contain any characters and has a capacity of 16 characters.

`public StringBuffer(int capacity)`

Parameters

`capacity`

The initial capacity of this `StringBufffer` object.

Throws

`NegativeArraySizeException`

If `capacity` is negative.

Description

Creates a `StringBuffer` object that does not contain any characters and has the specified capacity.

`public StringBuffer(String str)`

Parameters

`str`

A `String` object.

Description

Creates a `StringBuffer` object that contains the same sequence of characters as the given `String` object and has a capacity 16 greater than the length of the `String`.

Instance Methods

append

`public StringBuffer append(boolean b)`

Parameters

`b`

A `boolean` value.

Returns

This `StringBuffer` object.

Description

This method appends either `"true"` or `"false"` to the end of the sequence of characters stored in ths `StringBuffer` object, depending on the value of `b`.

`public synchronized StringBuffer append(char c)`

Parameters

`c`

A `char` value.

Returns

This `StringBuffer` object.

Description

This method appends the given character to the end of the sequence of characters stored in this `StringBuffer` object.

`public synchronized StringBuffer append(char str[])`

Parameters

`str`

An array of `char` values.

Returns

This `StringBuffer` object.

Description

This method appends the characters in the given array to the end of the sequence of characters stored in this `StringBuffer` object.

`public synchronized StringBuffer append(char str[], int offset, int len)`

Parameters

`str`

An array of `char` values.

`offset`

An offset into the array.

`len`

The number of characters from the array to be appended.

Returns

This `StringBuffer` object.

Throws

`StringIndexOutOfBoundsException`

If `offset` or `len` are out of range.

Description

This method appends the specified portion of the given array to the end of the character sequence stored in this `StringBuffer` object. The portion of the array that is appended starts `offset` elements from the beginning of the array and is `len` elements long.

`public StringBuffer append(double d)`

Parameters

`d`

A `double` value.

Returns

This `StringBuffer` object.

Description

This method converts the given `double` value to a string using `Double.toString(d)` and appends the resulting string to the end of the sequence of characters stored in this `StringBuffer` object.

`public StringBuffer append(float f)`

Parameters

`f`

A `float` value.

Returns

This `StringBuffer` object.

Description

This method converts the given `float` value to a string using `Float.toString(f)` and appends the resulting string to the end of the sequence of characters stored in this `StringBuffer` object.

`public StringBuffer append(int i)`

Parameters

`i`

An `int` value.

Returns

This `StringBuffer` object.

Description

This method converts the given `int` value to a string using `Integer.toString(i)` and appends the resulting string to the end of the sequence of characters stored in this `StringBuffer` object.

`public StringBuffer append(long l)`

Parameters

`l`

A `long` value.

Returns

This `StringBuffer` object.

Description

This method converts the given `long` value to a string using `Long.toString(l)` and appends the resulting string to the end of the sequence of characters stored in this `StringBuffer` object.

`public synchronized StringBuffer append(Object obj)`

Parameters

`obj`

A reference to an object.

Returns

This `StringBuffer` object.

Description

This method gets the string representation of the given object by calling `String.valueOf(obj)` and appends the resulting string to the end of the character sequence stored in this `StringBuffer` object.

`public synchronized StringBuffer append(String str)`

Parameters

`str`

A `String` object.

Returns

This `StringBuffer` object.

Description

This method appends the sequence of characters represented by the given `String` to the characters in this `StringBuffer` object. If `str` is `null`, the string `"null"` is appended.

capacity

`public int capacity()`

Returns

The capacity of this `StringBuffer` object.

Description

This method returns the current capacity of this object. The capacity of a `StringBuffer` object is the number of characters that its internal data structure can hold. A `StringBuffer` object automatically increases its capacity when it is asked to hold more characters than its current capacity allows.

charAt

`public synchronized char charAt(int index)`

Parameters

`index`

An index into the `StringBuffer`.

Returns

The character stored at the specified position in this `StringBuffer` object.

Throws

`StringIndexOutOfBoundsException`

If `index` is less than zero or greater than or equal to the length of the `StringBuffer` object.

Description

This method returns the character at the specified position in the `StringBuffer` object. The first character in the `StringBuffer` is at index `0`.

ensureCapacity

```
public synchronized void ensureCapacity(int minimumCapacity)
```

Parameters

`minimumCapacity`

The minimum desired capacity.

Description

This method ensures that the capacity of this `StringBuffer` object is at least the specified number of characters. If necessary, the capacity of this object is increased to the greater of `minimumCapacity` or double its current capacity plus two.

It is more efficient to ensure that the capacity of a `StringBuffer` object is sufficient to hold all of the additions that will be made to its contents, rather than let the `StringBuffer` increase its capacity in multiple increments.

getChars

```
 public synchronized void getChars(int srcBegin, int srcEnd, char dst[], int
```

`dstBegin)`

Parameters

`srcBegin`

The index of the first character to be copied.

`srcEnd`

The index after the last character to be copied.

`dst`

The destination `char` array.

`dstBegin`

An offset into the destination array.

Throws

`StringIndexOutOfBoundsException`

If `srcBegin`, `srcEnd`, or `dstBegin` is out of range.

Description

This method copies each character in the specified range of this `StringBuffer` object to the given array of `char` values. More specifically, the first character to be copied is at index `srcBegin`; the last character to be copied is at index `srcEnd-1`.

These characters are copied into `dst`, starting at index `dstBegin` and ending at index:

```
dstBegin + (srcEnd-srcBegin) - 1
```

insert

public StringBuffer insert(int offset, boolean b)

Parameters

`offset`

An offset into the `StringBuffer`.

`b`

A `boolean` value.

Returns

This `StringBuffer` object.

Throws

`StringIndexOutOfBoundsException`

If `offset` is out of range.

Description

This method inserts either `"true"` or `"false"` into the sequence of characters stored in this `StringBuffer` object, depending on the value of `b`. The string is inserted at a position `offset` characters from the beginning of the sequence. If `offset` is `0`, the string is inserted before the first character in the `StringBuffer`.

`public synchronized StringBuffer insert(int offset, char c)`

Parameters

`offset`

An offset into the `StringBuffer`.

`c`

A `char` value.

Returns

This `StringBuffer` object.

Throws

`StringIndexOutOfBoundsException`

If `offset` is less than zero or greater than or equal to the length of the `StringBuffer` object.

Description

This method inserts the given character into the sequence of characters stored in this `StringBuffer` object. The character is inserted at a position `offset` characters from the beginning of the sequence. If `offset` is `0`, the character is inserted before the first character in the `StringBuffer`.

public synchronized StringBuffer insert(int offset, char str[])

Parameters

`offset`

An offset into the `StringBuffer`.

`str`

An array of `char` values.

Returns

This `StringBuffer` object.

Throws

`StringIndexOutOfBoundsException`

If `offset` is less than zero or greater than or equal to the length of the `StringBuffer` object.

Description

This method inserts the characters in the given array into the sequence of characters stored in this `StringBuffer` object. The characters are inserted at a position `offset` characters from the beginning of the sequence. If `offset` is `0`, the characters are inserted before the first character in the `StringBuffer`.

`public StringBuffer insert(int offset, double d)`

Parameters

`offset`

An offset into the `StringBuffer`.

`d`

A `double` value.

Returns

This `StringBuffer` object.

Throws

`StringIndexOutOfBoundsException`

If `offset` is less than zero or greater than or equal to the length of the `StringBuffer` object.

Description

This method converts the given `double` value to a string using `Double.toString(d)` and inserts the resulting string into the sequence of characters stored in this `StringBuffer` object. The string is inserted at a position `offset` characters from the beginning of the sequence. If `offset` is `0`, the string is inserted before the first character in the `StringBuffer`.

`public StringBuffer insert(int offset, float f)`

Parameters

`offset`

An offset into the `StringBuffer`.

`f`

A `float` value.

Returns

This `StringBuffer` object.

Throws

`StringIndexOutOfBoundsException`

If `offset` is less than zero or greater than or equal to the length of the `StringBuffer` object.

Description

This method converts the given `float` value to a string using `Float.toString(f)` and inserts the resulting string into the sequence of characters stored in this `StringBuffer` object. The string is inserted at a position `offset` characters from the beginning of the sequence. If `offset` is `0`, the string is inserted before the first character in the `StringBuffer`.

`public StringBuffer insert(int offset, int i)`

Parameters

`offset`

An offset into the `StringBuffer`.

`i`

An `int` value.

Returns

This `StringBuffer` object.

Throws

`StringIndexOutOfBoundsException`

If `offset` is less than zero or greater than or equal to the length of the `StringBuffer` object.

Description

This method converts the given `int` value to a string using `Integer.toString(i)` and inserts the resulting string into the sequence of characters stored in this `StringBuffer` object. The string is inserted at a position `offset` characters from the beginning of the sequence. If `offset` is `0`, the string is inserted before the first character in the `StringBuffer`.

`public StringBuffer insert(int offset, long l)`

Parameters

`offset`

An offset into the `StringBuffer`.

`l`

A `long` value.

Returns

This `StringBuffer` object.

Throws

`StringIndexOutOfBoundsException`

If `offset` is less than zero or greater than or equal to the length of the `StringBuffer` object.

Description

This method converts the given `long` value to a string using `Long.toString(l)` and inserts the resulting string into the sequence of characters stored in this `StringBuffer` object. The string is inserted at a position `offset` characters from the beginning of the sequence. If `offset` is `0`, the string is inserted before the first character in the `StringBuffer`.

public synchronized StringBuffer insert(int offset, Object obj)

Parameters

`offset`

An offset into the `StringBuffer`.

`obj`

A reference to an object.

Returns

This `StringBuffer` object.

Throws

`StringIndexOutOfBoundsException`

If `offset` is less than zero or greater than or equal to the length of the `StringBuffer` object.

Description

This method gets the string representation of the given object by calling `String.valueOf(obj)` and inserts the resulting string into the sequence of characters stored in this `StringBuffer` object. The string is inserted at a position `offset` characters from the beginning of the sequence. If `offset` is `0`, the string is inserted before the first character in the `StringBuffer`.

public synchronized StringBuffer insert(int offset, String str)

Parameters

`offset`

An offset into the `StringBuffer`.

`str`

A `String` object.

Returns

This `StringBuffer` object.

Throws

`StringIndexOutOfBoundsException`

If `offset` is less than zero or greater than or equal to the length of the `StringBuffer` object.

Description

This method inserts the sequence of characters represented by the given `String` into the sequence of characters stored in this `StringBuffer` object. If `str` is `null`, the string `"null"` is inserted. The string is inserted at a position `offset` characters from the beginning of the sequence. If `offset` is `0`, the string is inserted before the first character in the `StringBuffer`.

length

`public int length()`

Returns

The number of characters stored in this `StringBuffer` object.

Description

This method returns the number of characters stored in this `StringBuffer` object. The length is distinct from the capacity of a `StringBuffer`, which is the number of characters that its internal data structure can hold.

reverse

`public synchronized StringBuffer reverse()`

Returns

> This `StringBuffer` object.

Description

> This method reverses the sequence of characters stored in this `StringBuffer` object.

setCharAt

`public synchronized void setCharAt(int index, char ch)`

Parameters

> `index`
>
> > The index of the character to be set.
>
> `ch`
>
> > A `char` value.

Throws

> `StringIndexOutOfBoundsException`
>
> > If `index` is less than zero or greater than or equal to the length of the `StringBuffer` object.

Description

> This method modifies the character located `index` characters from the beginning of the sequence of characters stored in this `StringBuffer` object. The current character at this position is replaced by the character `ch`.

setLength

`public synchronized void setLength(int newLength)`

Parameters

> `newLength`

The new length for this `StringBuffer`.

Throws

`StringIndexOutOfBoundsException`

If `index` is less than zero.

Description

This method sets the length of the sequence of characters stored in this `StringBuffer` object. If the length is set to be less than the current length, characters are lost from the end of the character sequence. If the length is set to be more than the current length, NUL characters (`\u0000`) are added to the end of the character sequence.

toString

`public String toString()`

Returns

A new `String` object that represents the same sequence of characters as the sequence of characters stored in this `StringBuffer` object.

Overrides

`Object.toString()`

Description

This method returns a new `String` object that represents the same sequence of characters as the sequence of characters stored in this `StringBuffer` object. Note that any subsequent changes to the contents of this `StringBuffer` object do not affect the contents of the `String` object created by this method.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`

`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`Class`, `Double`, `Float`, `Integer`, `Long`, `Object`, `String`, `StringIndexOutOfBoundsException`

PREVIOUS HOME NEXT
String BOOK INDEX StringIndexOutOfBoundsException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

StringIndexOutOfBoundsException

Name

StringIndexOutOfBoundsException

Synopsis

Class Name:

`java.lang.StringIndexOutOfBoundsException`

Superclass:

`java.lang.IndexOutOfBoundsException`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

A `StringIndexOutOfBoundsException` is thrown when a `String` or `StringBuffer` object detects an out-of-range index. An out-of-range index occurs when the index is less than zero, or greater than or equal to the length of the string.

Class Summary

```
public class java.lang.StringIndexOutOfBoundsException
             extends java.lang.IndexOutOfBoundsException {
  // Constructors
  public StringIndexOutOfBoundsException();
  public StringIndexOutOfBoundsException(int index);
  public StringIndexOutOfBoundsException(String s);
}
```

Constructors

StringIndexOutOfBoundsException

public StringIndexOutOfBoundsException()

Description

> This constructor creates a `StringIndexOutOfBoundsException` with no associated detail message.

public StringIndexOutOfBoundsException(int index)

Parameters

> `index`
>
> > The index value that was out of bounds

Description

> This constructor creates an `StringIndexOutOfBoundsException` with an associated message that mentions the specified index.

public StringIndexOutOfBoundsException(String s)

Parameters

> `s`

The detail message.

Description

This constructor creates a `StringIndexOutOfBoundsException` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Exception`, `IndexOutOfBoundsException`, `RuntimeException`, `Throwable`

PREVIOUS
StringBuffer

HOME
BOOK INDEX

NEXT
System

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

PREVIOUS

NEXT

System

Name

System

Synopsis

Class Name:

  `java.lang.System`

Superclass:

  `java.lang.Object`

Immediate Subclasses:

  None

Interfaces Implemented:

  None

Availability:

  JDK 1.0 or later

Description

The `System` class provides access to various information about the operating system environment in which a program is running. For example, the `System` class defines variables that allow access to the standard I/O streams and methods that allow a program to run the garbage collector and stop the Java virtual machine.

All of the variables and methods in the `System` class are `static`. In other words, it is not necessary to create an instance of the `System` class in order to use its variables and methods. In fact, the `System` class does not define any `public` constructors, so it cannot be instantiated.

The `System` class supports the concept of system properties that can be queried and set. The following properties are guaranteed always to be defined:

Property Name	**Description**
`file.encoding`	The character encoding for the default locale (Java 1.1 only)
`file.encoding.pkg`	The package that contains converters between local encodings and Unicode (Java 1.1 only)
`file.separator`	File separator (`'/'` on UNIX, `' \'` on Windows)
`java.class.path`	The class path
`java.class.version`	Java class version number
`java.compiler`	The just-in-time compiler to use, if any (Java 1.1 only)
`java.home`	Java installation directory
`java.vendor`	Java vendor-specific string
`java.vendor.url`	Java vendor URL
`java.version`	Java version number
`line.separator`	Line separator(`' \n'` on UNIX, `' \r\n'` on Windows)
`os.arch`	Operating system architecture
`os.name`	Operating system name
`os.version`	Operating system version
`path.separator`	Path separator (`':'` on UNIX, `','` on Windows)
`user.dir`	User's current working directory when the properties were initialized
`user.home`	User's home directory
`user.language`	The two-letter language code of the default locale (Java 1.1 only)
`user.name`	User's account name
`user.region`	The two-letter country code of the default locale (Java 1.1 only)
`user.timezone`	The default time zone (Java 1.1 only)

Additional properties may be defined by the run-time environment. The `-D` command-line option can be used to define system properties when a program is run.

The `Runtime` class is related to the `System` class; it provides access to information about the environment in which a program is running.

Class Summary

```
public final class java.lang.System extends java.lang.Object {
    // Constants
    public static final PrintStream err;
    public static final InputStream in;
    public static final PrintStream out;
    // Class Methods
    public static void arraycopy(Object src, int srcOffset,
                                 Object dst, int dstOffset, int length);
    public static long currentTimeMillis();
    public static void exit(int status);
    public static void gc();
    public static Properties getProperties();
    public static String getProperty(String key);
    public static String getProperty(String key, String default);
    public static SecurityManager getSecurityManager();
    public static String getenv(String name);           // Deprecated in 1.1
    public static native int identityHashCode(Object x);   // New in 1.1
    public static void load(String filename);
    public static void loadLibrary(String libname);
    public static void runFinalization();
    public static void runFinalizersOnExit(boolean value); // New in 1.1
    public static void setErr(PrintStream err);            // New in 1.1
    public static void setIn(InputStream in);              // New in 1.1
    public static void setOut(PrintStream out);            // New in 1.1
    public static void setProperties(Properties props);
    public static void setSecurityManager(SecurityManager s);
}
```

Variables

err

`public static final PrintStream err`

Description

> The standard error stream. In an application environment, this variable refers to a `java.io.PrintStream` object that is associated with the standard error output for the process running the Java virtual machine. In an applet environment, the `PrintStream` is likely to be associated with a separate window, although this is not guaranteed.
>
> The value of `err` can be set using the `setErr()` method. The value of `err` can only be set if the

currenly installed SecurityManager does not throw a SecurityException when the request is made.

Prior to to Java 1.1, err was not final. It has been made final as of Java 1.1 because the unchecked ability to set err is a security hole.

in

public static final InputStream in

Description

The standard input stream. In an application environment, this variable refers to a java.io.InputStream object that is associated with the standard input for the process running the Java virtual machine.

The value of in can be set using the setIn() method. The value of in can only be set if the currenly installed SecurityManager does not throw a SecurityException when the request is made.

Prior to to Java 1.1, in was not final. It has been made final as of Java 1.1 because the unchecked ability to set in is a security hole.

out

public static final PrintStream out

Description

The standard output stream. In an application environment, this variable refers to a java.io.PrintStream object that is associated with the standard output for the process running the Java virtual machine. In an applet environment, the PrintStream is likely to be associated with a separate window, although this is not guaranteed.

out is the most commonly used of the three I/O streams provided by the System class. Even in GUI-based applications, sending output to this stream can be useful for debugging. The usual idiom for sending output to this stream is:

```
System.out.println("Some text");
```

The value of out can be set using the setOut() method. The value of out can only be set if the currenly installed SecurityManager does not throw a SecurityException when the request is made.

Prior to to Java 1.1, out was not final. It has been made final as of Java 1.1 because the unchecked ability to set `out` is a security hole.

Class Methods

arraycopy

`public static void arraycopy(Object src, int src_position, Object dst, int dst_position, int length)`

Parameters

`src`

The source array.

`src_position`

An index into the source array.

`dst`

The destination array.

`dst_position`

An index into the destination array.

`length`

The number of elements to be copied.

Throws

`ArrayIndexOutOfBoundsException`

If the values of the `src_position`, `dst_position`, and `length` arguments imply accessing either array with an index that is less than zero or an index greater than or equal to the number of elements in the array.

`ArrayStoreException`

If the type of value stored in the `src` array cannot be stored in the `dst` array.

`NullPointerException`

If `src` or `dst` is `null`.

Description

This method copies a range of array elements from the `src` array to the `dst` array. The number of elements that are copied is specified by `length`. The elements at positions `src_position` through `src_position+length-1` in `src` are copied to the positions `dst_position` through `dst_position+length-1` in `dst`, respectively.

If `src` and `dst` refer to the same array, the copying is done as if the array elements were first copied to a temporary array and then copied to the destination array.

Before this method does any copying, it performs a number of checks. If either `src` or `dst` are `null`, the method throws a `NullPointerException` and `dst` is not modified.

If any of the following conditions are true, the method throws an `ArrayStoreException`, and `dst` is not modified:

- Either `src` or `dst` refers to an object that is not an array.
- `src` and `dst` refer to arrays whose element types are different primitive types.
- `src` refers to an array that has elements that contain a primitive type, while `dst` refers to an array that has elements that contain a reference type, or vice versa.

If any of the following conditions are true, the method throws an `ArrayIndexOutOfBoundsException`, and `dst` is not modified:

- `srcOffset`, `dstOffset`, or `length` is negative.
- `srcOffset+length` is greater than `src.length()`.
- `dstOffset+length` is greater than `dst.length()`.

Otherwise, if an element in the source array being copied cannot be converted to the type of the destination array using the rules of the assignment operator, the method throws an `ArrayStoreException` when the problem occurs. Since the problem is discovered during the copy operation, the state of the `dst` array reflects the incomplete copy operation.

currentTimeMillis

```
public static native long currentTimeMillis()
```

Returns

The current time as the number of milliseconds since 00:00:00 UTC, January 1, 1970.

Description

This method returns the current time as the number of milliseconds since 00:00:00 UTC, January 1, 1970. It will not overflow until the year 292280995.

The `java.util.Date` class provides more extensive facilities for dealing with times and dates.

exit

`public static void exit(int status)`

Parameters

`status`

The exit status code to use.

Throws

`SecurityException`

If the `checkExit()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method causes the Java virtual machine to exit with the given status code. This method works by calling the `exit()` method of the current `Runtime` object. By convention, a nonzero status code indicates abnormal termination. This method never returns.

gc

`public static void gc()`

Description

This method causes the Java virtual machine to run the garbage collector in the current thread. This method works by calling the `gc()` method of the current `Runtime` object.

The garbage collector finds objects that will never be used again because there are no live references to them. After it finds these objects, the garbage collector frees the storage occupied by these objects.

The garbage collector is normally run continuously in a thread with the lowest possible priority, so that it works intermittently to reclaim storage. The `gc()` method allows a program to invoke the garbage collector explicitly when necessary.

getProperties

`public static Properties getProperties()`

Returns

A `Properties` object that contains the values of all the system properies.

Throws

`SecurityException`

If the `checkPropertiesAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method returns all of the defined system properties encapsulated in a `java.util.Properties` object. If there are no system properties currently defined, a set of default system properties is created and initialized. As discussed in the description of the `System` class, some system properties are guaranteed always to be defined.

getProperty

`public static String getProperty(String key)`

Parameters

`key`

The name of a system property.

Returns

The value of the named system property or `null` if the named property is not defined.

Throws

`SecurityException`

If the `checkPropertyAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method returns the value of the named system property. If there is no definition for the named property, the method returns `null`. If there are no system properties currently defined, a set of default system properties is created and initialized. As discussed in the description of the `System` class, some system properties are guaranteed always to be defined.

`public static String getProperty(String key, String def)`

Parameters

`key`

The name of a system property.

`def`

A default value for the property.

Returns

The value of the named system property, or the default value if the named property is not defined.

Throws

`SecurityException`

If the `checkPropertyAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method returns the value of the named system property. If there is no definition for the named property, the method returns the default value as specified by the `def` parameter. If there are no system properties currently defined, a set of default system properties is created and initialized. As discussed earlier in the description of the `System` class, some system properties are guaranteed to always be defined.

getSecurityManager

`public static SecurityManager getSecurityManager()`

Returns

> A reference to the installed `SecurityManager` object or `null` if there is no `SecurityManager` object installed.

Description

> This method returns a reference to the installed `SecurityManager` object. If there is no `SecurityManager` object installed, the method returns `null`.

getenv

`public static String getenv(String name)`

Availability

> Deprecated as of JDK 1.1

Parameters

> `name`
>
> > The name of a system-dependent environment variable.

Returns

> The value of the environment variable or `null` if the variable is not defined.

Description

> This method is obsolete; it always throws an error. Use `getProperties()` and the `-D` option instead.

identityHashCode

`public static native int identityHashCode(Object x)`

Availability

> New as of JDK 1.1

Parameters

`x`

An object.

Returns

The identity hashcode value for the specified object.

Description

This method returns the same hashcode value for the specified object as would be returned by the default `hashCode()` method of `Object`, regardless of whether or not the object's class overrides `hashCode()`.

load

`public void load(String filename)`

Parameters

`filename`

A string that specifies the complete path of the file to be loaded.

Throws

`SecurityException`

If the `checkLink()` method of the `SecurityManager` throws a `SecurityException`.

`UnsatisfiedLinkError`

If the method is unsuccessful in loading the specified dynamically linked library.

Description

This method loads the specified dynamically linked library. This method works by calling the `load()` method of the current `Runtime` object.

loadLibrary

`public void loadLibrary(String libname)`

Parameters

`libname`

A string that specifies the name of a dynamically linked library.

Throws

`SecurityException`

If the `checkLink()` method of the `SecurityManager` throws a `SecurityException`.

`UnsatisfiedLinkError`

If the method is unsuccessful in loading the specified dynamically linked library.

Description

This method loads the specified dynamically linked library. It looks for the specified library in a platform-specific way. This method works by calling the `loadLibrary()` method of the current `Runtime` object.

runFinalization

`public static void runFinalization()`

Description

This method causes the Java virtual machine to run the `finalize()` methods of any objects in the finalization queue in the current thread. This method works by calling the `runFinalization()` method of the current `Runtime` object.

When the garbage collector discovers that there are no references to an object, it checks to see if the object has a `finalize()` method that has never been called. If the object has such a `finalize()` method, the object is placed in the finalization queue. While there is a reference to the object in the finalization queue, the object is no longer considered garbage collectable.

Normally, the objects in the finalization queue are handled by a separate finalization thread that runs continuously at a very low priority. The finalization thread removes an object from the queue and calls its `finalize()` method. As long as the `finalize()` method does not generate a reference to the object, the object again becomes available for garbage collection.

Because the finalization thread runs at a very low priority, there may be a long delay from the time that an object is put on the finalization queue until the time that its `finalize()` method is called. The `runFinalization()` method allows a program to run the `finalize()` methods explicitly. This can be useful when there is a shortage of some resource that is released by a `finalize()` method.

runFinalizersOnExit

`public static void runFinalizersOnExit(boolean value)`

Availability

New as of JDK 1.1

Parameters

`value`

A `boolean` value that specifies whether or not finalization occurs on exit.

Throws

`SecurityException`

If the `checkExit()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method specifies whether or not the `finalize()` methods of all objects that have `finalize()` methods are run before the Java virtual machine exits. By default, the finalizers are not run on exit. This method works by calling the `runFinalizersOnExit()` method of the current `Runtime` object.

setErr

`public static void setErr(PrintStream err)`

Availability

New as of JDK 1.1

Parameters

`err`

A `PrintStream` object to use for the standard error stream.

Throws

`SecurityException`

If the `checkExec()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method sets the standard error stream to be this `PrintStream` object.

setIn

`public static void setIn(InputStream in)`

Availability

New as of JDK 1.1

Parameters

`in`

A `InputStream` object to use for the standard input stream.

Throws

`SecurityException`

If the `checkExec()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method sets the standard input stream to be this `InputStream` object.

setOut

`public static void setOut(PrintStream out)`

Availability

New as of JDK 1.1

Parameters

out

A `PrintStream` object to use for the standard output stream.

Throws

SecurityException

If the `checkExec()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method sets the standard output stream to be this `PrintStream` object.

setProperties

`public static void setProperties(Properties props)`

Parameters

props

A reference to a `Properties` object.

Throws

SecurityException

If the `checkPropertiesAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method replaces the current set of system property definitions with a new set of system property definitions that are encapsulated by the given `Properties` object. As discussed in the description of the `System` class, some system properties are guaranteed to always be defined.

setSecurityManager

`public static void setSecurityManager(SecurityManager s)`

Parameters

s

A reference to a SecurityManager object.

Throws

SecurityException

If a SecurityManager object has already been installed.

Description

This method installs the given SecurityManager object. If s is null, then no SecurityManager object is installed. Once a SecurityManager object is installed, any subsequent calls to this method throw a SecurityException.

Inherited Methods

Method	Inherited From	Method	Inherited From
clone()	Object	equals(Object)	Object
finalize()	Object	getClass()	Object
hashCode()	Object	notify()	Object
notifyAll()	Object	toString()	Object
wait()	Object	wait(long)	Object
wait(long, int)	Object		

See Also

ArrayIndexOutOfBoundsException, ArrayStoreException, InputStream, NullPointerException, Object, PrintStream, Process, Runtime, SecurityException, SecurityManager, UnsatisfiedLinkError

PREVIOUS
StringIndexOutOfBoundsException
HOME
BOOK INDEX
NEXT
Thread

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Thread

Name

Thread

Synopsis

Class Name:

`java.lang.Thread`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

`java.lang.Runnable`

Availability:

JDK 1.0 or later

Description

The `Thread` class encapsulates all of the information about a single thread of control running in a Java environment. `Thread` objects are used to control threads in a multithreaded program.

The execution of Java code is always under the control of a `Thread` object. The `Thread` class provides a `static` method called `currentThread()` that can be used to get a reference to the `Thread` object that controls the current thread of execution.

In order for a `Thread` object to be useful, it must be associated with a method that it is supposed to run. Java provides two ways of associating a `Thread` object with a method:

- Declare a subclass of `Thread` that defines a `run()` method. When such a class is instantiated and the object's `start()` method is called, the thread invokes this `run()` method.

- Pass a reference to an object that implements the `Runnable` interface to a `Thread` constructor. When the `start()` method of such a `Thread` object is called, the thread invokes the `run()` method of the `Runnable` object.

After a thread is started, it dies when one of the following things happens:

- The `run()` method called by the `Thread` returns.

- An exception is thrown that causes the `run()` method to be exited.

- The `stop()` method of the `Thread` is called.

Class Summary

```
public class java.lang.Thread extends java.lang.Object
                              implements java.lang.Runnable {
    // Constants
    public final static int MAX_PRIORITY;
    public final static int MIN_PRIORITY;
    public final static int NORM_PRIORITY;
    // Constructors
    public Thread();
    public Thread(Runnable target);
    public Thread(Runnable target, String name);
    public Thread(String name);
    public Thread(ThreadGroup group, Runnable target);
    public Thread(ThreadGroup group, Runnable target, String name);
    public Thread(ThreadGroup group, String name);
    // Class Methods
    public static int activeCount();
    public static native Thread currentThread();
    public static void dumpStack();
    public static int enumerate(Thread tarray[]);
    public static boolean interrupted();
    public static native void sleep(long millis);
    public static void sleep(long millis, int nanos);
    public static native void yield();
    // Instance Methods
    public void checkAccess();
    public native int countStackFrames();
    public void destroy();
    public final String getName();
    public final int getPriority();
    public final ThreadGroup getThreadGroup();
    public void interrupt();
    public final native boolean isAlive();
    public final boolean isDaemon();
    public boolean isInterrupted();
    public final void join();
    public final synchronized void join(long millis);
    public final synchronized void join(long millis, int nanos);
    public final void resume();
    public void run();
    public final void setDaemon(boolean on);
    public final void setName(String name);
    public final void setPriority(int newPriority);
    public synchronized native void start();
    public final void stop();
    public final synchronized void stop(Throwable o);
    public final void suspend();
    public String toString();
}
```

Constants

MAX_PRIORITY

`public final static int MAX_PRIORITY = 10`

Description

The highest priority a thread can have.

MIN_PRIORITY

`public final static int MIN_PRIORITY = 1`

Description

The lowest priority a thread can have.

NORM_PRIORITY

`public final static int NORM_PRIORITY = 5`

Description

The default priority assigned to a thread.

Constructors

Thread

`public Thread()`

Throws

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

Creates a `Thread` object that belongs to the same `ThreadGroup` object as the current thread, has the same daemon attribute as the current thread, has the same priority as the current thread, and has a default name.

A `Thread` object created with this constructor invokes its own `run()` method when the `Thread` object's `start()` method is called. This is not useful unless the object belongs to a subclass of the `Thread` class that overrides the `run()` method.

Calling this constructor is equivalent to:

```
Thread(null, null, genName)
```

`genName` is an automatically generated name of the form `"Thread-"+n`, where `n` is an integer incremented each time a `Thread` object is created.

public Thread(String name)

Parameters

`name`

The name of this `Thread` object.

Throws

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

Creates a `Thread` object that belongs to the same `ThreadGroup` object as the current thread, has the same daemon attribute as the current thread, has the same priority as the current thread, and has the specified name.

A `Thread` object created with this constructor invokes its own `run()` method when the `Thread` object's `start()` method is called. This is not useful unless the object belongs to a subclass of the `Thread` class that overrides the `run()` method.

Calling this constructor is equivalent to:

```
Thread(null, null, name)
```

The uniqueness of the specified `Thread` object's name is not checked, which may be a problem for programs that attempt to identify `Thread` objects by their name.

`public Thread(ThreadGroup group, Runnable target)`

Parameters

`group`

The `ThreadGroup` object that this `Thread` object is to be added to.

`target`

A reference to an object that implements the `Runnable` interface.

Throws

SecurityException

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

Creates a `Thread` object that belongs to the specified `ThreadGroup` object, has the same daemon attribute as the current thread, has the same priority as the current thread, and has a default name.

A `Thread` object created with this constructor invokes the `run()` method of the specified `Runnable` object when the `Thread` object's `start()` method is called.

Calling this constructor is equivalent to:

```
Thread(group, target, genName)
```

`genName` is an automatically generated name of the form `"Thread-"+n`, where `n` is an integer that is incremented each time a `Thread` object is created.

`public Thread(ThreadGroup group, Runnable target, String name)`

Parameters

`group`

The `ThreadGroup` object that this `Thread` object is to be added to.

`target`

A reference to an object that implements the `Runnable` interface.

`name`

The name of this `Thread` object.

Throws

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

Creates a `Thread` object that belongs to the specified `ThreadGroup` object, has the same daemon attribute as the current thread, has the same priority as the current thread, and has the specified name.

A `Thread` object created with this constructor invokes the `run()` method of the specified `Runnable` object when the `Thread` object's `start()` method is called.

The uniqueness of the specified `Thread` object's name is not checked, which may be a problem for programs that attempt to identify `Thread` objects by their names.

`public Thread(ThreadGroup group, String name)`

Parameters

`group`

The `ThreadGroup` object that this `Thread` object is to be added to.

`name`

The name of this `Thread` object.

Throws

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

Creates a `Thread` object that belongs to the specified `ThreadGroup` object, has the same daemon attribute as the current thread, has the same priority as the current thread, and has the specified name.

A `Thread` object created with this constructor invokes its own `run()` method when the `Thread` object's `start()` method is called. This is not useful unless the object belongs to a subclass of the `Thread` class that overrides the `run()` method. Calling this constructor is equivalent to:

```
Thread(group, null, name)
```

The uniqueness of the specified `Thread` object's name is not checked, which may be a problem for programs that attempt to identify `Thread` objects by their name.

Class Methods

activeCount

`public static int activeCount()`

Returns

The current number of threads in the `ThreadGroup` of the currently running thread.

Description

This method returns the number of threads in the `ThreadGroup` of the currently running thread for which the `isAlive()` method returns `true`.

currentThread

`public static native Thread currentThread()`

Returns

A reference to the `Thread` object that controls the currently executing thread.

Description

This method returns a reference to the `Thread` object that controls the currently executing thread.

dumpStack

`public static void dumpStack()`

Description

This method outputs a stack trace of the currently running thread.

enumerate

`public static int enumerate(Thread tarray[])`

Parameters

`tarray`

A reference to an array of `Thread` objects.

Returns

The number of `Thread` objects stored in the array.

Description

This method stores a reference in the array for each of the `Thread` objects in the `ThreadGroup` of the currently running thread for which the `isAlive()` method returns `true`.

Calling this method is equivalent to:

```
currentThread().getThreadGroup().enumerate(tarray)
```

If the array is not big enough to contain references to all the `Thread` objects, only as many references as will fit are put into the array. No indication is given that some `Thread` objects were left out, so it is a good idea to call `activeCount()` before calling this method, to get an idea of how large to make the array.

interrupted

`public static boolean interrupted()`

Returns

`true` if the currently running thread has been interrupted; otherwise `false`.

Description

This method determines whether or not the currently running thread has been interrupted.

sleep

`public static native void sleep(long millis)`

Parameters

`millis`

The number of milliseconds that the currently running thread should sleep.

Throws

`InterruptedException`

If another thread interrupts the currently running thread.

Description

This method causes the currently running thread to sleep. The method does not return until at least the specified number of milliseconds have elapsed.

While a thread is sleeping, it retains ownership of all locks. The `Object` class defines a method called `wait()` that is similar to `sleep()` but causes the currently running thread to temporarily relinquish its locks.

public static void sleep(long millis, int nanos)

Parameters

`millis`

The number of milliseconds that the currently running thread should sleep.

`nanos`

An additional number of nanoseconds to sleep.

Throws

`InterruptedException`

If another thread interrupts the currently running thread.

Description

This method causes the currently running thread to sleep. The method does not return until at least the specified number of milliseconds have elapsed.

While a thread is sleeping, it retains ownership of all locks. The `Object` class defines a method called `wait()` that is similar to `sleep()` but causes the currently running thread to temporarily relinquish its locks.

Note that Sun's reference implementation of Java does not attempt to implement the precision implied by this method. Instead, it rounds to the nearest millisecond (unless `millis` is `0`, in which case it rounds up to 1 millisecond) and calls `sleep(long)`.

yield

`public static native void yield()`

Description

> This method causes the currently running thread to yield control of the processor so that another thread can be scheduled.

Instance Methods

checkAccess

`public void checkAccess()`

Throws

> `SecurityException`
>
> > If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

> This method determines if the currently running thread has permission to modify this `Thread` object.

countStackFrames

`public native int countStackFrames()`

Returns

> The number of pending method invocations on this thread's stack.

Description

> This method returns the number of pending method invocations on this thread's stack.

destroy

`public void destroy()`

Description

> This method is meant to terminate this thread without any of the usual cleanup (i.e., any locks held by the thread are not released). This method provides a last-resort way to terminate a thread. While a thread can defeat its `stop()` method by catching objects thrown from it, there is nothing that a thread can do to stop itself from being destroyed.
>
> Note that Sun's reference implementation of Java does not implement the documented functionality of this method. Instead, the implementation of this method just throws a `NoSuchMethodError`.

getName

`public final String getName()`

Returns

> The name of this thread.

Description

> This method returns the name of this `Thread` object.

getPriority

`public final int getPriority()`

Returns

> The priority of this thread.

Description

> This method returns the priority of this `Thread` object.

getThreadGroup

`public final ThreadGroup getThreadGroup()`

Returns

The `ThreadGroup` of this thread.

Description

This method returns a reference to the `ThreadGroup` that this `Thread` object belongs to.

interrupt

`public void interrupt()`

Description

This method interrupts this `Thread` object.

Note that prior to version 1.1, Sun's reference implementation of Java does not implement the documented functionality of this method. Instead, the method just sets a `private` flag that indicates that an interrupt has been requested. None of the methods that should throw an `InterruptedException` currently do. However, the `interrupted()` and `isInterrupted()` methods do return `true` after this method has been called.

isAlive

`public final native boolean isAlive()`

Returns

`true` if this thread is alive; otherwise `false`.

Description

This method determines whether or not this `Thread` object is alive. A `Thread` object is alive if it has been started and has not yet died. In other words, it has been scheduled to run before and can still be scheduled to run again. A thread is generally alive after its `start()` method is called and until its `stop()` method is called.

isDaemon

`public final boolean isDaemon()`

Returns

> `true` if the thread is a daemon thread; otherwise `false`.

Description

> This method determines whether or not this thread is a daemon thread, based on the value of the daemon attribute of this `Thread` object.

isInterrupted

`public boolean isInterrupted()`

Returns

> `true` if this thread has been interrupted; otherwise `false`.

Description

> This method determines whether or not this `Thread` object has been interrupted.

join

`public final void join()`

Throws

> `InterruptedException`
>
> > If another thread interrupts this thread.

Description

> This method allows the thread that calls it to wait for the `Thread` associated with this method to die. The method returns when the `Thread` dies. If this thread is already dead, then this method returns immediately.

`public final synchronized void join(long millis)`

Parameters

`millis`

The maximum number of milliseconds to wait for this thread to die.

Throws

`InterruptedException`

If another thread interrupts this thread.

Description

This method causes a thread to wait to die. The method returns when this `Thread` object dies or after the specified number of milliseconds has elapsed, whichever comes first. However, if the specified number of milliseconds is zero, the method will wait forever for this thread to die. If this thread is already dead, the method returns immediately.

`public final synchronized void join(long millis, int nanos)`

Parameters

`millis`

The maximum number of milliseconds to wait for this thread to die.

`nanos`

An additional number of nanoseconds to wait.

Throws

`InterruptedException`

If another thread interrupts this thread.

Description

This method causes a thread to wait to die. The method returns when this `Thread` object dies or after the specified amount of time has elapsed, whichever comes first. However, if `millis` and `nanos` are zero, the method will wait forever for this thread to die. If this thread is already dead, the method returns immediately.

Note that Sun's reference implementation of Java does not attempt to implement the precision implied by this method. Instead, it rounds to the nearest millisecond (unless `millis` is `0`, in which case it rounds up to 1 millisecond) and calls `join(long)`.

resume

`public final void resume()`

Throws

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method resumes a suspended thread. The method causes this `Thread` object to once again be eligible to be run. Calling this method for a thread that is not suspended has no effect.

run

`public void run()`

Implements

`Runnable.run()`

Description

A `Thread` object's `start()` method causes the thread to invoke a `run()` method. If this `Thread` object was created without a specified `Runnable` object, the `Thread` object's own `run()` method is executed. This behavior is only useful in a subclass of `Thread` that overrides this `run()` method, since the `run()` method of the `Thread` class does not do anything.

setDaemon

`public final void setDaemon(boolean on)`

Parameters

`on`

The new value for this thread's daemon attribute.

Throws

`IllegalThreadStateException`

If this method is called after this thread has been started and while it is still alive.

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method sets the daemon attribute of this `Thread` object to the given value. This method must be called before the thread is started. If a thread dies and there are no other threads except daemon threads alive, the Java virtual machine stops.

setName

`public final void setName(String name)`

Parameters

`name`

The new name for this thread.

Throws

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a

SecurityException.

Description

This method sets the name of this Thread object to the given value. The uniqueness of the specified Thread object's name is not checked, which may be a problem for programs that attempt to identify Thread objects by their name.

setPriority

public final void setPriority(int newPriority)

Parameters

newPriority

The new priority for this thread.

Throws

IllegalArgumentException

If the given priority is less than MIN_PRIORITY or greater than MAX_PRIORITY.

SecurityException

If the checkAccess() method of the SecurityManager throws a SecurityException.

Description

This method sets the priority of this Thread to the given value.

start

public synchronized native void start()

Throws

IllegalThreadStateException

If this `Thread` object's `start()` method has been called before.

Description

This method starts this `Thread` object, allowing it to be scheduled for execution. The top-level code that is executed by the thread is the `run()` method of the `Runnable` object specified in the constructor that was used to create this object. If no such object was specified, the top-level code executed by the thread is this object's `run()` method.

It is not permitted to start a thread more than once.

stop

`public final void stop()`

Throws

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method causes this `Thread` object to stop executing by throwing a `ThreadDeath` object. The object is thrown in this thread, even if the method is called from a different thread. This thread is forced to stop whatever it is doing and throw a newly created `ThreadDeath` object. If this thread was suspended, it is resumed; if it was sleeping, it is awakened. Normally, you should not catch `ThreadDeath` objects in a `try` statement. If you need to catch `ThreadDeath` objects to detect a `Thread` is about to die, the `try` statement that catches `ThreadDeath` objects should rethrow them.

When an object is thrown out of the `run()` method associated with a `Thread`, the `uncaughtException()` method of the `ThreadGroup` for that `Thread` is called. The `uncaughtException()` method normally outputs a stack trace. However, `uncaughtException()` treats a `ThreadDeath` object as a special case by not outputting a stack trace. When the `uncaughtException()` method returns, the thread is dead. The thread is never scheduled to run again.

If this `Thread` object's `stop()` method is called before this thread is started, the `ThreadDeath` object is thrown as soon as the thread is started.

`public final synchronized void stop(Throwable o)`

Parameters

`o`

The object to be thrown.

Throws

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method causes this `Thread` object to stop executing by throwing the given object. Normally, the `stop()` method that takes no arguments and throws a `ThreadDeath` object should be called instead of this method. However, if it is necessary to stop a thread by throwing some other type of object, this method can be used.

The object is thrown in this thread, even if the method is called from a different thread. This thread is forced to stop whatever it is doing and throw the `Throwable` object `o`. If this thread was suspended, it is resumed; if it was sleeping, it is awakened.

When an object is thrown out of the `run()` method associated with a `Thread`, the `uncaughtException()` method of the `ThreadGroup` for that `Thread` is called. If the thrown object is not an instance of the `ThreadDeath` class, `uncaughtException()` calls the thrown object's `printStackTrace()` method and then the thread dies. The thread is never scheduled to run again.

If this `Thread` object's `stop()` method is called before this thread is started, the `ThreadDeath` object is thrown as soon as the thread is started.

suspend

`public final void suspend()`

Throws

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method suspends a thread. The method causes this `Thread` object to temporarily be ineligible to be run. The thread becomes eligible to be run again after its `resume()` method is called. Calling this method for a thread that is already suspended has no effect.

toString

`public String toString()`

Returns

A string representation of this `Thread` object.

Overrides

`Object.toString()`

Description

This method returns a string representation of this `Thread` object.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

IllegalThreadStateException, InterruptedException, Object, Runnable, SecurityException, SecurityManager, ThreadGroup

ThreadDeath

Name

ThreadDeath

Synopsis

Class Name:

`java.lang.ThreadDeath`

Superclass:

`java.lang.Error`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

A `ThreadDeath` object is thown by the `stop()` method of a `Thread` object to kill the thread. Catching

ThreadDeath objects is not recommended. If it is necessary to catch a ThreadDeath object, it is important to rethrow the object so that it is possible to cleanly stop the catching thread.

Class Summary

```
public class java.lang.ThreadDeath extends java.lang.Error {
  // Constructors
  public ThreadDeath();
}
```

Constructors

ThreadDeath

public ThreadDeath()

Description

> This constructor creates a ThreadDeath object with no associated detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
clone()	Object	equals(Object)	Object
fillInStackTrace()	Throwable	finalize()	Object
getClass()	Object	getLocalizedMessage()	Throwable
getMessage()	Throwable	hashCode()	Object
notify()	Object	notifyAll()	Object
printStackTrace()	Throwable	printStackTrace(PrintStream)	Throwable
printStackTrace(PrintWriter)	Throwable	toString()	Object
wait()	Object	wait(long)	Object
wait(long, int)	Object		

See Also

Error, Thread, Throwable

ThreadGroup

Name

ThreadGroup

Synopsis

Class Name:

```
java.lang.ThreadGroup
```

Superclass:

```
java.lang.Object
```

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The ThreadGroup class implements a grouping scheme for threads. A ThreadGroup object can own Thread objects and other ThreadGroup objects. The ThreadGroup class provides methods that allow a ThreadGroup object to control its Thread and ThreadGroup objects as a group. For example, suspend() and resume() methods of a ThreadGroup object call the suspend() and resume() methods of each of the Thread and ThreadGroup objects that belong to the particular ThreadGroup.

When a Java program starts, a ThreadGroup object is created to own the first Thread. Any additional ThreadGroup objects are explicitly created by the program.

Class Summary

```
public class java.lang.ThreadGroup extends java.lang.Object {
    // Constructors
    public ThreadGroup(String name);
    public ThreadGroup(ThreadGroup parent, String name;
    // Instance Methods
    public int activeCount();
    public int activeGroupCount();
    public boolean allowThreadSuspension(boolean b);       // New in 1.1
    public final void checkAccess();
    public final void destroy();
    public int enumerate(Thread list[]);
    public int enumerate(Thread list[], boolean recurse);
    public int enumerate(ThreadGroup list[]);
    public int enumerate(ThreadGroup list[], boolean recurse);
    public final int getMaxPriority();
    public final String getName();
    public final ThreadGroup getParent();
    public final boolean isDaemon();
    public synchronized boolean isDestroyed();             // New in 1.1
    public void list();
    public final boolean parentOf(ThreadGroup g);
    public final void resume();
    public final void setDaemon(boolean daemon);
    public final void setMaxPriority(int pri);
    public final void stop();
    public final void suspend();
    public String toString();
    public void uncaughtException(Thread t, Throwable e);
}
```

Constructors

ThreadGroup

`public ThreadGroup(String name)`

Parameters

`name`

The name of this `ThreadGroup` object.

Throws

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

Creates a `ThreadGroup` object that has the specified name and the same parent `ThreadGroup` as the current thread.

`public ThreadGroup(ThreadGroup parent, String name)`

Parameters

`parent`

The `ThreadGroup` object that this `ThreadGroup` object is to be added to.

`name`

The name of this `ThreadGroup` object.

Throws

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a

SecurityException.

Description

Creates a ThreadGroup object with the specified name and parent ThreadGroup object.

Instance Methods

activeCount

public int activeCount()

Returns

An approximation of the current number of threads in this ThreadGroup object and any child ThreadGroup objects.

Description

This method returns an approximation of the number of threads that belong to this ThreadGroup object and any child ThreadGroup objects. The count is approximate because a thread can die after it is counted, but before the complete count is returned. Also, after a child ThreadGroup is counted but before the total count is returned, additional Thread and ThreadGroup objects can be added to a child ThreadGroup.

activeGroupCount

public int activeGroupCount()

Returns

An approximation of the current number of child ThreadGroup objects in this ThreadGroup object.

Description

This method returns an approximation of the number of child ThreadGroup objects that belong to this ThreadGroup object. The count is approximate because after a child ThreadGroup is counted but before the total count is returned, additional ThreadGroup objects can be added to a child ThreadGroup.

allowThreadSuspension

`public boolean allowThreadSuspension(boolean b)`

Availability

New as of JDK 1.1

Parameters

`b`

A `boolean` value that specifies whether or not the run-time system is allowed to suspend threads due to low memory.

Returns

The `boolean` value `true`.

Description

This method specifies whether or not the Java virtual machine is allowed to suspend threads due to low memory.

checkAccess

`public final void checkAccess()`

Throws

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method determines if the currently running thread has permission to modify this `ThreadGroup` object.

destroy

`public final void destroy()`

Throws

`IllegalThreadStateException`

If this `ThreadGroup` object is not empty, or if it has already been destroyed.

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method destroys this `ThreadGroup` object and any child `ThreadGroup` objects. The `ThreadGroup` must not contain any `Thread` objects. This method also removes the `ThreadGroup` object from its parent `ThreadGroup` object.

enumerate

`public int enumerate(Thread list[])`

Parameters

`list`

A reference to an array of `Thread` objects.

Returns

The number of `Thread` objects stored in the array.

Description

This method stores a reference in the array for each of the `Thread` objects that belongs to this `ThreadGroup` or any of its child `ThreadGroup` objects.

If the array is not big enough to contain references to all the `Thread` objects, only as many references as will fit are put into the array. No indication is given that some `Thread` objects were left out, so it is a good idea to call `activeCount()` before calling this method, to get an idea of how large to make the array.

`public int enumerate(Thread list[], boolean recurse)`

Parameters

`list`

A reference to an array of `Thread` objects.

`recurse`

A `boolean` value that specifies whether or not to include `Thread` objects that belong to child `ThreadGroup` objects of this `ThreadGroup` object.

Returns

The number of `Thread` objects stored in the array.

Description

This method stores a reference in the array for each of the `Thread` objects that belongs to this `ThreadGroup` object. If `recurse` is `true`, the method also stores a reference for each of the `Thread` objects that belongs to a child `ThreadGroup` object of this `ThreadGroup`.

If the array is not big enough to contain references to all the `Thread` objects, only as many references as will fit are put into the array. No indication is given that some `Thread` objects were left out, so it is a good idea to call `activeCount()` before calling this method, to get an idea of how large to make the array.

`public int enumerate(ThreadGroup list[])`

Parameters

`list`

A reference to an array of `ThreadGroup` objects.

Returns

The number of `ThreadGroup` objects stored in the array.

Description

This method stores a reference in the array for each `ThreadGroup` object that belongs to this `ThreadGroup` or any of its child `ThreadGroup` objects.

If the array is not big enough to contain references to all the `ThreadGroup` objects, only as many references as will fit are put into the array. No indication is given that some `ThreadGroup` objects were left out, so it is a good idea to call `activeGroupCount()` before calling this method, to get an idea of how large to make the array.

`public int enumerate(Thread list[], boolean recurse)`

Parameters

`list`

A reference to an array of `ThreadGroup` objects.

`recurse`

A `boolean` value that specifies whether or not to include `ThreadGroup` objects that belong to child `ThreadGroup` objects of this `ThreadGroup` object.

Returns

The number of `ThreadGroup` objects stored in the array.

Description

This method stores a reference in the array for each of the `ThreadGroup` objects that belongs to this `ThreadGroup` object. If `recurse` is `true`, the method also stores a reference for each of the `ThreadGroup` objects that belongs to a child `ThreadGroup` object of this `ThreadGroup`.

If the array is not big enough to contain references to all the `ThreadGroup` objects, only as many references as will fit are put into the array. No indication is given that some `ThreadGroup` objects were left out, so it is a good idea to call `activeGroupCount()` before calling this method, to get an idea of how large to make the array.

getMaxPriority

`public final int getMaxPriority()`

Returns

> The maximum priority that can be assigned to `Thread` objects that belong to this `ThreadGroup` object.

Description

> This method returns the maximum priority that can be assigned to `Thread` objects that belong to this `ThreadGroup` object.
>
> It is possible for a `ThreadGroup` to contain `Thread` objects that have higher priorities than this maximum, if they were given that higher priority before the maximum was set to a lower value.

getName

`public final String getName()`

Returns

> The name of this `ThreadGroup` object.

Description

> This method returns the name of this `ThreadGroup` object.

getParent

`public final ThreadGroup getParent()`

Returns

> The parent `ThreadGroup` object of this `ThreadGroup`, or `null` if this `ThreadGroup` is the root of the thread group hierarchy.

Description

> This method returns the parent `ThreadGroup` object of this `ThreadGroup` object. If this `ThreadGroup` is at the root of the thread group hierarchy and has no parent, the method returns `null`.

isDaemon

`public final boolean isDaemon()`

Returns

> `true` if this `ThreadGroup` is a daemon thread group; otherwise `false`.

Description

> This method determines whether or not this `ThreadGroup` is a daemon thread group, based on the value of daemon attribute of this `ThreadGroup` object. A daemon thread group is destroyed when the last `Thread` in it is stopped, or the last `ThreadGroup` in it is destroyed.

isDestroyed

`public synchronized boolean isDestroyed()`

Availability

> New as of JDK 1.1

Returns

> `true` if this `ThreadGroup` has been destroyed; otherwise `false`.

Description

> This method determines whether or not this `ThreadGroup` has been destroyed.

list

`public void list()`

Description

> This method outputs a listing of the contents of this `ThreadGroup` object to `System.out`.

parentOf

`public final boolean parentOf(ThreadGroup g)`

Parameters

`g`

A `ThreadGroup` object.

Returns

`true` if this `ThreadGroup` object is the same `ThreadGroup`, or a direct or indirect parent of the specified `ThreadGroup`; otherwise `false`.

Description

This method determines if this `ThreadGroup` object is the same as the specified `ThreadGroup` or one of its ancestors in the thread-group hierarchy.

resume

`public final void resume()`

Throws

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method resumes each `Thread` object that directly or indirectly belongs to this `ThreadGroup` object by calling its `resume()` method.

setDaemon

`public final void setDaemon(boolean daemon)`

Parameters

`daemon`

The new value for this `ThreadGroup` object's daemon attribute.

Throws

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method sets the daemon attribute of this `ThreadGroup` object to the given value. A daemon thread group is destroyed when the last `Thread` in it is stopped, or the last `ThreadGroup` in it is destroyed.

setMaxPriority

`public final void setMaxPriority(int pri)`

Parameters

`pri`

The new maximum priority for `Thread` objects that belong to this `ThreadGroup` object.

Description

This method sets the maximum priority that can be assigned to `Thread` objects that belong to this `ThreadGroup` object.

It is possible for a `ThreadGroup` to contain `Thread` objects that have higher priorities than this maximum, if they were given that higher priority before the maximum was set to a lower value.

stop

`public final void stop()`

Throws

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method stops each `Thread` object that directly or indirectly belongs to this `ThreadGroup` object by calling its `stop()` method.

suspend

`public final void suspend()`

Throws

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method suspends each `Thread` object that directly or indirectly belongs to this `ThreadGroup` object by calling its `suspend()` method.

toString

`public String toString()`

Returns

A string representation of this `ThreadGroup` object.

Overrides

`Object.toString()`

Description

This method returns a string representation of this `ThreadGroup` object.

uncaughtException

`public void uncaughtException(Thread t, Throwable e)`

Parameters

`t`

A reference to a `Thread` that just died because of an uncaught exception.

`e`

The uncaught exception.

Description

This method is called when a `Thread` object that belongs to this `ThreadGroup` object dies because of an uncaught exception. If this `ThreadGroup` object has a parent `ThreadGroup` object, this method just calls the parent's `uncaughtException()` method. Otherwise, this method must determine whether the uncaught exception is an instance of `ThreadDeath`. If it is, nothing is done. If it is not, the method calls the `printStackTrace()` method of the exception object.

If this method is overridden, the overriding method should end with a call to `super.uncaughtException()`.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`IllegalThreadStateException`, `Object`, `Runnable`, `SecurityException`, `SecurityManager`, `Thread`, `Throwable`

Throwable

Name

Throwable

Synopsis

Class Name:

```
java.lang.Throwable
```

Superclass:

```
java.lang.Object
```

Immediate Subclasses:

```
java.lang.Error, java.lang.Exception
```

Interfaces Implemented:

```
java.io.Serializable
```

Availability:

JDK 1.0 or later

Description

The `Throwable` class is the superclass of all objects that can be thrown by the `throw` statement in Java. This is a requirement of the `throw` statement.

A `Throwable` object can have an associated message that provides more detail about the particular error or exception that is being thrown.

The `Throwable` class provides a method that outputs information about the state of the system when an exception object is created. This method can be useful in debugging Java programs.

The subclasses of `Throwable` that are provided with Java do not add functionality to `Throwable`. Instead, they offer more specific classifications of errors and exceptions.

Class Summary

```
public class java.lang.Throwable extends java.lang.Object
                                 implements java.lang.Serializable {
    // Constructors
    public Throwable();
    public Throwable(String message);
    // Instance Methods
    public native Throwable fillInStackTrace();
    public String getLocalizedMessage();              // New in 1.1
    public String getMessage();
    public void printStackTrace();
    public void printStackTrace(PrintStream s);
    public void printStackTrace(PrintWriter s);       // New in 1.1
    public String toString();
}
```

Constructors

Throwable

`public Throwable()`

Description

Creates a `Throwable` object with no associated message. This constructor calls `fillInStackTrace()` so that information is available for `printStackTrace()`.

`public Throwable(String message)`

Parameters

`message`

A message string to be associated with the object.

Description

Create a `Throwable` object with an associated message. This constructor calls `fillInStackTrace()` so that information is available for `printStackTrace()`.

Instance Methods

fillInStackTrace

`public native Throwable fillInStackTrace()`

Returns

A reference to this object.

Description

This method puts stack trace information in this `Throwable` object. It is not usually necessary to explicitly call this method, since it is called by the constructors of the class. However, this method can be useful when rethrowing an object. If the stack trace information in the object needs to reflect the location that the object is rethrows from, `fillInStackTrace()` should be called.

getLocalizedMessage

`public String getLocalizedMessage()`

Availability

New as of JDK 1.1

Returns

A localized version of the `String` object associated with this `Throwable` object, or `null` if there is no message associated with this object.

Description

This method creates a localized version of the message that was associated with this object by its constructor.

The `getLocalizedMessage()` method in `Throwable` always returns the same result as `getMessage()`. A subclass must override this method to produce a locale-specific message.

getMessage

`public String getMessage()`

Returns

The `String` object associated with this `Throwable` object, or `null` if there is no message associated with this object.

Description

This method returns any string message that was associated with this object by its constructor.

printStackTrace

`public void printStackTrace()`

Description

This method outputs the string representation of this `Throwable` object and a stack trace to `System.err`.

`public void printStackTrace(PrintStream s)`

Parameters

s

A `java.io.PrintStream` object.

Description

This method outputs the string representation of this `Throwable` object and a stack trace to the specified `PrintStream` object.

`public void printStackTrace(PrintStream w)`

Availability

New as of JDK 1.1

Parameters

s

A `java.io.PrintWriter` object.

Description

This method outputs the string representation of this `Throwable` object and a stack trace to the specified `PrintWriter` object.

toString

`public String toString()`

Returns

A string representation of this object.

Overrides

`Object.toString()`

Description

This method returns a string representation of this `Throwable` object.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`Error`, `Exception`, `Object`

UnknownError

Name

UnknownError

Synopsis

Class Name:

`java.lang.UnknownError`

Superclass:

`java.lang.VirtualMachineError`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

An `UnknownError` is thrown when an error of unknown origins is detected in the run-time system.

Class Summary

```
public class java.lang.UnknownError
             extends java.lang.VirtualMachineError {
  // Constructors
  public UnknownError();
  public UnknownError(String s);
}
```

Constructors

UnknownError

public UnknownError()

Description

This constructor creates an `UnknownError` with no associated detail message.

public UnknownError(String s)

Parameters

`s`

The detail message.

Description

This constructor creates an `UnknownError` with the specified detail message.

Inherited Methods

Method	**Inherited From**	**Method**	**Inherited From**
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`

`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Error`, `Throwable`, `VirtualMachineError`

PREVIOUS HOME NEXT
Throwable BOOK INDEX UnsatisfiedLinkError

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

UnsatisfiedLinkError

Name

UnsatisfiedLinkError

Synopsis

Class Name:

`java.lang.UnsatisfiedLinkError`

Superclass:

`java.lang.LinkageError`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

An `UnsatisfiedLinkError` is thrown when the implementation of a native method cannot be found.

Class Summary

```
public class java.lang.UnsatisfiedLinkError
             extends java.lang.LinkageError {
  // Constructors
  public UnsatisfiedLinkError();
  public UnsatisfiedLinkError(String s);
}
```

Constructors

UnsatisfiedLinkError

public UnsatisfiedLinkError()

Description

This constructor creates an `UnsatisfiedLinkError` with no associated detail message.

public UnsatisfiedLinkError(String s)

Parameters

`s`

The detail message.

Description

This constructor creates an `UnsatisfiedLinkError` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`

`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Error`, `LinkageError`, `Throwable`

PREVIOUS | HOME | NEXT
UnknownError | BOOK INDEX | VerifyError

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

VerifyError

Name

VerifyError

Synopsis

Class Name:

`java.lang.VerifyError`

Superclass:

`java.lang.LinkageError`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

A `VerifyError` is thrown when the byte-code verifier detects that a class file, though well-formed, contains some sort of internal inconsistency or security problem.

As part of loading the byte-codes for a class, the Java virtual machine may run the *.class* file through the byte-code verifier. The default mode of the virtual machine causes it not to verify classes that are found locally, however. Thus, after compiling an applet and running it locally, you may still get a `VerifyError` when you put it on a web server.

Class Summary

```
public class java.lang.VerifyError extends java.lang.LinkageError {
  // Constructors
  public VerifyError();
  public VerifyError(String s);
}
```

Constructors

VerifyError

public VerifyError()

Description

This constructor creates a `VerifyError` with no associated detail message.

public VerifyError(String s)

Parameters

`s`

The detail message.

Description

This constructor creates a `VerifyError` with the specified detail message.

Inherited Methods

Method	**Inherited From**	**Method**	**Inherited From**

`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Error`, `LinkageError`, `Throwable`

PREVIOUS UnsatisfiedLinkError | HOME | NEXT VirtualMachineError
BOOK INDEX

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

VirtualMachineError

Name

VirtualMachineError

Synopsis

Class Name:

`java.lang.VirtualMachineError`

Superclass:

`java.lang.Error`

Immediate Subclasses:

`java.lang.InternalError`,

`java.lang.OutOfMemoryError`,

`java.lang.StackOverflowError`,

`java.lang.UnknownError`

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The appropriate subclass of `VirtualMachineError` is thrown to indicate that the Java virtual machine has encountered an error.

Class Summary

```
public class java.lang.VirtualMachineError extends java.lang.Error {
  // Constructors
  public VirtualMachineError();
  public VirtualMachineError(String s);
}
```

Constructors

VirtualMachineError

public VirtualMachineError()

Description

This constructor creates a `VirtualMachineError` with no associated detail message.

public VirtualMachineError(String s)

Parameters

`s`

The detail message.

Description

This constructor creates a `VirtualMachineError` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Error`, `InternalError`, `OutOfMemoryError`, `StackOverflowError`, `Throwable`, `UnknownError`

PREVIOUS HOME NEXT
VerifyError BOOK INDEX Void

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Void

Name

Void

Synopsis

Class Name:

`java.lang.Void`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability

New as of JDK 1.1

Description

The `Void` class is an uninstantiable wrapper for the primitive type `void`. The class contains simply a reference to the `Class` object that represents the primitive type `void`. The `Void` class is necessary as of JDK 1.1 to support the Reflection API and class literals.

Class Summary

```
public final class java.lang.Void extends java.lang.Object {
  // Constants
  public static final Class TYPE;
}
```

Constants

TYPE

public static final Class TYPE

The `Class` object that represents the primitive type `void`. It is always true that `Void.TYPE == void.class`.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Byte`, `Character`, `Class`, `Double`, `Float`, `Integer`, `Long`, `Short`

Constructor

Name

Constructor

Synopsis

Class Name:

    `java.lang.reflect.Constructor`

Superclass:

    `java.lang.Object`

Immediate Subclasses:

    None

Interfaces Implemented:

    `java.lang.reflect.Member`

Availability:

    New as of JDK 1.1

Description

The `Constructor` class represents a constructor of a class. A `Constructor` object can be obtained by calling the `getConstructor()` method of a `Class` object. `Constructor` provides methods for getting the name, modifiers, parameters, exceptions, and declaring class of a constructor. The `newInstance()` method can create a new instance of the class that declares a constructor.

Class Summary

```
public final class java.lang.reflect.Constructor extends java.lang.Object
                   implements java.lang.reflect.Member {
  // Instance Methods
  public boolean equals(Object obj);
  public Class getDeclaringClass();
  public Class[] getExceptionTypes();
  public native int getModifiers();
  public String getName();
  public Class[] getParameterTypes();
  public int hashCode();
  public native Object newInstance(Object[] initargs);
  public String toString();
}
```

Instance Methods

equals

public boolean equals(Object obj)

Parameters

obj

The object to be compared with this object.

Returns

true if the objects are equal; false if they are not.

Overrides

Object.equals()

Description

This method returns true if obj is an instance of Constructor, and it is equivalent to this Constructor.

getDeclaringClass

public Class getDeclaringClass()

Returns

The Class object that represents the class that declared this constructor.

Implements

`Member.getDeclaringClass()`

Description

This method returns the `Class` object for the class in which this constructor is declared.

getExceptionTypes

public Class[] getExceptionTypes()

Returns

An array that contains the `Class` objects that describe the exceptions that can be thrown by this constructor.

Description

This method returns an array of `Class` objects that represents the `throws` clause of this constructor. If the constructor does not throw any exceptions, an array of length `0` is returned. As of Java 1.1.2, this method is not properly supported: it always returns an empty array.

getModifiers

public native int getModifiers()

Returns

An integer that represents the modifier keywords used to declare this constructor.

Implements

`Member.getModifiers()`

Description

This method returns an integer value that represents the modifiers of this constructor. The `Modifier` class should decode the returned value.

getName

public String getName()

Returns

The name of this constructor as a `String`.

Implements

`Member.getName()`

Description

This method returns the name of this constructor, which is always the same as the name of the declaring class.

getParameterTypes

public Class[] getParameterTypes()

Returns

An array that contains the `Class` objects that describe the parameter types that this constructor accepts.

Description

This method returns an array of `Class` objects that represents the parameter types this constructor accepts. The parameter types are listed in the order in which they are declared. If the constructor does not take any parameters, an array of length `0` is returned.

hashCode

public int hashCode()

Returns

A hashcode for this object.

Overrides

`Object.hashCode()`

Description

This method returns a hashcode for this `Constructor`.

newInstance

`public native Object newInstance(Object[] initargs) throws InstantiationException, IllegalAccessException, IllegalArgumentException, InvocationTargetException`

Parameters

`initargs`

An array of arguments to be passed to this constructor.

Returns

The newly created object.

Throws

`InstantiationException`

If the declaring class of this constructor is `abstract`.

`IllegalAccessException`

If the constructor is inaccessible.

`IllegalArgumentException`

If `initargs` is the wrong length or contains any value of the wrong type.

`InvocationTargetException`

If the constructor itself throws an exception.

Description

This method executes the constructor represented by this object using the given array of arguments. Thus, it creates and initializes a new instance of the declaring class of the constructor. If a particular parameter is of a primitive type, the corresponding argument is automatically unwrapped and converted to the appropriate type, if possible. If that is not possible, an `IllegalArgumentException` is thrown. If the constructor itself throws an exception, the exception is placed in an `InvocationTargetException`, which is then thrown to the caller of `newInstance()`. If the constructor completes normally, the newly created instance is returned.

toString

public String toString()

Returns

A string representation of this object.

Overrides

`Object.toString()`

Description

This method returns a string representation of this `Constructor`. This string contains the access modifiers of the constructor, if any, followed by the fully qualified name of the declaring class and a list of the parameters of the constructor, if any. The list is enclosed by parentheses, and the individual parameters are separated by commas. If the constructor does not have any parameters, just the parentheses are included in the string.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`Class`, `Field`, `InstantiationException`, `IllegalAccessException`, `IllegalArgumentException`, `InvocationTargetException`, `Member`, `Method`, `Modifier`, `Object`

PREVIOUS
Array

HOME
BOOK INDEX

NEXT
Field

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

BigInteger

Name

BigInteger

Synopsis

Class Name:

`java.math.BigInteger`

Superclass:

`java.lang.Number`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `BigInteger` class represents an arbitrary-precision integer value. You should use this class if a `long` is not big enough for your purposes. The number in a `BigInteger` is represented by a sign value and a magnitude, which is an arbitrarily large array of bytes. A `BigInteger` cannot overflow.

Most of the methods in `BigInteger` perform mathematical operations or make comparisons with other `BigInteger` objects. BigInteger also defines some methods for handling modular arithmetic and determining primality that are needed for cryptographic purposes.

Class Summary

```
public class java.math.BigInteger extends java.lang.Number {
  // Constructors
  public BigInteger(byte[] val);
  public BigInteger(int signum, byte[] magnitude);
  public BigInteger(String val);
  public BigInteger(String val, int radix);
  public BigInteger(int numBits, Random rndSrc);
  public BigInteger(int bitLength, int certainty, Random rnd);
  // Class Methods
  public static BigInteger valueOf(long val);
  // Instance Methods
  public BigInteger abs();
  public BigInteger add(BigInteger val);
  public BigInteger and(BigInteger val);
  public BigInteger andNot(BigInteger val);
  public int bitCount();
  public int bitLength();
  public BigInteger clearBit(int n);
  public int compareTo(BigInteger val);
  public BigInteger divide(BigInteger val);
  public BigInteger[] divideAndRemainder(BigInteger val);
  public double doubleValue();
  public boolean equals(Object x);
  public BigInteger flipBit(int n);
  public float floatValue();
  public BigInteger gcd(BigInteger val);
  public int getLowestSetBit();
  public int hashCode();
  public int intValue();
  public boolean isProbablePrime(int certainty);
  public long longValue();
  public BigInteger max(BigInteger val);
  public BigInteger min(BigInteger val);
  public BigInteger mod(BigInteger m);
  public BigInteger modInverse(BigInteger m);
  public BigInteger modPow(BigInteger exponent, BigInteger m);
  public BigInteger multiply(BigInteger val);
  public BigInteger negate();
  public BigInteger not();
  public BigInteger or(BigInteger val);
  public BigInteger pow(int exponent);
  public BigInteger remainder(BigInteger val);
  public BigInteger setBit(int n);
  public BigInteger shiftLeft(int n);
  public BigInteger shiftRight(int n);
  public int signum();
  public BigInteger subtract(BigInteger val);
  public boolean testBit(int n);
  public byte[] toByteArray();
  public String toString();
  public String toString(int radix);
  public BigInteger xor(BigInteger val);
}
```

Constructors

BigInteger

public BigInteger(byte[] val) throws NumberFormatException

Parameters

`val`

The initial value.

Throws

`NumberFormatException`

If the array does not contain any bytes.

Description

This constructor creates a `BigInteger` with the given initial value. The value is expressed as a two's complement signed integer, with the most significant byte in the first position (`val[0]`) of the array (big-endian). The most significant bit of the most significant byte is the sign bit.

`public BigInteger(int signum, byte[] magnitude) throws NumberFormatException`

Parameters

`signum`

The sign of the value: `-1` indicates negative, `0` indicates zero, and `1` indicates positive.

`magnitude`

The initial magnitude of the value.

Throws

`NumberFormatException`

If `signum` is invalid or if `signum` is 0 but `magnitude` is not 0.

Description

This constructor creates a `BigInteger` with the given initial value and sign. The magnitude is expressed as a big-endian byte array.

public BigInteger(String val) throws NumberFormatException

Parameters

`val`

The initial value.

Throws

`NumberFormatException`

If the string cannot be parsed into a valid `BigInteger`.

Description

This constructor creates a `BigInteger` with the initial value specified by the `String`. The string can contain an optional minus sign followed by zero or more decimal digits. The mapping from characters to digits is provided by the `Character.digit()` method.

`public BigInteger(String val, int radix) throws NumberFormatException`

Parameters

`val`

The initial value.

`radix`

The radix to use to parse the given string.

Throws

`NumberFormatException`

If the string cannot be parsed, or if the radix is not in the allowed range (`Character.MIN_RADIX` through `Character.MAX_RADIX`).

Description

This constructor creates a `BigInteger` with the initial value specified by the `String` using the given radix. The string can contain an optional minus sign followed by zero or more digits in the specified radix. The mapping from characters to digits is provided by the `Character.digit()` method.

`public BigInteger(int numBits, Random rndSrc) throws IllegalArgumentException`

Parameters

`numBits`

The maximum number of bits in the returned number.

`rndSrc`

The source of the random bits.

Throws

`IllegalArgumentException`

If `numBits` is less than zero.

Description

This constructor creates a random `BigInteger` in the range `0` to `2^numBits -1`.

public BigInteger(int bitLength, int certainty, Random rnd)

Parameters

`bitLength`

The maximum number of bits in the returned number.

`certainty`

The certainty that the returned value is a prime number.

`rnd`

The source of the random bits.

Throws

`ArithmeticException`

If numBits is less than 2.

Description

This constructor creates a random BigInteger in the range 0 to 2^numBits-1 that is probably a prime number. The probability that the returned number is prime is greater than 1-.5^certainty. In other words, the higher the value of certainty, the more likely the BigInteger is to be prime, and also the longer it takes for the constructor to create the BigInteger.

Class Methods

valueOf

public static BigInteger valueOf(long val)

Parameters

val

The initial value.

Returns

A BigInteger that represents the given value.

Description

This method creates a BigInteger from the given long value.

Instance Methods

abs

public BigInteger abs()

Returns

A BigInteger that contains the absolute value of this number.

Description

This method returns the absolute value of this BigInteger. If this BigInteger is nonnegative, it is returned. Otherwise, a new BigInteger that contains the absolute value of this BigInteger is returned.

add

public BigInteger add(BigInteger val) throws ArithmeticException

Parameters

`val`

The number to be added.

Returns

A new `BigInteger` that contains the sum of this number and the given value.

Throws

`ArithmeticException`

If any kind of arithmetic error occurs.

Description

This method returns the sum of this `BigInteger` and the given `BigInteger` as a new `BigInteger`.

and

public BigInteger and(BigInteger val)

Parameters

`val`

The number to be ANDed.

Returns

A new `BigInteger` that contains the bitwise AND of this number and the given value.

Description

This method returns the bitwise AND of this `BigInteger` and the supplied `BigInteger` as a new `BigInteger`.

andNot

public BigInteger andNot(BigInteger val)

Parameters

`val`

The number to be combined with this `BigInteger`.

Returns

A new `BigInteger` that contains the bitwise AND of this number and the bitwise negation of the given value.

Description

This method returns the bitwise AND of this `BigInteger` and the bitwise negation of the given `BigInteger` as a new `BigInteger`. Calling this method is equivalent to calling `and(val.not())`.

bitCount

public int bitCount()

Returns

The number of bits that differ from this `BigInteger`'s sign bit.

Description

This method returns the number of bits in the two's complement representation of this `BigInteger` that differ from the sign bit of this `BigInteger`.

bitLength

public int bitLength()

Returns

The number of bits needed to represent this number, excluding a sign bit.

Description

This method returns the minimum number of bits needed to represent this number, not counting a sign bit.

clearBit

public BigInteger clearBit(int n) throws ArithmeticException

Parameters

`n`

The bit to clear.

Returns

A new `BigInteger` that contains the value of this `BigInteger` with the given bit cleared.

Throws

`ArithmeticException`

If `n` is less than 0.

Description

This method returns a new `BigInteger` that is equal to this `BigInteger`, except that the given bit is cleared, or set to zero.

compareTo

public int compareTo(BigInteger val)

Parameters

`val`

The value to be compared.

Returns

`-1` if this number is less than `val`, `0` if this number is equal to `val`, or `1` if this number is greater than `val`.

Description

This method compares this `BigInteger` to the given `BigInteger` and returns a value that indicates the result of the comparison. This method can be used to implement all six of the standard boolean comparison operators: `==`, `!=`, `<=`, `<`, `>=`, and `>`.

divide

public BigInteger divide(BigInteger val) throws ArithmeticException

Parameters

`val`

The divisor.

Returns

A new `BigInteger` that contains the result (quotient) of dividing this number by the given value.

Throws

`ArithmeticException`

If `val` is zero.

Description

This method returns the quotient that results from dividing this `BigInteger` by the given `BigInteger` as a new `BigInteger`. Any fractional remainder is discarded.

divideAndRemainder

```
public BigInteger[] divideAndRemainder(BigInteger val) throws ArithmeticException
```

Parameters

`val`

The divisor.

Returns

An array of `BigInteger` objects that contains the quotient and remainder (in that order) that result from dividing this number by the given value.

Throws

`ArithmeticException`

If `val` is zero.

Description

This method returns the quotient and remainder that results from dividing this `BigInteger` by the given `BigInteger` as an array of new `BigInteger` objects. The first element of the array is equal to `divide(val)`; the second element is equal to `remainder(val)`.

doubleValue

public double doubleValue()

Returns

The value of this `BigInteger` as a `double`.

Overrides

`Number.doubleValue()`

Description

This method returns the value of this `BigInteger` as a `double`. If the value exceeds the limits of a `double`, `Double.POSITIVE_INFINITY` or `Double.NEGATIVE_INFINITY` is returned.

equals

public boolean equals(Object x)

Parameters

`x`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `x` is an instance of `BigInteger`, and it represents the same value as this `BigInteger`.

flipBit

public BigInteger flipBit(int n)

Parameters

`n`

The bit to toggle.

Returns

A new `BigInteger` that contains the value of this `BigInteger` with the given bit toggled.

Throws

`ArithmeticException`

If `n` is less than 0.

Description

This method returns a new `BigInteger` that is equal to this `BigInteger`, except that the given bit is toggled. In other words, if the given bit is `0`, it is set to one, or if it is `1`, it is set to zero.

floatValue

public float floatValue()

Returns

The value of this `BigInteger` as a `float`.

Overrides

`Number.floatValue()`

Description

This method returns the value of this `BigInteger` as a `float`. If the value exceeds the limits of a `float`, `Float.POSITIVE_INFINITY` or `Float.NEGATIVE_INFINITY` is returned.

gcd

public BigInteger gcd(BigInteger val)

Parameters

`val`

The number to be compared.

Returns

A new `BigInteger` that contains the greatest common denominator of this number and the given number.

Description

This method calculates the greatest common denominator of the absolute value of this `BigInteger` and the absolute value of the given `BigInteger`, and returns it as a new `BigInteger`. If both values are `0`, the method returns a `BigInteger` that contains the value `0`.

getLowestSetBit

public int getLowestSetBit()

Returns

> The index of the lowest-order bit with a value of `1`, or `-1` if there are no bits that are `1`.

Description

> This method returns the index of the lowest-order, or rightmost, bit with a value of `1`.

hashCode

public int hashCode()

Returns

> A hashcode for this object.

Overrides

> `Object.hashCode()`

Description

> This method returns a hashcode for this `BigInteger`.

intValue

public int intValue()

Returns

> The value of this `BigInteger` as an `int`.

Overrides

> `Number.intValue()`

Description

> This method returns the value of this `BigInteger` as an `int`. If the value exceeds the limits of an `int`, the excessive high-order bits are discarded.

isProbablePrime

public boolean isProbablePrime(int certainty)

Parameters

`certainty`

The "certainty" that this number is prime, where a higher value indicates more certainty.

Returns

`true` if this number is probably prime; `false` if it is definitely not prime.

Description

This method returns `true` if this number has a probability of being prime that is greater than `1-.5^certainty`. If the number is definitely not prime, `false` is returned.

longValue

public long longValue()

Returns

The value of this `BigInteger` as a `long`.

Overrides

`Number.longValue()`

Description

This method returns the value of this `BigInteger` as a `long`. If the value exceeds the limits of a `long`, the excessive high-order bits are discarded.

max

public BigInteger max(BigInteger val)

Parameters

`val`

The number to be compared.

Returns

The `BigInteger` that represents the greater of this number and the given value.

Description

This method returns the greater of this `BigInteger` and the given `BigInteger`.

min

public BigInteger min(BigInteger val)

Parameters

`val`

The number to be compared.

Returns

The `BigInteger` that represents the lesser of this number and the given value.

Description

This method returns the lesser of this `BigInteger` and the given `BigInteger`.

mod

public BigInteger mod(BigInteger m)

Parameters

`m`

The number to use.

Returns

A new `BigInteger` that contains the modulus of this number and the given number.

Throws

`ArithmeticException`

If `m` is less than or equal to zero.

Description

This method returns a new `BigInteger` that contains the value of this `BigInteger` mod `m`.

modInverse

```
public BigInteger modInverse(BigInteger m) throws ArithmeticException
```

Parameters

`m`

The number to use.

Returns

A new `BigInteger` that contains the multiplicative inverse of the modulus of this number and the given number.

Throws

`ArithmeticException`

If `m` is less than or equal to zero, or if the result cannot be calculated.

Description

This method returns a new `BigInteger` that contains the multiplicative inverse of the value of this `BigInteger` mod `m`.

modPow

public BigInteger modInverse(BigInteger exponent, BigInteger m)

Parameters

`exponent`

The exponent.

`m`

The number to use.

Returns

A new `BigInteger` that contains the modulus of this number raised to the given power and the given number.

Throws

`ArithmeticException`

If `m` is less than or equal to zero.

Description

This method returns a new `BigInteger` that contains the value of this `BigInteger` raised to the given power mod `m`.

multiply

public BigInteger multiply(BigInteger val)

Parameters

`val`

The number to be multiplied.

Returns

A new `BigInteger` that contains the product of this number and the given number.

Description

This method multiplies this `BigInteger` by the given `BigInteger` and returns the product as a new `BigInteger`.

negate

public BigInteger negate()

Returns

A new `BigInteger` that contains the negative of this number.

Description

This method returns a new `BigInteger` that is identical to this `BigInteger` except that its sign is reversed.

not

public BigInteger not()

Returns

A new `BigInteger` that contains the bitwise negation of this number.

Description

This method returns a new `BigInteger` that is calculated by inverting every bit of this `BigInteger`.

or

public BigInteger or(BigInteger val)

Parameters

`val`

The value to be ORed.

Returns

A new `BigInteger` that contains the bitwise OR of this number and the given value.

Description

This method returns the bitwise OR of this `BigInteger` and the given `BigInteger` as a new `BigInteger`.

pow

public BigInteger pow(int exponent) throws ArithmeticException

Parameters

`exponent`

The exponent.

Returns

A new `BigInteger` that contains the result of raising this number to the given power.

Throws

`ArithmeticException`

If `exponent` is less than zero.

Description

This method raises this `BigInteger` to the given power and returns the result as a new `BigInteger`.

remainder

```
public BigInteger remainder(BigInteger val) throws ArithmeticException
```

Parameters

`val`

The divisor.

Returns

A new `BigInteger` that contains the remainder that results from dividing this number by the given value.

Throws

`ArithmeticException`

If `val` is zero.

Description

This method returns the remainder that results from dividing this `BigInteger` by the given `BigInteger` as a new `BigInteger`.

setBit

public BigInteger setBit(int n) throws ArithmeticException

Parameters

`n`

The bit to set.

Returns

A new `BigInteger` that contains the value of this `BigInteger` with the given bit set.

Throws

`ArithmeticException`

If `n` is less than zero.

Description

This method returns a new `BigInteger` that is equal to this `BigInteger`, except that the given bit is set to `1`.

shiftLeft

public BigInteger shiftLeft(int n)

Parameters

`n`

The number of bits to shift.

Returns

A new `BigInteger` that contains the result of shifting the bits of this number left by the given number of bits.

Description

This method returns a new `BigInteger` that contains the value of this `BigInteger` left-shifted by the given number of bits.

shiftRight

public BigInteger shiftRight(int n)

Parameters

`n`

The number of bits to shift.

Returns

A new `BigInteger` that contains the result of shifting the bits of this number right by the given number of bits with sign extension.

Description

This method returns a new `BigInteger` that contains the value of this `BigInteger` right-shifted by the given number of bits with sign extension.

signum

public int signum()

Returns

`-1` is this number is negative, `0` if this number is zero, or `1` if this number is positive.

Description

This method returns a value that indicates the sign of this `BigInteger`.

subtract

public BigInteger subtract(BigInteger val)

Parameters

`val`

The number to be subtracted.

Returns

A new `BigDecimal` that contains the result of subtracting the given number from this number.

Description

This method subtracts the given `BigInteger` from this `BigInteger` and returns the result as a new `BigInteger`.

testBit

public boolean testBit(int n) throws ArithmeticException

Parameters

`n`

The bit to test.

Returns

`true` if the specified bit is `1`; `false` if the specified bit is `0`.

Throws

`ArithmeticException`

If `n` is less than zero.

Description

This method returns `true` if the specified bit in this `BigInteger` is `1`. Otherwise the method returns `false`.

toByteArray

public byte[] toByteArray()

Returns

An array of bytes that represents this object.

Description

This method returns an array of bytes that contains the two's complement representation of this `BigInteger`. The most significant byte is in the first position (`val[0]`) of the array. The array can be used with the `BigInteger(byte[])` constructor to reconstruct the number.

toString

public String toString()

Returns

A string representation of this object in decimal form.

Overrides

`Object.toString()`

Description

This method returns a string representation of this `BigInteger` in decimal form. A minus sign represents the sign if necessary. The mapping from digits to characters is provided by the `Character.forDigit()` method.

public String toString(int radix)

Parameters

`radix`

The radix to use.

Returns

A string representation of this object in the given radix.

Overrides

`Object.toString()`

Description

This method returns a string representation of this `BigInteger` for the given radix. A minus sign is used to represent the sign if necessary. The mapping from digits to characters is provided by the `Character.forDigit()` method.

xor

public BigInteger xor(BigInteger val)

Parameters

`val`

The value to be XORed.

Returns

A new `BigInteger` that contains the bitwise XOR of this number and the given value.

Description

This method returns the bitwise XOR of this `BigInteger` and the given `BigInteger` as a new `BigInteger`.

Inherited Methods

Method	Inherited From	Method	Inherited From
`byteValue()`	`Number`	`clone()`	`Object`
`getClass()`	`Object`	`finalize()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`shortValue()`	`Number`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`ArithmeticException`, `BigDecimal`, `Character`, `Double`, `Float`, `IllegalArgumentException`, `Integer`, `Long`, `Number`, `NumberFormatException`

PREVIOUS HOME NEXT
BigDecimal BOOK INDEX BindException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

ConnectException

Name

ConnectException

Synopsis

Class Name:

`java.net.ConnectException`

Superclass:

`java.net.SocketException`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

A `ConnectException` is thrown when a socket connection cannot be established with a remote machine. This type of exception usually indicates that there is no listening process on the remote machine.

Class Summary

```
public class java.net.ConnectException extends java.net.SocketException {
  // Constructors
  public ConnectException();
  public ConnectException(String msg);
}
```

Constructors

ConnectException

public ConnectException()

Description

This constructor `create`s a `ConnectException` with no associated detail message.

public ConnectException(String msg)

Parameters

`msg`

The detail message.

Description

This constructor create a `ConnectException` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`

`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Throwable`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Exception`, `IOException`, `SocketException`

PREVIOUS | HOME | NEXT
BindException | BOOK INDEX | ContentHandler

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

ContentHandler

Name

ContentHandler

Synopsis

Class Name:

```
java.net.ContentHandler
```

Superclass:

```
java.lang.Object
```

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The `ContentHandler` class is an `abstract` class that defines a method to read data from a

URLConnection and then create an Object appropriate for the type of content it has read. Each subclass of ContentHandler handles a specific type of content (i.e., MIME type).

You do not create ContentHandler objects directly; they are created by an object that implements the ContentHandlerFactory interface. A ContentHandlerFactory object selects and creates an appropriate ContentHandler for the content type. If you write your own ContentHandler subclasses, you should also write your own ContentHandlerFactory. The content handler factory for an application is set by a call to URLConnection.setContentHandlerFactory().

An application does not normally call the getContent() method of a ContentHandler directly; it should call URL.getContent() or URLConnection.getContent() instead.

A ContentHandler works in conjunction with a URLStreamHandler, but their roles do not overlap. The URLStreamHandler deals with the specifics of a protocol, such as negotiating with a server to retrieve a resource, while the ContentHandler expects a data stream from which it can construct an object.

Class Summary

```
public abstract class java.net.ContentHandler extends java.lang.Object {
  // Instance Methods
  public abstract Object getContent(URLConnection urlc) throws IOException;
}
```

Instance Methods

getContent

public abstract Object getContent(URLConnection urlc) throws IOException

Parameters

urlc

A URLConnection that is the data source.

Returns

The Object created from the data source.

Throws

IOException

If any kind of I/O error occurs.

Description

This method reads data from the given `URLConnection` and returns the object that is represented by the data.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`int hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`ContentHandlerFactory`, `IOException`, `URL`, `URLConnection`, `URLStreamHandler`

PREVIOUS
ConnectException

HOME
BOOK INDEX

NEXT
ContentHandlerFactory

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

ContentHandlerFactory

Name

ContentHandlerFactory

Synopsis

Interface Name:

```
java.net.ContentHandlerFactory
```

Super-interface:

None

Immediate Sub-interfaces:

None

Implemented By:

None

Availability:

JDK 1.0 or later

Description

The `ContentHandlerFactory` interface defines a method that creates and returns an appropriate `ContentHandler` object for a given MIME type. The interface is implemented by classes that select `ContentHandler` subclasses to process content.

The `URLStreamHandler` class uses a `ContentHandlerFactory` to create `ContentHandler` objects. The content type is usually implied by the portion of the URL following the last period. For example, given the following URL:

```
http://www.tolstoi.org/anna.html
```

the MIME content type is `text/html`. A `ContentHandlerFactory` that recognizes `text/html` returns a `ContentHandler` object that can process that kind of content.

Interface Declaration

```
public abstract interface java.net.ContentHandlerFactory {
  // Methods
  public abstract ContentHandler createContentHandler(String mimetype);
}
```

Methods

createContentHandler

`public abstract ContentHandler createContentHandler( String mimetype)`

Parameters

`mimetype`

A `String` that represents a MIME type.

Returns

A `ContentHandler` object that can read the specified type of content.

Description

This method creates an object of the appropriate subclass of `ContentHandler` that can read and process data for the given MIME type.

See Also

`ContentHandler`, `URLStreamHandler`

PREVIOUS ContentHandler | HOME BOOK INDEX | NEXT DatagramPacket

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

DatagramPacket

Name

DatagramPacket

Synopsis

Class Name:

`java.net.DatagramPacket`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The `DatagramPacket` class represents a packet of data that can be sent and received over the network using a `DatagramSocket`. The class is used to implement connectionless data communication.

Class Summary

```
public final class java.net.DatagramPacket extends java.lang.Object {
  // Constructors
  public DatagramPacket(byte[] ibuf, int ilength);
  public DatagramPacket(byte[] ibuf, int ilength,
                        InetAddress iaddr, int iport);
  // Instance Methods
  public synchronized InetAddress getAddress();
  public synchronized byte[] getData();
  public synchronized int getLength();
  public synchronized int getPort();
  public synchronized void setAddress(InetAddress iaddr);   // New in 1.1
  public synchronized void setData(byte[] ibuf);            // New in 1.1
  public synchronized void setLength(int ilength);          // New in 1.1
  public synchronized void setPort(int iport);              // New in 1.1
}
```

Constructors

DatagramPacket

public DatagramPacket(byte ibuf[], int ilength)

Parameters

`ibuf`

The data buffer for receiving incoming bytes.

`ilength`

The number of bytes to read.

Description

This constructor creates a `DatagramPacket` that receives data. The value of `ilength` must be less than or equal to `ibuf.length`. This `DatagramPacket` can be passed to `DatagramSocket.receive()`.

public DatagramPacket(byte ibuf[], int ilength, InetAddress iaddr, int iport)

Parameters

`ibuf`

The data buffer for the packet.

`ilength`

The number of bytes to send.

`iaddr`

The destination address.

`iport`

The destination port number.

Description

This constructor creates a `DatagramPacket` that sends packets of length `ilength` to the given port of the specified address. The value of `ilength` must be less than or equal to `ibuf.length`. The packets are sent using `DatagramSocket.send()`.

Instance Methods

getAddress

public synchronized InetAddress getAddress()

Returns

The IP address of the packet.

Description

If this packet has been received, the method returns the address of the machine that sent it. If the packet is being sent, the method returns the destination address.

getData

public synchronized byte[] getData()

Returns

The packet data.

Description

This method returns the data buffer associated with this `DatagramPacket` object. This data is either the data being sent or the data that has been received.

getLength

public synchronized int getLength()

Returns

The packet length.

Description

This method returns the length of the message in the buffer associated with this `DatagramPacket`. This length is either the length of the data being sent or the length of the data that has been received.

getPort

public synchronized int getPort()

Returns

The `port` number of the packet.

Description

If this packet has been received, the method returns the port number of the machine that sent it. If the packet is being sent, the method returns the destination port number.

setAddress

public synchronized void setAddress(InetAddress iaddr)

Availability

New as of JDK 1.1

Parameters

`iaddr`

The destination address for the packet.

Description

This method sets the destination address for this packet. When the packet is sent using `DatagramSocket.send()`, it is sent to the specified address.

setData

public synchronized void setData(byte[] ibuf)

Availability

New as of JDK 1.1

Parameters

`ibuf`

The data buffer for the packet.

Description

This method sets the data for this packet. When the packet is sent using `DatagramSocket.send()`, the specified data is sent.

setLength

public synchronized void setLength(int ilength)

Availability

New as of JDK 1.1

Parameters

`ilength`

The number of bytes to send.

Description

This method sets the length of the data to be sent for this packet. When the packet is sent using `DatagramSocket.send()`, the specified amount of data is sent.

setPort

public synchronized void setPort(int iport)

Availability

New as of JDK 1.1

Parameters

`iport`

The port number for the packet.

Description

This method sets the destination port number for this packet. When the packet is sent using `DatagramSocket.send()`, it is sent to the specified port.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`DatagramSocket`, `InetAddress`

DatagramSocket

Name

DatagramSocket

Synopsis

Class Name:

`java.net.DatagramSocket`

Superclass:

`java.lang.Object`

Immediate Subclasses:

`java.net.MulticastSocket`

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The DatagramSocket class implements packet-oriented, connectionless data communication. In Internet parlance, this is the User Datagram Protocol, commonly known as UDP (see RFC 768). Each packet wanders through the network, routed by its destination address. Different packets can take different paths through the network and may arrive in a different order than they were sent. Furthermore, packets are not even guaranteed to reach their destination. It is up to an application that uses DatagramSocket to determine if data is out of order or missing. While these features may seem like disadvantages of DatagramSocket, there is also some advantage to using this class. Primarily, communication using DatagramSocket is faster than Socket stream communication because of the lack of overhead involved.

Class Summary

```
public class java.net.DatagramSocket extends java.lang.Object {
  // Constructors
  public DatagramSocket();
  public DatagramSocket(int port);
  public DatagramSocket(int port, InetAddress laddr);     // New in 1.1
  // Instance Methods
  public void close();
  public InetAddress getLocalAddress();                   // New in 1.1
  public int getLocalPort();
  public synchronized int getSoTimeout();                 // New in 1.1
  public synchronized void receive(DatagramPacket p);
  public void send(DatagramPacket p);
  public synchronized void setSoTimeout(int timeout);     // New in 1.1
}
```

Constructors

DatagramSocket

public DatagramSocket() throws SocketException

Throws

SocketException

If any kind of socket error occurs.

SecurityException

If the application is not allowed to listen on the port.

Description

This constructor creates a `DatagramSocket` that is bound to any available port on the local host machine.

public DatagramSocket(int port) throws SocketException

Parameters

`port`

A port number.

Throws

`SocketException`

If any kind of socket error occurs.

`SecurityException`

If the application is not allowed to listen on the given port.

Description

This constructor creates a `DatagramSocket` that is bound to the given port on the local host machine.

`public DatagramSocket(int port, InetAddress laddr) throws SocketException`

Availability

New as of JDK 1.1

Parameters

`port`

A port number.

`laddr`

A local address.

Throws

> `SocketException`
>
> > If any kind of socket error occurs.
>
> `SecurityException`
>
> > If the application is not allowed to listen on the given port on the specified host.

Description

> This constructor creates a `DatagramSocket` that is bound to the given port on the specified local host machine.

Instance Methods

close

public void close()

Description

> This method closes the socket, releasing any system resources it holds.

getLocalAddress

public InetAddress getLocalAddress()

Availability

> New as of JDK 1.1

Returns

> The local address of the socket.

Throws

> `SecurityException`

If the application is not allowed to retrieve the address.

Description

This method returns the local address to which this `DatagramSocket` is bound.

getLocalPort

public int getLocalPort()

Returns

`The port` number of the socket.

Description

This method returns the local port to which this `DatagramSocket` is bound.

getSoTimeout

public synchronized int getSoTimeout() throws SocketException

Availability

New as of JDK 1.1

Returns

The receive time-out value for the socket.

Throws

`SocketException`

If any kind of socket error occurs.

Description

This method returns the receive time-out value for this socket. A value of zero indicates that the socket waits indefinitely for an incoming packet, while a non-zero value indicates the number of milliseconds it waits.

receive

`public synchronized void receive(DatagramPacket p) throws IOException`

Parameters

`p`

The `DatagramPacket` that receives incoming data.

Throws

`IOException`

If any kind of I/O error occurs.

`SecurityException`

If the application is not allowed to receive data from the packet's source.

`InterruptedIOException`

If a packet does not arrive before the time-out period expires.

Description

This method receives a datagram packet on this socket. After this method returns, the given `DatagramPacket` contains the packet's data and length, and the sender's address and port number. If the data that was sent is longer that the given packet's data buffer, the data is truncated.

If a time-out value is specified using the `setSoTimeout()` method, the method either returns with the received packet or times out, throwing an `InterruptedIOException`. If no time-out value is specified, the method blocks until it receives a packet.

send

public void send(DatagramPacket p) throws IOException

Parameters

`p`

The `DatagramPacket` to be sent.

Throws

`IOException`

If any kind of I/O error occurs.

`SecurityException`

If the application is not allowed to send data to the packet's destination.

Description

This method sends a packet from this socket. The packet data, packet length, destination address, and destination port number are specified by the given `DatagramPacket`.

setSoTimeout

`public synchronized void setSoTimeout(int timeout) throws SocketException`

Availability

New as of JDK 1.1

Parameters

`timeout`

The new time-out value, in milliseconds, for this socket.

Throws

`SocketException`

If any kind of socket error occurs.

Description

This method is used to set the time-out value that is used for `receive()`. A non-zero value specifies the length of time, in milliseconds, that the `DatagramSocket` should wait for an incoming packet. A value of zero indicates that the `DatagramSocket` should wait indefinitely for an incoming packet. If a time-out value is needed, this method must be called before `receive()`.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`DatagramPacket`, `DatagramSocketImpl`, `InetAddress`, `InterruptedIOException`, `IOException`, `MulticastSocket`, `SecurityException`, `Socket`, `SocketException`

PREVIOUS
DatagramPacket

HOME
BOOK INDEX

NEXT
DatagramSocketImpl

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

DatagramSocketImpl

Name

DatagramSocketImpl

Synopsis

Class Name:

`java.net.DatagramSocketImpl`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `DatagramSocketImpl` class is an `abstract` class that defines the bulk of the methods that make the `DatagramSocket` and `MulticastSocket` classes work. Non-`public` subclasses of `DatagramSocketImpl` provide platform-specific implementations of datagram socket communication.

Class Summary

```
public abstract class java.net.DatagramSocketImpl
                      extends java.lang.Object {
  // Variables
  protected FileDescriptor fd;
  protected int localPort;
  // Protected Instance Methods
  protected abstract void bind(int lport, InetAddress laddr);
  protected abstract void close();
  protected abstract void create();
  protected FileDescriptor getFileDescriptor();
  protected int getLocalPort();
  protected abstract byte getTTL();
  protected abstract void join(InetAddress inetaddr);
  protected abstract void leave(InetAddress inetaddr);
  protected abstract int peek(InetAddress i);
  protected abstract void receive(DatagramPacket p);
  protected abstract void send(DatagramPacket p);
  protected abstract void setTTL(byte ttl);
}
```

Variables

fd

protected FileDescriptor fd

Description

> The file descriptor that represents this socket.

localPort

protected int localPort

Description

> The local port number of this socket.

Protected Instance Methods

bind

`protected abstract void bind(int lport, InetAddress laddr) throws SocketException`

Parameters

`lport`

A port number.

`laddr`

A local address.

Throws

`SocketException`

If any kind of socket error occurs.

Description

This method binds the socket to the given address and port. If the address or the port is unavailable, an exception is thrown.

close

protected void close()

Description

This method closes the socket, releasing any system resources it holds.

create

protected abstract void create() throws SocketException

Throws

`SocketException`

If a socket error occurs.

Description

This method creates a socket that is not bound to an address and port.

getFileDescriptor

protected FileDescriptor getFileDescriptor()

Returns

The file descriptor for this socket.

Description

This method returns the file descriptor associated with this `DatagramSocketImpl`.

getLocalPort

protected int getLocalPort()

Returns

The port number for this socket.

Description

This method returns the local port to which this `DatagramSocketImpl` is bound.

getTTL

protected abstract byte getTTL() throws IOException

Returns

The time-to-live (TTL) value for this socket.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method returns the TTL value for this socket. This value is the number of hops that an outgoing packet can traverse before it is discarded.

join

`protected abstract void join(InetAddress inetaddr) throws IOException`

Parameters

`inetaddr`

The IP address of the group to join.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method is used by `MulticastSocket` to join a multicast group. An exception is thrown if the given address is not a multicast address. While the socket is part of a group, it receives all packets that are sent to the group.

leave

```
protected abstract void leave(InetAddress inetaddr) throws IOException
```

Parameters

`inetaddr`

The IP address of the group to leave.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method is used by `MulticastSocket` to leave a multicast group. An exception is thrown if the given address is not a multicast address.

peek

protected abstract int peek(InetAddress i) throws IOException

Parameters

`i`

A reference to an `InetAddress` object.

Returns

The port number of the next incoming packet.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method places the address of the next incoming packet in the given `InetAddress` object. The method also returns the port number of the next incoming packet. The method looks at the address of an incoming packet to determine if it should be accepted.

receive

`protected abstract void receive(DatagramPacket p) throws IOException`

Parameters

`p`

The `DatagramPacket` that receives incoming data.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method receives a datagram packet on this socket. After this method returns, the given `DatagramPacket` contains the packet's data and length, and the sender's address and port number. If the data that was sent is longer that the given packet's data buffer, the data is truncated.

send

protected abstract void send(DatagramPacket p) throws IOException

Parameters

`p`

The `DatagramPacket` to be sent.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method sends a packet from this socket. The packet data, packet length, destination address, and destination port number are specified by the given `DatagramPacket`.

setTTL

protected abstract void setTTL(byte ttl) throws IOException

Parameters

`ttl`

The new TTL value for this socket.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method is used to set the TTL value of the socket. The TTL value is the number of hops that an outgoing packet can traverse before it is discarded.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`DatagramPacket`, `DatagramSocket`, `FileDescriptor`, `InetAddress`, `IOException`, `MulticastSocket`

PREVIOUS	HOME	NEXT
DatagramSocket	BOOK INDEX	FileNameMap

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

FileNameMap

Name

FileNameMap

Synopsis

Interface Name:

`java.net.FileNameMap`

Super-interface:

None

Immediate Sub-interfaces:

None

Implemented By:

None

Availability:

New as of JDK 1.1

Description

The `FileNameMap` interface defines a method that maps filenames to MIME types. The interface is implemented by classes that provide this mapping. The mapping is typically done by examining the file extension of the filename, or in other words, the part of the filename that follows the final period.

Interface Declaration

```
public abstract interface java.net.FileNameMap {
  // Methods
  public abstract String getContentTypeFor(String fileName);
}
```

Methods

getContentTypeFor

public abstract String getContentTypeFor(String fileName)

Parameters

> `fileName`
>
> > A `String` that contains a filename.

Returns

> The `String` that contains the MIME type that corresponds to the filename.

Description

> This method attempts to determine the MIME type of a file by examining its filename.

See Also

`ContentHandler`, `ContentHandlerFactory`

HttpURLConnection

Name

HttpURLConnection

Synopsis

Class Name:

`java.net.HttpURLConnection`

Superclass:

`java.net.URLConnection`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `HttpURLConnection` class is an `abstract` class that is a subclass of `URLConnection`. `HttpURLConnection` defines many of the HTTP server response codes as constants and provides methods for parsing server responses.

An `HttpURLConnection` object defines a network connection to a resource specified by a URL. Essentially, the `HttpURLConnection` object represents the HTTP request for that resource.

Class Summary

```
public abstract class java.net.HttpURLConnection
                      extends java.net.URLConnection {
  // Constants
  public final static int HTTP_ACCEPTED;
  public final static int HTTP_BAD_GATEWAY;
  public final static int HTTP_BAD_METHOD;
  public final static int HTTP_BAD_REQUEST;
  public final static int HTTP_CLIENT_TIMEOUT;
  public final static int HTTP_CONFLICT;
  public final static int HTTP_CREATED;
  public final static int HTTP_ENTITY_TOO_LARGE;
  public final static int HTTP_FORBIDDEN;
  public final static int HTTP_GATEWAY_TIMEOUT;
  public final static int HTTP_GONE;
  public final static int HTTP_INTERNAL_ERROR;
  public final static int HTTP_LENGTH_REQUIRED;
  public final static int HTTP_MOVED_PERM;
  public final static int HTTP_MOVED_TEMP;
  public final static int HTTP_MULT_CHOICE;
  public final static int HTTP_NOT_ACCEPTABLE;
  public final static int HTTP_NOT_AUTHORITATIVE;
  public final static int HTTP_NOT_FOUND;
  public final static int HTTP_NOT_MODIFIED;
  public final static int HTTP_NO_CONTENT;
  public final static int HTTP_OK;
  public final static int HTTP_PARTIAL;
  public final static int HTTP_PAYMENT_REQUIRED;
  public final static int HTTP_PRECON_FAILED;
  public final static int HTTP_PROXY_AUTH;
  public final static int HTTP_REQ_TOO_LONG;
  public final static int HTTP_RESET;
  public final static int HTTP_SEE_OTHER;
  public final static int HTTP_SERVER_ERROR;
  public final static int HTTP_UNAUTHORIZED;
  public final static int HTTP_UNAVAILABLE;
  public final static int HTTP_UNSUPPORTED_TYPE;
  public final static int HTTP_USE_PROXY;
  public final static int HTTP_VERSION;
  // Variables
  protected String method;
  protected int responseCode;
  protected String responseMessage;
  // Constructors
  protected HttpURLConnection(URL u);
  // Class Methods
  public static boolean getFollowRedirects();
  public static void setFollowRedirects(boolean set);
  // Instance Methods
  public abstract void disconnect();
  public String getRequestMethod();
  public int getResponseCode();
  public String getResponseMessage();
  public void setRequestMethod(String method);
  public abstract boolean usingProxy();
}
```

Constants

HTTP_ACCEPTED

public final static int HTTP_ACCEPTED = 202

Description

> The HTTP response code that means the request has been accepted by the server.

HTTP_BAD_GATEWAY

public final static int HTTP_BAD_GATEWAY = 502

Description

> The HTTP response code that means the server, acting as a gateway, has received a bad response from another server.

HTTP_BAD_METHOD

public final static int HTTP_BAD_METHOD = 405

Description

> The HTTP response code that means the requested method is not allowed for the requested resource.

HTTP_BAD_REQUEST

public final static int HTTP_BAD_REQUEST = 400

Description

> The HTTP response code that means the request was syntactically incorrect.

HTTP_CLIENT_TIMEOUT

public final static int HTTP_CLIENT_TIMEOUT = 408

Description

> The HTTP response code that means the server has not received a request from the client in the time it expected.

HTTP_CONFLICT

public final static int HTTP_CONFLICT = 409

Description

The HTTP response code that means there is a conflict with the state of the requested resource.

HTTP_CREATED

public final static int HTTP_CREATED = 201

Description

The HTTP response code that means a new resource has been created as the result of the request.

HTTP_ENTITY_TOO_LARGE

public final static int HTTP_ENTITY_TOO_LARGE = 413

Description

The HTTP response code that means the request contains an entity that is too large for the server.

HTTP_FORBIDDEN

public final static int HTTP_FORBIDDEN = 403

Description

The HTTP response code that means the client does not have permission to access the requested resource.

HTTP_GATEWAY_TIMEOUT

public final static int HTTP_GATEWAY_TIMEOUT = 504

Description

The HTTP response code that means the server, acting as a gateway, has not received a response from an upstream server in the time it expected.

HTTP_GONE

public final static int HTTP_GONE = 410

Description

The HTTP response code that means the requested resource is no longer available.

HTTP_INTERNAL_ERROR

public final static int HTTP_INTERNAL_ERROR = 501

Description

The HTTP response code that means the server does not or cannot support the client's request.

HTTP_LENGTH_REQUIRED

public final static int HTTP_LENGTH_REQUIRED = 411

Description

> The HTTP response code that means the server won't accept the request without a length indication.

HTTP_MOVED_PERM

public final static int HTTP_MOVED_PERM = 301

Description

> The HTTP response code that means the requested resource has moved permanently.

HTTP_MOVED_TEMP

public final static int HTTP_MOVED_TEMP = 302

Description

> The HTTP response code that means the requested resource has moved temporarily.

HTTP_MULT_CHOICE

public final static int HTTP_MULT_CHOICE = 300

Description

> The HTTP response code that means the requested resource is available in multiple representations.

HTTP_NOT_ACCEPTABLE

public final static int HTTP_NOT_ACCEPTABLE = 406

Description

> The HTTP response code that means the requested resource is not available in a representation that is acceptable to the client.

HTTP_NOT_AUTHORITATIVE

public final static int HTTP_NOT_AUTHORITATIVE = 203

Description

> The HTTP response code that means the information returned may be a copy.

HTTP_NOT_FOUND

public final static int HTTP_NOT_FOUND = 404

Description

The HTTP response code that means the server could not find the requested resource.

HTTP_NOT_MODIFIED

public final static int HTTP_NOT_MODIFIED = 304

Description

The HTTP response code that means the requested resource has not been modified.

HTTP_NO_CONTENT

public final static int HTTP_NO_CONTENT = 204

Description

The HTTP response code that means the request has been processed, but there is no new information.

HTTP_OK

public final static int HTTP_OK = 200

Description

The HTTP response code that means the request succeeded.

HTTP_PARTIAL

public final static int HTTP_PARTIAL = 206

Description

The HTTP response code that means the partial request has been fulfilled.

HTTP_PAYMENT_REQUIRED

public final static int HTTP_PAYMENT_REQUIRED = 402

Description

An HTTP response code that is reserved for future use.

HTTP_PRECON_FAILED

public final static int HTTP_PRECON_FAILED = 412

Description

The HTTP response code that means the precondition in the request has failed.

HTTP_PROXY_AUTH

public final static int HTTP_PROXY_AUTH = 407

Description

The HTTP response code that means the client needs to authenticate itself with the proxy.

HTTP_REQ_TOO_LONG

public final static int HTTP_REQ_TOO_LONG = 414

Description

The HTTP response code that means the client request is too long.

HTTP_RESET

public final static int HTTP_RESET = 205

Description

The HTTP response code that means the request has been fulfilled, and the client should reset its view.

HTTP_SEE_OTHER

public final static int HTTP_SEE_OTHER = 303

Description

The HTTP response code that means the requested resource is available at another URL.

HTTP_SERVER_ERROR

public final static int HTTP_SERVER_ERROR = 500

Description

The HTTP response code that means the server encountered a problem and could not fulfill the request.

HTTP_UNAUTHORIZED

public final static int HTTP_UNAUTHORIZED = 401

Description

The HTTP response code that means the client is not authenticated for the requested resource.

HTTP_UNAVAILABLE

public final static int HTTP_UNAVAILABLE = 503

Description

The HTTP response code that means the server is temporarily unable to fulfill the request.

HTTP_UNSUPPORTED_TYPE

public final static int HTTP_UNSUPPORTED_TYPE = 415

Description

The HTTP response code that means the server cannot process the type of the request.

HTTP_USE_PROXY

public final static int HTTP_USE_PROXY = 305

Description

The HTTP response code that means a proxy must be used to access the requested resource.

HTTP_VERSION

public final static int HTTP_VERSION = 505

Description

The HTTP response code that means the server does not support the HTTP version used in the request.

Variables

method

protected String method = "GET"

Description

The method of this request. Valid values are: `"DELETE"`, `"GET"`, `"HEAD"`, `"OPTIONS"`, `"POST"`, `"PUT"`, and `"TRACE"`.

responseCode

protected int responseCode = -1

Description

The response code from the server, which may be one of the `HTTP_` constants.

responseMessage

protected String responseMessage = null

Description

The response message from the server that corresponds to `responseCode`.

Constructors

HttpURLConnection

protected HttpURLConnection(URL u)

Parameters

`u`

A `URL` object that represents a resource.

Description

This constructor creates an `HttpURLConnection` for the given `URL` object.

Class Methods

getFollowRedirects

public static boolean getFollowRedirects()

Returns

A `boolean` value that indicates whether or not HTTP redirect codes are automatically followed.

Description

This method indicates whether or not this `HttpURLConnection` follows HTTP redirects. The default value is `false`.

setFollowRedirects

public static void setFollowRedirects(boolean set)

Parameters

set

A boolean value that specifies whether or not HTTP redirects should be followed.

Throws

SecurityException

If there is a SecurityManager installed and its checkSetFactory() method throws a SecurityException.

Description

This method specifies whether or not this HttpURLConnection follows HTTP redirects.

Instance Methods

disconnect

public abstract void disconnect()

Description

This method closes the connection to the server.

getRequestMethod

public String getRequestMethod()

Returns

The method of this request.

Description

This method returns the method of this request.

getResponseCode

public int getResponseCode() throws IOException

Returns

The response code from the server.

Throws

IOException

If any kind of I/O error occurs.

Description

This method returns the code the server sent in response to this request. For example, suppose the server response is:

```
HTTP/1.0 404 Not Found
```

In this case, the method returns integer value `404`.

getResponseMessage

public int getResponseMessage() throws IOException

Returns

The response message from the server.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method returns the message the server sent in response to this request. For example, suppose the server response is:

```
HTTP/1.0 404 Not Found
```

In this case, the method returns the string `"Not Found"`.

setRequestMethod

`public void setRequestMethod(String method) throws ProtocolException`

Parameters

`method`

The new method for this request.

Throws

`ProtocolException`

If the connection has already been made or if `method` is invalid.

Description

This method sets the method of this request. Valid values are: `"DELETE"`, `"GET"`, `"HEAD"`, `"OPTIONS"`, `"POST"`, `"PUT"`, and `"TRACE"`.

usingProxy

public abstract boolean usingProxy()

Returns

A `boolean` value that indicates whether or not this `HttpURLConnection` is using a proxy.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method returns a flag that indicates if this connection is going through a proxy or not.

Inherited Variables

Variable	Inherited From	Variable	Inherited From
`allowUserInteraction`	`URLConnection`	`connected`	`URLConnection`
`doInput`	`URLConnection`	`doOutput`	`URLConnection`
`ifModifiedSince`	`URLConnection`	`url`	`URLConnection`
`useCaches`	`URLConnection`		

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`connect()`	`URLConnection`
`equals(Object)`	`Object`	`finalize()`	`Object`
`getAllowUserInteraction()`	`URLConnection`	`getClass()`	`Object`
`getContent()`	`URLConnection`	`getContentEncoding()`	`URLConnection`
`getContentLength()`	`URLConnection`	`getContentType()`	`URLConnection`
`getDate()`	`URLConnection`	`getDefaultUseCaches()`	`URLConnection`
`getDoInput()`	`URLConnection`	`getDoOutput()`	`URLConnection`
`getExpiration()`	`URLConnection`	`getHeaderField(String)`	`URLConnection`
`getHeaderField(int)`	`URLConnection`	`getHeaderFieldDate(String, long)`	`URLConnection`
`getHeaderFieldInt(String, int)`	`URLConnection`	`getHeaderFieldKey(int)`	`URLConnection`
`getIfModifiedSince()`	`URLConnection`	`getInputStream()`	`URLConnection`
`getLastModified()`	`URLConnection`	`getOutputStream()`	`URLConnection`

`getRequestProperty(String)`	`URLConnection`	`getURL()`	`URLConnection`
`getUseCaches()`	`URLConnection`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`setAllowUserInteraction(boolean)`	`URLConnection`	`setDefaultUseCaches(boolean)`	`URLConnection`
`setDoInput(boolean)`	`URLConnection`	`setDoOutput(boolean)`	`URLConnection`
`setIfModifiedSince(long)`	`URLConnection`	`setRequestProperty(String, String)`	`URLConnection`
`setUseCaches(boolean)`	`URLConnection`	`toString()`	`URLConnection`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`IOException`, `ProtocolException`, `SecurityException`, `SecurityManager`, `URL`, `URLConnection`

PREVIOUS HOME NEXT
FileNameMap BOOK INDEX InetAddress

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

InetAddress

Name

InetAddress

Synopsis

Class Name:

`java.net.InetAddress`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

`java.io.Serializable`

Availability:

JDK 1.0 or later

Description

The `InetAddress` class encapsulates an Internet Protocol (IP) address. `InetAddress` objects are used by the various classes that are responsible for specifying the destination addresses of outbound network packets, such as `DatagramSocket`, `MulticastSocket`, and `Socket`. `InetAddress` does not provide any `public` constructors. Instead, you must use the `static` methods `getAllByName()`, `getByName()`, and `getLocalHost()` to create `InetAddress` objects.

Class Summary

```
public final class java.net.InetAddress extends java.lang.Object
                   implements java.io.Serializable {
  // Class Methods
  public static InetAddress[] getAllByName(String host);
  public static InetAddress getByName(String host);
  public static InetAddress getLocalHost();
  // Instance Methods
  public boolean equals(Object obj);
  public byte[] getAddress();
  public String getHostAddress();                // New in 1.1
  public String getHostName();
  public int hashCode();
  public boolean isMulticastAddress();           // New in 1.1
  public String toString();
}
```

Class Methods

getAllByName

`public static InetAddress[] getAllByName(String host) throws UnknownHostException`

Parameters

`host`

A `String` that contains a hostname.

Returns

An array of `InetAddress` objects that corresponds to the given name.

Throws

`SecurityException`

If the application is not allowed to connect to `host`.

`UnknownHostException`

If `host` cannot be resolved.

Description

This method finds all of the IP addresses that correspond to the given hostname. The hostname can be a machine name, such as `"almond.nuts.com"`, or a string representation of an IP address, such as `"208.25.146.1"`.

getByName

```
public static InetAddress getByName(String host) throws UnknownHostException
```

Parameters

`host`

A `String` that contains a host name.

Returns

An `InetAddress` that corresponds to the given name.

Throws

`SecurityException`

If the application is not allowed to connect to `host`.

`UnknownHostException`

If `host` cannot be resolved.

Description

This method returns the primary IP address that correspond to the given hostname. The hostname can be a machine name, such as `"almond.nuts.com"`, or a string representation of an IP address, such as `"208.25.146.1"`.

getLocalHost

```
public static InetAddress getLocalHost() throws UnknownHostException
```

Returns

An `InetAddress` that corresponds to the name of the local machine.

Throws

`SecurityException`

If the application is not allowed to connect to `host`.

`UnknownHostException`

If `host` cannot be resolved.

Description

This method finds the IP address of the local machine.

Instance Methods

equals

public boolean equals(Object obj)

Parameters

`obj`

The object to be compared with this object.

Returns

`true` if the objects are equivalent; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `obj` is an instance of `InetAddress` that specifies the same IP address as the object this method is associated with.

getAddress

public byte[] getAddress()

Returns

A byte array with elements that correspond to the bytes of the IP address that this object represents.

Description

This method returns the IP address associated with this object as an array of bytes in network order. That means that the first element of the array contains the highest order byte, and the last element of the array contains the lowest order byte.

getHostAddress

public String getHostAddress()

Availability

New as of JDK 1.1

Returns

A `String` that contains the IP address of this object.

Description

This method returns a string representation of the IP address associated with this object. For example: `"206.175.64.78"`.

getHostName

public String getHostName()

Returns

The hostname associated with this object.

Description

In most cases, this method returns the hostname that corresponds to the IP address associated with this object. However, there are a few special cases:

- If the address associated with this object is address of the local machine, the method may return `null`.
- If the method cannot determine a home name to go with the address associated with this object, the method returns a string representation of the address.
- If the application is not allowed to know the hostname, the method returns a string representation of the address.

hashCode

public int hashCode()

Returns

The hashcode based on the IP address of the object.

Overrides

`Object.hashCode()`

Description

This method returns a hashcode for this object, based on the IP address associated with this object.

isMulticastAddress

public boolean isMulticastAddress()

Availability

New as of JDK 1.1

Returns

`true` if this object represents a multicast address; `false` otherwise.

Description

This method returns a flag that indicates if this object represents an IP multicast address. A multicast address is a Class D address, which means that its four highest-order bits are set to 1110. In other words, multicast addresses are in the range 224.0.0.1 through 239.255.255.255 inclusive.

toString

public String toString()

Returns

The string representation of this `InetAddress`.

Overrides

`Object.toString()`

Description

This method returns a `String` that contains both the hostname and IP address of this object.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`notify()`	`Object`

`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`DatagramSocket`, `MulticastSocket`, `SecurityException`, `Serializable`, `Socket`, `UnknownHostException`

PREVIOUS HOME NEXT
HttpURLConnection BOOK INDEX MalformedURLException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

MalformedURLException

Name

MalformedURLException

Synopsis

Class Name:

```
java.net.MalformedURLException
```

Superclass:

```
java.io.IOException
```

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

A `MalformedURLException` is thrown when a malformed URL is encountered, which can occur if a URL does not contain a valid protocol or if the string is unparsable.

Class Summary

```
public class java.net.MalformedURLException extends java.io.IOException {
  // Constructors
  public MalformedURLException();
  public MalformedURLException(String msg);
}
```

Constructors

MalformedURLException

public MalformedURLException()

Description

This constructor `create`s a `MalformedURLException` with no associated detail message.

public MalformedURLException(String msg)

Parameters

`msg`

The detail message.

Description

This constructor creates a `MalformedURLException` with the specified detail message.

Inherited Methods

Method	**Inherited From**	**Method**	**Inherited From**
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`

`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Throwable`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Exception`, `IOException`, `RuntimeException`

MulticastSocket

Name

MulticastSocket

Synopsis

Class Name:

`java.net.MulticastSocket`

Superclass:

`java.net.DatagramSocket`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `MulticastSocket` class implements packet-oriented, connectionless, multicast data communication. In Internet parlance, this is the User Datagram Protocol (UDP) with additional functionality for joining and leaving groups of other multicast hosts on the Internet. A multicast group is specified by a Class D address, which means that the four highest-order bits are set to 1110. In other words, multicast addresses are in the range 224.0.0.1 through 239.255.255.255 inclusive.

`MulticastSocket` inherits most of its functionality from `DatagramSocket`; it adds the ability to join and leave multicast groups. When a `MulticastSocket` joins a group, it receives all of the packets destined for the group. Any `DatagramSocket` or `MulticastSocket` can send packets to a multicast group.

Class Summary

```
public final class java.net.MulticastSocket
                   extends java.net.DatagramSocket {
  // Constructors
  public MulticastSocket();
  public MulticastSocket(int port);
  // Instance Methods
  public InetAddress getInterface();
  public byte getTTL();
  public void joinGroup(InetAddress mcastaddr);
  public void leaveGroup(InetAddress mcastaddr)
  public synchronized void send(DatagramPacket p, byte ttl);
  public void setInterface(InetAddress inf);
  public void setTTL(byte ttl);
}
```

Constructors

MulticastSocket

public MulticastSocket() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

`SecurityException`

If the application is not allowed to listen on the port.

Description

This constructor creates a `MulticastSocket` that is bound to any available port on the local host machine.

public MulticastSocket(int port) throws IOException

Parameters

`port`

A port number.

Throws

`IOException`

If any kind of I/O error occurs.

`SecurityException`

If the application is not allowed to listen on the given port.

Description

This constructor creates a `MulticastSocket` that is bound to the given port on the local host machine.

Instance Methods

getInterface

public InetAddress getInterface() throws SocketException

Returns

The address of the network interface used for outgoing multicast packets.

Throws

`SocketException`

If any kind of socket error occurs.

Description

This method returns the IP address that this `MulticastSocket` uses to send out packets to multicast destinations.

getTTL

public byte getTTL() throws IOException

Returns

> The time-to-live (TTL) value for this socket.

Throws

> `IOException`
>
> > If any kind of I/O error occurs.

Description

> This method returns the TTL value for this socket. This value is the number of hops an outgoing packet can traverse before it is discarded.

joinGroup

public void joinGroup(InetAddress mcastaddr) throws IOException

Parameters

> `mcastaddr`
>
> > The IP address of the group to join.

Throws

> `IOException`
>
> > If any kind of I/O error occurs.
>
> `SecurityException`
>
> > If the application is not allowed to access the given multicast address.

Description

> This method is used to join a multicast group. An exception is thrown if the given address is not a multicast address. While the socket is part of a group, it receives all the packets that are sent to the group.

leaveGroup

public void leaveGroup(InetAddress mcastaddr) throws IOException

Parameters

`mcastaddr`

The IP address of the group to leave.

Throws

`IOException`

If any kind of I/O error occurs.

`SecurityException`

If the application is not allowed to access the given multicast address.

Description

This method is used to leave a multicast group. An exception is thrown if the given address is not a multicast address.

send

```
public synchronized void send(DatagramPacket p, byte ttl) throws IOException
```

Parameters

`p`

The `DatagramPacket` to be sent.

`ttl`

The time-to-live (TTL) value for this packet.

Throws

`IOException`

If any kind of I/O error occurs.

`SecurityException`

If the application is not allowed to send data to the packet's destination.

Description

This method sends a packet from this socket using the given TTL value. The packet data, packet length, destination address, and destination port number are specified by the given `DatagramPacket`.

Generally, it is easier to use `setTTL()` to set the TTL value for the socket, then use `send(DatagramPacket)` to send data. This method is provided for special cases.

setInterface

public void setInterface(InetAddress inf) throws SocketException

Parameters

`inf`

The new address of the network interface for multicast packets.

Throws

`SocketException`

If any kind of socket error occurs.

Description

This method is used to set the address that is used for outgoing multicast packets.

setTTL

public void setTTL(byte ttl) throws IOException

Parameters

`ttl`

The new time-to-live (TTL) value for this socket.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method is used to set the TTL value of the socket. The TTL value is the number of hops an outgoing packet can traverse before it is discarded.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`close()`	`DatagramSocket`
`equals(Object)`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalAddress()`	`DatagramSocket`
`getLocalPort()`	`DatagramSocket`	`getSoTimeout()`	`DatagramSocket`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`receive(DatagramPacket)`	`DatagramSocket`
`send(DatagramPacket)`	`DatagramSocket`	`setSoTimeout(int)`	`DatagramSocket`
`toString()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`DatagramPacket`, `DatagramSocket`, `DatagramSocketImpl`, `InetAddress`, `IOException`, `SecurityException`, `SocketException`

NoRouteToHostException

Name

NoRouteToHostException

Synopsis

Class Name:

    `java.net.NoRouteToHostException`

Superclass:

    `java.net.SocketException`

Immediate Subclasses:

    None

Interfaces Implemented:

    None

Availability:

    New as of JDK 1.1

Description

A `NoRouteToHostException` is thrown when a socket connection cannot be established with a remote host.

This type of exception usually indicates that a firewall is blocking access to the host, or that an intermediate router is down.

Class Summary

```
public class java.net.NoRouteToHostException
             extends java.net.SocketException {
  // Constructors
  public NoRouteToHostException();
  public NoRouteToHostException(String msg);
}
```

Constructors

NoRouteToHostException

public NoRouteToHostException()

Description

This constructor `create`s a `NoRouteToHostException` with no associated detail message.

public NoRouteToHostException(String msg)

Parameters

`msg`

The detail message.

Description

This constructor creates a `NoRouteToHostException` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`

`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Throwable`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Exception`, `IOException`, `RuntimeException`, `SocketException`

PREVIOUS HOME NEXT
MulticastSocket BOOK INDEX ProtocolException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

ProtocolException

Name

ProtocolException

Synopsis

Class Name:

`java.net.ProtocolException`

Superclass:

`java.io.IOException`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

A `ProtocolException` is thrown to indicate that there has been an error in an underlying protocol, such as an HTTP or TCP protocol error.

Class Summary

```
public class java.net.ProtocolException extends java.io.IOException {
  // Constructors
  public ProtocolException();
  public ProtocolException(String host);
}
```

Constructors

ProtocolException

public ProtocolException()

Description

> This constructor `create`s a `ProtocolException` with no associated detail message.

public ProtocolException(String host)

Parameters

> `host`
>
> > The detail message.

Description

> This constructor creates a `ProtocolException` with the specified detail message, which should be the host that caused the underlying protocol error.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`

`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Throwable`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Exception`, `IOException`, `RuntimeException`

PREVIOUS | HOME | NEXT

NoRouteToHostException | BOOK INDEX | ServerSocket

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

ServerSocket

Name

ServerSocket

Synopsis

Class Name:

`java.net.ServerSocket`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The `ServerSocket` class represents a socket that listens for connection requests from clients on a specified port. When a connection is requested, a `Socket` object is created to handle the communication.

The low-level network access in `ServerSocket` is provided by a subclass of the `abstract` class `SocketImpl`. An application can change the server socket factory that creates the `SocketImpl` subclass by supplying a `SocketImplFactory` using the `setSocketFactory()` method. This feature allows an application to create sockets that are appropriate for the local network configuration and accommodate such things as firewalls.

Class Summary

```
public class java.net.ServerSocket extends java.lang.Object {
  // Constructors
  public ServerSocket(int port);
  public ServerSocket(int port, int backlog);
  public ServerSocket(int port, int backlog,
                      InetAddress bindAddr);              // New in 1.1
  // Class Methods
  public static synchronized void setSocketFactory(SocketImplFactory fac);
  // Instance Methods
  public Socket accept();
  public void close();
  public InetAddress getInetAddress();
  public int getLocalPort();
  public synchronized int getSoTimeout()                  // New in 1.1
  public synchronized void setSoTimeout(int timeout);     // New in 1.1
  public String toString();
  // Protected Instance Methods
  protected final void implAccept(Socket s);              // New in 1.1
}
```

Constructors

ServerSocket

public ServerSocket(int port) throws IOException

Parameters

`port`

A port number, or 0 for any available port.

Throws

`IOException`

If any kind of I/O error occurs.

`SecurityException`

If the application is not allowed to listen on the given port.

Description

This method creates a server socket that listens for a connection on the given port. A default of 50 pending connections can be queued by the `ServerSocket`. Calling `accept()` removes a pending connections from the queue. If the queue is full, additional connection requests are refused.

If an application has specified a server socket factory, the `createSocketImpl()` method of that factory is called to create the actual socket implementation. Otherwise, the constructor creates a plain socket.

public ServerSocket(int port, int backlog) throws IOException

Parameters

`port`

A port number, or 0 for any available port.

`backlog`

The maximum length of the pending connections queue.

Throws

`IOException`

If any kind of I/O error occurs.

`SecurityException`

If the application is not allowed to listen on the given port.

Description

This method creates a server socket that listens for a connection on the given port. The `backlog` parameter specifies how many pending connections can be queued by the `ServerSocket`. Calling `accept()` removes a pending connection from the queue. If the queue is full, additional connection requests are refused.

If an application has specified a server socket factory, the `createSocketImpl()` method of that factory is called to create the actual socket implementation. Otherwise, the constructor creates a plain socket.

`public ServerSocket(int port, int backlog, InetAddress bindAddr) throws IOException`

Availability

New as of JDK 1.1

Parameters

`port`

A port number, or 0 for any available port.

`backlog`

The maximum length of the pending connections queue.

`bindAddr`

A local address.

Throws

`IOException`

If any kind of I/O error occurs.

`SecurityException`

If the application is not allowed to listen on the given port.

Description

This method creates a server socket that listens for a connection on the given port of `bindAddr`. On machines with multiple addresses, `bindAddr` specifies the address on which this `ServerSocket` listens. The `backlog` parameter specifies how many pending connections can be queued by the `ServerSocket`. Calling `accept()` removes pending connections from the queue. If the queue is full, additional connection requests are refused.

If an application has specified a server socket factory, the `createSocketImpl()` method of that factory is called to create the actual socket implementation. Otherwise, the constructor creates a plain socket.

Class Methods

setSocketFactory

`public static synchronized void setSocketFactory( SocketImplFactory fac) throws IOException`

Parameters

`fac`

An object that implements `SocketImplFactory`.

Throws

`IOException`

If the factory has already been defined.

`SecurityException`

If the application does not have permission to set the factory.

Description

This method is used to set the `SocketImplFactory`. This factory object produces instances of subclasses of `SocketImpl` that do the low-level work of server sockets. When a `ServerSocket` constructor is called, the `createSocketImpl()` method of the factory is called to create the server socket implementation.

Instance Methods

accept

public Socket accept() throws IOException

Returns

A `Socket` object for the connection.

Throws

`IOException`

If any kind of I/O error occurs.

`SecurityException`

If the application is not allowed to accept the connection.

Description

This method accepts a connection and returns a `Socket` object to handle communication. If there are pending connections, the method accepts a pending connection from the queue and returns immediately. Otherwise, the method may block until a connection is requested. If a time-out value has been specified using the `setSoTimeout()` method, `accept()` may time out and throw an `InterruptedIOException` if no connection is requested in the time-out period.

close

public void close() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Description

> This method closes this server socket, releasing any system resources it holds.

getInetAddress

public InetAddress getInetAddress()

Returns

> The IP address to which this `ServerSocket` is listening.

Description

> Generally, this method returns the address of the local host. However, a `ServerSocket` can be constructed with an alternate address using `ServerSocket(int, int, InetAddress)`. The method returns `null` if the socket is not yet connected.

getLocalPort

public int getLocalPort()

Returns

> The port number to which this `ServerSocket` is listening.

Description

> This method returns the port number to which this object is connected.

getSoTimeout

public synchronized int getSoTimeout() throws IOException

Availability

> New as of JDK 1.1

Returns

> The receive timeout value for the socket.

Throws

> `IOException`
>
> > If any kind of I/O error occurs.

Description

This method returns the receive time-out value for this socket. A value of zero indicates that `accept()` waits indefinitely for an incoming packet, while a non-zero value indicates the number of milliseconds it waits.

setSoTimeout

`public synchronized void setSoTimeout(int timeout) throws SocketException`

Availability

New as of JDK 1.1

Parameters

`timeout`

The new time-out value, in milliseconds, for this socket.

Throws

`SocketException`

If any kind of socket error occurs.

Description

This method is used to set the time-out value that is used for `accept()`. A nonzero value is the length of time, in milliseconds, the `ServerSocket` should wait for a connection. A value of zero indicates that the `ServerSocket` should wait indefinitely. If a time-out value is needed, this method must be called before `accept()`.

toString

public String toString()

Returns

The string representation of this `ServerSocket`.

Overrides

`Object.toString()`

Description

This method returns a `String` that contains the address and port of this `ServerSocket`.

Protected Instance Methods

implAccept

protected final void implAccept(Socket s) throws IOException

Availability

New as of JDK 1.1

Parameters

`s`

The `Socket` object to be connected.

Throws

`IOException`

If any kind of I/O error occurs.

`SecurityException`

If the application is not allowed to accept the connection.

Description

This method is a helper method for `accept()`. It can be overridden in subclasses of `ServerSocket` to support new subclasses of `Socket`.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`InetAddress`, `IOException`, `SecurityException`, `Socket`, `SocketException`, `SocketImpl`, `SocketImplFactory`

PREVIOUS HOME NEXT
ProtocolException BOOK INDEX Socket

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Socket

Name

Socket

Synopsis

Class Name:

`java.net.Socket`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The `Socket` class implements stream-based, connection-oriented, reliable data communication. Although `Socket` objects are often used with the Transmission Control Protocol, commonly known as TCP, they are independent of the actual protocol being used. The `Socket` class encapsulates client logic that is common to connection-oriented protocols. Sockets are two-way data pipes that are connected on either end to an address and port number. As of JDK 1.1, new constructors allow you to specify the local address and port as well as the remote address and port.

A `Socket` object uses an object that belongs to a subclass of the `abstract` class `SocketImpl` to access protocol-specific logic. A program can specify the subclass of `SocketImpl` that is used by passing an appropriate `SocketImplFactory` object to the `setSocketImplFactory()` method before any `Socket` objects are created. This feature allows a program to create sockets that are able to accommodate such things as firewalls or even work with different protocols.

Class Summary

```
public class java.net.Socket extends java.lang.Object {
  // Constructors
  public Socket(String host, int port);
  public Socket(InetAddress address, int port);
  public Socket(String host, int port,
                InetAddress localAddr, int localPort);     // New in 1.1
  public Socket(InetAddress address, int port,
                InetAddress localAddr, int localPort);     // New in 1.1
  public Socket(String host, int port,
                boolean stream);                     // Deprecated in 1.1
  public Socket(InetAddress host, int port,
                boolean stream);                     // Deprecated in 1.1
  protected Socket();                                      // New in 1.1
  protected Socket(SocketImpl impl);                       // New in 1.1
  // Class Methods
  public static synchronized void setSocketImplFactory(
                                  SocketImplFactory fac);
  // Instance Methods
  public synchronized void close();
  public InetAddress getInetAddress();
  public InputStream getInputStream();
  public InetAddress getLocalAddress();                    // New in 1.1
  public int getLocalPort();
  public OutputStream getOutputStream();
  public int getPort();
  public int getSoLinger();                                // New in 1.1
  public synchronized int getSoTimeout();                  // New in 1.1
  public boolean getTcpNoDelay();                          // New in 1.1
  public void setSoLinger(boolean on, int val);            // New in 1.1
  public synchronized void setSoTimeout(int timeout);      // New in 1.1
  public void setTcpNoDelay(boolean on);                   // New in 1.1
  public String toString();
}
```

Constructors

Socket

`public Socket(String host, int port) throws IOException, UnknownHostException`

Parameters

`host`

The name of a remote machine.

`port`

A port on a remote machine.

Throws

`IOException`

If any kind of I/O error occurs.

`SecurityException`

If the application is not allowed to connect to the given host and port.

`UnknownHostException`

If the IP address of the given hostname cannot be determined.

Description

This constructor creates a `Socket` and connects it to the specified port on the given host.

If a program has specified a socket factory, the `createSocketImpl()` method of that factory is called to create the actual socket implementation. Otherwise, the constructor creates a plain socket.

public Socket(InetAddress address, int port) throws IOException

Parameters

`address`

The IP address of a remote machine.

`port`

A port on a remote machine.

Throws

`IOException`

If any kind of I/O error occurs.

`SecurityException`

If the application is not allowed to connect to the given address and port.

Description

This constructor creates a `Socket` and connects it to the specified port on the host at the given address.

If a program has specified a socket factory, the `createSocketImpl()` method of that factory is called to create the actual socket implementation. Otherwise, the constructor creates a plain socket.

`public Socket(String host, int port, InetAddress localAddr, int localPort) throws IOException`

Availability

New as of JDK 1.1

Parameters

`host`

The name of a remote machine.

`port`

A port on a remote machine.

`localAddr`

An IP address on the local host.

`localPort`

A port on the local host.

Throws

`IOException`

If any kind of I/O error occurs.

`SecurityException`

If the application is not allowed to connect to the given host and port.

Description

This constructor creates a `Socket` and connects it to the specified port on the given host. The constructor also binds the `Socket` to the specified local address and port.

If a program has specified a socket factory, the `createSocketImpl()` method of that factory is called to create the actual socket implementation. Otherwise, the constructor creates a plain socket.

`public Socket(InetAddress address, int port, InetAddress localAddr, int localPort) throws IOException`

Availability

New as of JDK 1.1

Parameters

`address`

The IP address of a remote machine.

`port`

A port on a remote machine.

`localAddr`

An IP address on the local host.

`localPort`

A port on the local host.

Throws

`IOException`

If any kind of I/O error occurs.

`SecurityException`

If the application is not allowed to connect to the given address and port.

Description

This constructor creates a `Socket` and connects it to the specified port on the host at the given address. The constructor also binds the `Socket` to the specified local address and port.

If a program has specified a socket factory, the `createSocketImpl()` method of that factory is called to create the actual socket implementation. Otherwise, the constructor creates a plain socket.

`public Socket(String host, int port, boolean stream) throws IOException`

Availability

Deprecated as of JDK 1.1

Parameters

`host`

The name of a remote machine.

`port`

A port on a remote machine.

`stream`

A `boolean` value that indicates if this socket is a stream socket.

Throws

`IOException`

If any kind of I/O error occurs.

`SecurityException`

If the application is not allowed to connect to the given host and port.

Description

This constructor creates a `Socket` and connects it to the specified port on the given host.

If the `stream` argument is `true`, a stream socket is created. Otherwise, a datagram socket is created. This constructor is deprecated as of JDK 1.1; use `DatagramSocket` for datagrams.

If a program has specified a socket factory, the `createSocketImpl()` method of that factory is called to create the actual socket implementation. Otherwise, the constructor creates a plain socket.

`public Socket(InetAddress address, int port, boolean stream) throws IOException`

Availability

Deprecated as of JDK 1.1

Parameters

`address`

The IP address of a remote machine.

`port`

A port on a remote machine.

`stream`

A `boolean` value that indicates if this socket is a stream socket.

Throws

`IOException`

If any kind of I/O error occurs.

`SecurityException`

If the application is not allowed to connect to the given host and port.

Description

This constructor creates a `Socket` and connects it to the specified port on the host at the given address.

If the `stream` argument is `true`, a stream socket is created. Otherwise, a datagram socket is created. This constructor is deprecated as of JDK 1.1; use `DatagramSocket` for datagrams.

If a program has specified a socket factory, the `createSocketImpl()` method of that factory is called to create the actual socket implementation. Otherwise, the constructor creates a plain socket.

protectedSocket()

Availability

New as of JDK 1.1

Description

This constructor creates a `Socket` that uses an instance of the system default `SocketImpl` subclass for its low-level network access.

protectedSocket(SocketImpl impl) throws SocketException

Availability

New as of JDK 1.1

Parameters

`impl`

The socket implementation to use.

Throws

`SocketException`

This exception is never thrown by this constructor.

Description

This constructor creates a `Socket` that uses the given object for its low-level network access.

Class Methods

setSocketImplFactory

`public static synchronized void setSocketImplFactory( SocketImplFactory fac) throws IOException`

Parameters

`fac`

An object that implements `SocketImplFactory`.

Throws

`IOException`

If the factory has already been defined.

`SecurityException`

If the application does not have permission to set the factory.

Description

This method sets the `SocketImplFactory`. This factory produces instances of subclasses of `SocketImpl` that do the low-level work of sockets. When a `Socket` constructor is called, the `createSocketImpl()` method of the factory is called to create the socket implementation.

Instance Methods

close

public synchronized void close() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method closes this socket, releasing any system resources it holds.

getInetAddress

public InetAddress getInetAddress()

Returns

The remote IP address to which this `Socket` is connected.

Description

This method returns the IP address of the remote host to which this socket is connected.

getInputStream

public InputStream getInputStream() throws IOException

Returns

An `InputStream` that wraps this socket.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method returns an `InputStream` that reads data from the socket.

getLocalAddress

public InetAddress getLocalAddress()

Availability

New as of JDK 1.1

Returns

The local IP address from which this `Socket` originates.

Description

This method returns the local address that is the origin of the socket.

getLocalPort

public int getLocalPort()

Returns

The local port number from which this `Socket` originates.

Description

This method returns the local port number that is the origin of the socket.

getOutputStream

public OutputStream getOutputStream() throws IOException

Returns

An `OutputStream` that wraps this socket.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method returns an `OutputStream` that sends data through the socket.

getPort

public int getPort()

Returns

The remote port number to which this `Socket` is connected.

Description

This method returns the port number of the remote host to which this socket is connected.

getSoLinger

public int getSoLinger() throws SocketException

Availability

New as of JDK 1.1

Returns

The close time-out value for the socket.

Throws

`SocketException`

If any kind of socket error occurs.

Description

This method returns the close time-out value for this socket. A value of `-1` or `0` indicates that `close()`closes the socket immediately. A value greater than `0` indicates the amount of time, in seconds, that `close()` blocks, waiting for unsent data to be sent.

getSoTimeout

public synchronized int getSoTimeout() throws SocketException

Availability

New as of JDK 1.1

Returns

The read time-out value for the socket.

Throws

`SocketException`

If any kind of socket error occurs.

Description

This method returns the read time-out value for this socket. A value of zero indicates that the `read()` method of the associated `InputStream` waits indefinitely for an incoming packet, while a non-zero value indicates the number of milliseconds it waits before throwing an `InterruptedIOException`.

getTcpNoDelay

public boolean getTcpNoDelay() throws SocketException

Availability

New as of JDK 1.1

Returns

`true` if Nagle's algorithm is disabled for this connection; `false` otherwise.

Throws

`SocketException`

If any kind of socket error occurs.

Description

This method indicates whether Nagle's algorithm is disabled for this socket or not. Said another way, it indicates whether the `TCPNODELAY` option is enabled or not.

In essence, Nagle's algorithm takes small outgoing packets that are closely spaced in time and combines them into larger packets. This improves overall efficiency, since each packet has a certain amount of overhead; however, it does so at the expense of immediacy.

setSoLinger

`public void setSoLinger(boolean on, int val) throws SocketException`

Availability

New as of JDK 1.1

Parameters

`on`

A `boolean` value that specifies whether or not `close()` blocks

`val`

The new close time-out value, in seconds, for this socket.

Throws

`SocketException`

If any kind of socket error occurs.

Description

This method sets the close timeout value for this socket. If `val` is `-1` or `0`, or if `on` is `false`, `close()` closes the socket immediately. If `on` is `true` and `val` is greater than 0, `val` indicates the amount of time, in seconds, that `close()` blocks, waiting for unsent data to be sent.

setSoTimeout

`public synchronized void setSoTimeout(int timeout) throws SocketException`

Availability

New as of JDK 1.1

Parameters

`timeout`

The new read time-out value, in milliseconds, for the socket.

Throws

`SocketException`

If any kind of socket error occurs.

Description

This method is used to set the time-out value that is used for the `read()` method of the corresponding `InputStream`. A non-zero value is the length of time, in milliseconds, that the `Socket` should wait for data before throwing an `InterruptedIOException`. A value of zero indicates that the `Socket` should wait indefinitely. If a timeout value is needed, this method must be called before `read()`.

setTcpNoDelay

public void setTcpNoDelay(boolean on) throws SocketException

Availability

New as of JDK 1.1

Parameters

`on`

A `boolean` value that specifies whether or not to disable Nagle's algorithm.

Throws

`SocketException`

If any kind of socket error occurs.

Description

This method specifies whether Nagle's algorithm is disabled for this socket or not. Said another way, it determines whether the `TCPNODELAY` option is enabled or not.

In essence, Nagle's algorithm takes small outgoing packets that are closely spaced in time and combines them into larger packets. This improves overall efficiency, since each packet has a certain amount of overhead; however, it does so at the expense of immediacy.

toString

public String toString()

Returns

The string representation of this `Socket`.

Overrides

`Object.toString()`

Description

This method returns a `String` that contains the address and port of this `Socket`.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`DatagramSocket`, `InetAddress`, `InputStream`, `IOException`, `OutputStream`, `SecurityException`, `SocketException`, `SocketImpl`, `SocketImplFactory`, `UnknownHostException`

SocketException

Name

SocketException

Synopsis

Class Name:

`java.net.SocketException`

Superclass:

`java.io.IOException`

Immediate Subclasses:

`java.net.BindException`,

`java.net.ConnectException`,

`java.net.NoRouteToHostException`

Interfaces Implemented:

None

Availability:

JDK 1.0 and later

Description

A `SocketException` is thrown when an error occurs while a socket is being used. As of JDK 1.1, there are some more specific subclasses of `SocketException`, namely `BindException`, `ConnectException`, and `NoRouteToHostException`.

Class Summary

```
public class java.net.SocketException extends java.io.IOException {
  // Constructors
  public SocketException();
  public SocketException(String msg);
}
```

Constructors

SocketException

public SocketException()

Description

> This constructor `create`s a `SocketException` with no associated detail message.

public SocketException(String msg)

Parameters

> `msg`
>
> > The detail message.

Description

> This constructor creates a `SocketException` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`

fillInStackTrace()	Throwable	finalize()	Object
getClass()	Object	getLocalizedMessage()	Throwable
getMessage()	Throwable	hashCode()	Object
notify()	Object	notifyAll()	Object
printStackTrace()	Throwable	printStackTrace(PrintStream)	Throwable
printStackTrace(PrintWriter)	Throwable	toString()	Throwable
wait()	Object	wait(long)	Object
wait(long, int)	Object		

See Also

BindException, ConnectException, Exception, IOException, NoRouteToHostException, RuntimeException

PREVIOUS HOME NEXT
Socket BOOK INDEX SocketImpl

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

SocketImpl

Name

SocketImpl

Synopsis

Class Name:

`java.net.SocketImpl`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The `SocketImpl` class is an `abstract` class that defines the bulk of the methods that make the `Socket` and `ServerSocket` classes work. Thus, `SocketImpl` is used to create both client and server sockets. Non-`public` subclasses of `SocketImpl` provide platform-specific implementations of stream-based socket communication. A plain socket implements the methods in `SocketImpl` as described; other implementations could provide socket communication through a proxy or firewall.

Class Summary

```
public abstract class java.net.SocketImpl extends java.lang.Object {
  // Variables
  protected InetAddress address;
  protected FileDescriptor fd;
  protected int localport;
  protected int port;
  // Instance Methods
  public String toString();
  // Protected Instance Methods
  protected abstract void accept(SocketImpl s);
  protected abstract int available();
  protected abstract void bind(InetAddress host, int port);
  protected abstract void close();
  protected abstract void connect(String host, int port);
  protected abstract void connect(InetAddress address, int port);
  protected abstract void create(boolean stream);
  protected FileDescriptor getFileDescriptor();
  protected InetAddress getInetAddress();
  protected abstract InputStream getInputStream();
  protected int getLocalPort();
  protected abstract OutputStream getOutputStream();
  protected int getPort();
  protected abstract void listen(int backlog);
}
```

Variables

address

protected InetAddress address

Description

> The remote IP address to which this socket is connected.

fd

protected FileDescriptor fd

Description

> The file descriptor that represents this socket.

localPort

protected int localPort

Description

The local port number of this socket.

port

protected int port

Description

The remote port number of this socket.

Instance Methods

toString

public String toString()

Returns

The string representation of this `SocketImpl`.

Overrides

`Object.toString()`

Description

This method returns a `String` that contains a representation of this object.

Protected Instance Methods

accept

protected abstract void accept(SocketImpl s) throws IOException

Parameters

`s`

A `SocketImpl` to connect.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method accepts a connection. The method connects the given socket `s` to a remote host in response to the remote host's connection request on this `SocketImpl`.

available

protected abstract int available() throws IOException

Returns

The number of bytes that can be read without blocking.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method returns the number of bytes that can be read from the socket without waiting for more data to arrive.

bind

```
protected abstract void bind(InetAddress host, int port) throws IOException
```

Parameters

`host`

An IP address.

`port`

A port number.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method binds the socket to the given address and port. If the address or the port is unavailable, an exception is thrown.

close

protected abstract void close() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method closes the socket, releasing any system resources it holds.

connect

`protected abstract void connect(String host, int port) throws IOException`

Parameters

`host`

A remote hostname.

`port`

A port number on the remote host.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method connects this socket to the specified port on the given host.

`protected abstract void connect(InetAddress address, int port) throws IOException`

Parameters

`address`

A remote IP address.

`port`

A port number on the remote host.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method connects this socket to the specified port on the host at the given address.

create

protected abstract void create(boolean stream) throws IOException

Parameters

`stream`

A `boolean` value that indicates if this socket is a stream socket.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method creates a socket that is not bound and not connected. If the `stream` argument is `true`, a stream socket is created. Otherwise, a datagram socket is created.

getFileDescriptor

protected final FileDescriptor getFileDescriptor

Returns

The file descriptor for this socket.

Description

This method returns the file descriptor associated with this `SocketImpl`.

getInetAddress

protected InetAddress getInetAddress()

Returns

The remote IP address to which this `SocketImpl` is connected.

Description

This method returns the IP address of the remote host to which this `SocketImpl` is connected.

getInputStream

protected abstract InputStream getInputStream() throws IOException

Returns

An `InputStream` that wraps this socket.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method returns an `InputStream` that reads data from the socket.

getLocalPort

protected int getLocalPort()

Returns

The local port number from which this `SocketImpl` originates.

Description

This method returns the local port number that is the origin of the socket.

getOutputStream

`protected abstract OutputStream getOutputStream() throws IOException`

Returns

An `OutputStream` that wraps this socket.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method returns an `OutputStream` that sends data through the socket.

getPort

protected int getPort()

Returns

The remote port number to which this `SocketImpl` is connected.

Description

This method returns the port number of the remote host to which this socket is connected.

listen

protected abstract void listen(int backlog) throws IOException

Parameters

`backlog`

The maximum length of pending connections queue.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This object can directly accept a connection if its `accept()` method has been called and is waiting for a connection. Otherwise, the local system rejects connections to this socket unless `listen()` has been called.

This method requests that the local system listen for connections and accept them on behalf of this object. The accepted connections are placed in a queue of the specified length. When there are connections in the queue, a call to this object's `accept()` method removes a connection from the queue and immediately returns. If the queue is full, additional connection requests are refused.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`FileDescriptor`, `InetAddress`, `InputStream`, `IOException`, `OutputStream`, `ServerSocket`, `Socket`, `SocketImplFactory`

PREVIOUS | HOME | NEXT
SocketException | BOOK INDEX | SocketImplFactory

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

SocketImplFactory

Name

SocketImplFactory

Synopsis

Interface Name:

```
java.net.SocketImplFactory
```

Super-interface:

None

Immediate Sub-interfaces:

None

Implemented By:

None

Availability:

JDK 1.0 or later

Description

The SocketImplFactory interface defines a method that creates a SocketImpl object. The interface is implemented by classes that select SocketImpl subclasses to support the Socket and ServerSocket classes.

Interface Declaration

```
public abstract interface java.net.SocketImplFactory {
  // Methods
  public abstract SocketImpl createSocketImpl();
}
```

Methods

createSocketImpl

public abstract SocketImpl createSocketImpl()

Returns

> A SocketImpl object.

Description

> This method creates an instance of a subclass of SocketImpl that is appropriate for the environment.

See Also

ServerSocket, Socket, SocketImpl

PREVIOUS
SocketImpl

HOME
BOOK INDEX

NEXT
URL

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

URL

Name

URL

Synopsis

Class Name:

```
java.net.URL
```

Superclass:

```
java.lang.Object
```

Immediate Subclasses:

None

Interfaces Implemented:

```
java.io.Serializable
```

Availability:

JDK 1.0 or later

Description

The `URL` class represents a Uniform Resource Locator, or URL. The class provides methods for retrieving the various parts of a URL and also access to the resource itself.

An absolute URL consists of a protocol, a hostname, a port number, a filename, and an optional reference, or anchor. For example, consider the following URL:

```
http://www.woolf.net:81/books/Orlando/chapter4.html#p6
```

This URL consists of the following parts:

Part	Value
Protocol	`http`
Hostname	`www.woolf.net`
Port number	`81`
Filename	`/books/Orlando/chapter4.html`
Reference	`p6`

A relative URL specifies only enough information to locate the resource relative to another URL. The filename component is the only part that must be specified for a relative URL. If the protocol, hostname, or port number is not specified, the value is taken from a fully specified URL. For example, the following is a relative URL based on the absolute URL above:

```
chapter6.html
```

This relative URL is equivalent to the following absolute URL:

```
http://www.woolf.net:81/books/Orlando/chapter6.html
```

The `URL` class also provides access to the resource itself, through the `getContent()`, `openConnection()`, and `openStream()` methods. However, these are all convenience functions: other classes do the actual work of accessing the resource.

A protocol handler is an object that knows how to deal with a specific protocol. For example, an `http` protocol handler opens a connection to an `http` host. In `java.net`, subclasses of `URLStreamHandler` deal with different protocols. A `URLStreamHandlerFactory` selects a subclass of `URLStreamHandler` based on a MIME type. Once the `URLStreamHandler` has established a connection with a host using a specific protocol, a subclass of `ContentHandler` retrieves resource data from the host and creates an object from it.

Class Summary

```
public final class java.net.URL extends java.lang.Object
                                implements java.io.Serializable {
  // Constructors
  public URL(String spec);
  public URL(URL context, String spec);
  public URL(String protocol, String host, String file);
  public URL(String protocol, String host, int port, String file);
  // Class Methods
  public static synchronized void setURLStreamHandlerFactory(
                                  URLStreamHandlerFactory fac);
  // Instance Methods
  public boolean equals(Object obj);
  public final Object getContent();
  public String getFile();
  public String getHost();
  public int getPort();
  public String getProtocol();
  public String getRef();
  public int hashCode();
  public URLConnection openConnection();
  public final InputStream openStream();
  public boolean sameFile(URL other);
  public String toExternalForm();
  public String toString();
  // Protected Instance Methods
  protected void set(String protocol, String host, int port,
                     String file, String ref);
}
```

Constructors

URL

public URL(String spec) throws MalformedURLException

Parameters

`spec`

A `String` that represents a URL.

Throws

`MalformedURLException`

If the string is incorrectly constructed or specifies an unknown protocol.

Description

This constructor creates a `URL` by parsing the given string. The string should specify an absolute URL. Calling this constructor is equivalent to calling `URL(null,_3C/tt_3E spec).`

public URL(URL context, String spec) throws MalformedURLException

`Parameters`

`context`

`A base URL that provides the context for parsing spec.`

`spec`

`A String that represents a URL.`

Throws

MalformedURLException

If the string is incorrectly constructed or specifies an unknown protocol.

Description

This constructor creates a URL relative to the base URL specified by context. If context is not null, and spec specifies a partial URL, the missing parts of spec are inherited from context.

The given string is first parsed to see if it specifies a protocol. If the string contains a colon (:) before the first occurrence of a slash (/), the characters before the colon comprise the protocol.

If spec does not specify a protocol, and context is not null, the protocol is inherited from context, as are the hostname, port number, and filename. If context is null in this situation, the constructor throws a MalformedURLException.

If spec does specify a protocol, and context is null or specifies a different protocol, the context argument is ignored and spec should specify an absolute URL. If context specifies the same protocol as spec, the hostname, port number, and filename from context are inherited.

Once the constructor has created a fully specified URL object, it searches for an appropriate protocol handler of type URLStreamHandler, as described for URL(String,_3C/tt_3E String, int, String). Then the parseURL(default.htm) method of the URLStreamHandleris called to parse the remainder of the URL so that the fields in spec can override any values inherited from context.

public URL(String protocol, String host, String file) throws MalformedURLException

Parameters

protocol

A protocol.

host

A hostname.

file

A filename.

Throws

MalformedURLException

If an unknown protocol is specified.

Description

This constructor creates a URL with the given protocol, hostname, and filename. The port number is set to the default port for the given protocol. Calling this constructor is equivalent to calling URL(protocol,_3C/tt_3E host, -1, file).

public URL(String protocol, String host, int port, String file) throws MalformedURLException

Parameters

protocol

A protocol.

host

A hostname.

port

A port number or -1 to use the default port for the protocol.

file

A filename.

Throws

MalformedURLException

If an unknown protocol is specified.

Description

This constructor creates a URL with the given protocol, hostname, port number, and filename.

If this is the first URL object being created with the specified protocol, a protocol handler of type URLStreamHandler is created for the protocol. Here are the steps that are taken to create a protocol handler:

1. If an application has set up a URLStreamHandlerFactory by calling setURLStreamHandlerFactory(), the constructor calls the createURLStreamHandler() method of that object to create the protocol

handler. The protocol is passed as a String argument to that method.

2. If no URLStreamHandlerFactory has been established, or the createURLStreamHandler() method returns null, the constructor retrieves the value of the system property java.protocol.handler.pkgs. If this value is not null, it is interpreted as a list of packages separated by vertical bar (|) characters. The constructor then tries to load the class named *package*.*protocol*.Handler, where *package* is the name of the first package in the list and *protocol* is the name of the protocol. If the class exists, and is a subclass of URLStreamHandler, it is used as the URLStreamHandler for the protocol. If the class does not exist, or if it exists but is not a subclass of URLStreamHandler, the constructor tries the next package in the list.

3. If the previous step fails to find an appropriate protocol handler, the constructor tries to load the class named sun.net.www.protocol.*protocol*.Handler, where *protocol* is the name of the protocol. If the class exists and is a subclass of URLStreamHandler, it is used as the URLStreamHandler for the protocol. If the class does not exist, or if it exists but is not a subclass of URLStreamHandler, a MalformedURLException is thrown.

Class Methods

setURLStreamHandlerFactory

public static synchronized void setURLStreamHandlerFactory(URLStreamHandlerFactory fac)

Parameters

fac

An object that implements URLStreamHandlerFactory.

Throws

Error

If the factory has already been defined.

SecurityException

If the application does not have permission to set the factory.

Description

This method tells the URL class to use the given URLStreamHandlerFactory object for handling all URL objects. The purpose of this mechanism is to allow a program that hosts applets, such as a web browser, control over the creation of URLStreamHandler objects.

Instance Methods

equals

public boolean equals(Object obj)

Parameters

obj

The object to be compared with this object.

Returns

true

if the objects are equivalent;

false

if they are not.

Overrides

Object.equals()

Description

This method returns true if obj is an instance of URL with the same protocol, hostname, port number, and filename as this URL. The reference is only compared if it is not null in this URL.

getContent

public Object getContent() throws IOException

Returns

The Object created from the resource represented by this URL.

Throws

IOException

If any kind of I/O error occurs.

Description

This method returns the content of the URL, encapsulated in an object that is appropriate for the type of the content. The method is shorthand for calling openConnection().getContent(), which uses a ContentHandler object to retrieve the content.

getFile

public String getFile()

Returns

The filename of the URL.

Description

This method returns the name of the file of this URL. Note that the file can be misleading; although the resource represented by this URL may be a file, it can also be generated on the fly by the server.

getHost

public String getHost()

Returns

The hostname of the URL.

Description

This method returns the hostname from this URL.

getPort

public int getPort()

Returns

The port number of the URL.

Description

This method returns the port number of this URL. If a port number is not specified for this URL, meaning it uses the default port for the protocol, -1 is returned.

getProtocol

public String getProtocol()

Returns

The protocol of the URL.

Description

This method returns the protocol of this URL. Some examples of protocols are: http, ftp, and mailto.

getRef

public String getRef()

Returns

The reference of the URL.

Description

This method returns the reference, or anchor, of this URL.

hashCode

public int hashCode()

Returns

The hashcode of the URL.

Overrides

Object.hashCode()

Description

This method returns a hashcode for this object.

openConnection

public URLConnection openConnection() throws IOException

Returns

A URLConnection object for the URL.

Throws

IOException

If any kind of I/O error occurs.

Description

This method returns a URLConnection than manages a connection to the resource represented by this URL. If there is not already an open connection, the method opens a connection by calling the openConnection() method of the URLStreamHandler for this URL. A URLStreamHandler for the protocol of the URL is created by the constructor of the URL.

openStream

public final InputStream openStream() throws IOException

Returns

A InputStream that reads from this URL.

Throws

IOException

If any kind of I/O error occurs.

Description

This method returns an InputStream object that reads the content of the given URL. The method is shorthand for calling openConnection().getInputStream().

sameFile

public boolean sameFile(URL other)

Parameters

other

The URL to compare.

Returns

A boolean value that indicates if this URL is equivalent to other with the exception of references.

Description

This method returns true if this object and the given URL object specify the same protocol, specify hosts that have the same IP address, specify the same port number, and specify the same filename. The filename comparison is case-sensitive. References specified by the URLs are not considered by this method. This method is a helper method for equals().

toExternalForm

public String toExternalForm)

Returns

A string representation of the URL.

Description

This method returns a string representation of this URL. The string representation is determined by the protocol of the URL. The method calls the toExternalForm() method of the URLStreamHandler for this URL. A URLStreamHandler for the protocol of the URL is created by the constructor of the URL.

toString

public String toString()

Returns

A string representation of the URL.

Overrides

Object.toString()

Description

This method returns a string representation of this URL by calling toExternalForm().

Protected Instance Methods

set

protected void set(String protocol, String host, int port, String file, String ref)

Parameters

protocol

A protocol.

host

A hostname.

port

A port number.

file

A filename.

ref

A reference.

Description

This method sets the protocol, hostname, port number, filename, and reference of this URL. The method is called by a URLStreamHandler to set the parts of the URL. A URLStreamHandler for the protocol of the URL is created by the constructor of the URL. It is this URLStreamHandler that parses the URL string. This method is used after parsing to set the values of the URL.

Inherited Methods

Method	Inherited From	Method	Inherited From
clone()	Object	finalize()	Object
getClass()	Object	notify()	Object
notifyAll()	Object	wait()	Object
wait(long)	Object	wait(long, int)	Object

See Also

ContentHandler, Error, InputStream, IOException, MalformedURLException, SecurityException, URLConnection, URLStreamHandler, URLStreamHandlerFactory

URLConnection

Name

URLConnection

Synopsis

Class Name:

`java.net.URLConnection`

Superclass:

`java.lang.Object`

Immediate Subclasses:

`java.net.HttpURLConnection`

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The `URLConnection` class is an `abstract` class that represents a connection to a `URL`. A subclass of `URLConnection` supports a protocol-specific connection. A `URLConnection` can both read from and write to a `URL`.

A `URLConnection` object is created when the `openConnection()` method is called for a `URL` object. At this point, the actual connection has not yet been made, so setup parameters and general request properties can be modified for the specific connection. The various `set` methods control the setup parameters and request properties. Then the actual connection is made by calling the `connect()` method. Finally, the remote object becomes available, and the header fields and the content are accessed using various `get` methods.

The `URLConnection` class defines quite a few methods for setting parameters and retrieving information. Fortunately, for simple connections, all of the setup parameters and request properties can be left alone, as they all have reasonable default values. In most cases, you'll only be interested in the `getInputStream()` and `getContent()` methods. `getInputStream()` provides an `InputStream` that reads from the `URL`, while `getContent()` uses a `ContentHandler` to return an `Object` that represents the content of the resource. These methods are mirrored by the `openStream()` and `getContent()` convenience methods in the `URL` class.

Class Summary

```
public abstract class java.net.URLConnection extends java.lang.Object {
  // Variables
  protected boolean allowUserInteraction;
  protected boolean connected;
  protected boolean doInput;
  protected boolean doOutput;
  public static FileNameMap fileNameMap;            // New in 1.1
  protected long ifModifiedSince;
  protected URL url;
  protected boolean useCaches;
  // Constructors
  protected URLConnection(URL url);
  // Class Methods
  public static boolean getDefaultAllowUserInteraction();
  public static String getDefaultRequestProperty(String key);
  protected static String guessContentTypeFromName(String fname);
  public static String guessContentTypeFromStream(InputStream is);
  public static synchronized void setContentHandlerFactory(
                                  ContentHandlerFactory fac);
  public static void setDefaultAllowUserInteraction(
                     boolean defaultallowuserinteraction);
  public static void setDefaultRequestProperty(String key,
                     String value);
  // Instance Methods
  public abstract void connect();
  public boolean getAllowUserInteraction();
  public Object getContent()
  public String getContentEncoding();
  public int getContentLength();
  public String getContentType();
  public long getDate();
  public boolean getDefaultUseCaches();
  public boolean getDoInput();
  public boolean getDoOutput();
  public long getExpiration();
  public String getHeaderField(int n);
  public String getHeaderField(String name);
  public long getHeaderFieldDate(String name, long default);
  public int getHeaderFieldInt(String name, int default);
  public String getHeaderFieldKey(int n);
  public long getIfModifiedSince();
  public InputStream getInputStream();
  public long getLastModified();
  public OutputStream getOutputStream();
  public String getRequestProperty(String key);
  public URL getURL(default.htm);
  public boolean getUseCaches();
  public void setAllowUserInteraction(boolean allowuserinteraction);
  public void setDefaultUseCaches(boolean defaultusecaches);
  public void setDoInput(boolean doinput);
  public void setDoOutput(boolean dooutput);
  public void setIfModifiedSince(long ifmodifiedsince);
  public void setRequestProperty(String key, String value);
  public void setUseCaches(boolean usecaches);
  public String toString();
}
```

Variables

allowUserInteraction

protected boolean allowUserInteraction

Description

> A flag that indicates whether or not user interaction is enabled for this connection. If this variable is `true`, it is possible to allow user interactions such as popping up dialog boxes. If it is `false`, no user interaction is allowed. The variable can be set by `setAllowUserInteraction()` and retrieved by `getAllowUserInteraction()`. The default value is `false`, unless the `setDefaultAllowUserInteraction()` method has been called, in which case that method call controls the default value.

connected

protected boolean connected

Description

> A flag that indicates whether or not this object has established a connection to a remote host.

doInput

protected boolean doInput

Description

> A flag that indicates whether or not this connection is enabled for input. Setting this variable to `true` indicates that the connection is going to read data. The variable can be set by `setDoInput()` and retrieved by `getDoInput()`. The default value is `true`.

doOutput

protected boolean doOutput

Description

> A flag that indicates whether or not this connection is enabled for output. Setting this variable to `true` indicates that the connection is going to write data. The variable can be set by `setDoOutput()` and retrieved by `getDoOutput()`. The default value is `false`.

fileNameMap

public static FileNameMap fileNameMap

Availability

> New as of JDK 1.1

Description

> A reference to the object that maps filename extensions to MIME type strings. This variable is used in `guessContentTypeFromName()`.

ifModifiedSince

protected long ifModifiedSince

Description

> A time value, specified as the number of seconds since January 1, 1970, that controls whether or not a resource is fetched based on its last modification time. Some protocols do not bother to retrieve a resource if there is a current local cached copy. However, if the resource has been modified more recently than `ifModifiedSince`, it is retrieved. If `ifModifiedSince` is 0, the resource is always retrieved. The variable can be set by `setIfModifiedSince()` and retrieved by `getIfModifiedSince()`. The default value is 0, which means that the resource must always be retrieved.

url

protected URL url

Description

> The resource to which this `URLConnection` connects. This variable is set to the value of the `URL` argument in the `URLConnection` constructor. It can be retrieved by `getURL(default.htm)`.

useCaches

protected boolean useCaches

Description

A flag that indicates whether or not locally cached resources are used. Some protocols cache documents. If this variable is `true`, the protocol is allowed to use caching whenever possible. If it is `false`, the protocol must always try to retrieve the resource. The variable can be set by `setUseCaches()` and retrieved by `getUseCaches()`. The default value is `true`, unless the `setDefaultUseCaches()` method has been called, in which case that method call controls the default value.

Constructors

URLConnection

protected URLConnection(URL url)

Parameters

`url`

The `URL` to access.

Description

This constructor creates a `URLConnection` object to manage a connection to the destination specified by the given `URL`. The actual connection is not created, however.

Class Methods

getDefaultAllowUserInteraction

public static boolean getDefaultAllowUserInteraction()

Returns

`true` if user interaction is allowed by default; `false` otherwise.

Description

This method returns the default value of the `allowUserInteraction` variable for all instances of `URLConnection`. The default value is `false`, unless the `setDefaultAllowUserInteraction()` method has been called, in which case that method call controls the default value.

getDefaultRequestProperty

public static String getDefaultRequestProperty(String key)

Parameters

`key`

The name of a request property.

Returns

The default value of the named request property.

Description

This method returns the default value for the request property named by the `key` parameter.

guessContentTypeFromName

protected static String guessContentTypeFromName(String fname)

Parameters

`fname`

A `String` that contains the name of a file.

Returns

A guess at the MIME type of the given file, or `null` if a guess cannot be made.

Description

This method uses the `FileNameMap` specified by the variable `fileNameMap` to return a MIME content type for the given filename.

guessContentTypeFromStream

```
 protected static String guessContentTypeFromStream(InputStream is) throws
IOException
```

Parameters

`is`

The input stream to inspect

Returns

A guess at the MIME type of the given input stream, or `null` if a guess cannot be made.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method looks at the first few bytes of an `InputStream` and returns a guess of the MIME content type. Note that the `InputStream` must support marks, which usually means that there is a `BufferedInputStream` at some level. Here are some strings that are recognized and the inferred content type:

String	MIME Type Guess
`#def`	`image/x-bitmap`
`<!`	`text/html`
`<body`	`text/html`
`<head>`	`text/html`
`<html>`	`text/html`
`! XPM2`	`image/x-pixmap`
`GIF8`	`image/gif`

setContentHandlerFactory

```
public static synchronized void setContentHandlerFactory(ContentHandlerFactory fac)
```

Parameters

`fac`

An object that implements `ContentHandlerFactory`.

Throws

`Error`

If the factory has already been defined.

`SecurityException`

If the application does not have permission to set the factory.

Description

This method tells the `URLConnection` class to use the given `ContentHandlerFactory` object for all `URLConnection` objects. The purpose of this mechanism is to allow a program that hosts applets, such as a web browser, control over the creation of `ContentHandler` objects.

setDefaultAllowUserInteraction

```
 public static void setDefaultAllowUserInteraction( boolean
defaultallowuserinteraction)
```

Parameters

`defaultallowuserinteraction`

The new default value.

Description

This method sets the default value of the `allowUserInteraction` variable for all new instances of `URLConnection`.

setDefaultRequestProperty

```
 public static void setDefaultRequestProperty(String key, String value)
```

Parameters

`key`

The name of a request property.

`value`

The new default value.

Description

This method sets the default value of the request property named by the `key` parameter.

Instance Methods

connect

public abstract void connect() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Description

When a `URLConnection` object is created, it is not immediately connected to the resource specified by its associated `URL` object. This method actually establishes the connection. If this method is called after the connection has been established, it does nothing.

Before the connection is established, various parameters can be set with the `set` methods. After the connection has been established, it is an error to try to set these parameters.

As they retrieve information about the resource specified by the `URL` object, many of the `get` methods require that the connection be established. If the connection has not been established when one of these methods is called, the connection is established implicitly.

getAllowUserInteraction

public boolean getAllowUserInteraction()

Returns

`true` if user interaction is allowed for this connection; `false` otherwise.

Description

This method returns the value of the `allowUserInteraction` variable for this `URLConnection`.

getContent

```
public Object getContent() throws IOException, UnknownServiceException
```

Returns

An `Object` created from this `URLConnection`.

Throws

`IOException`

If any kind of I/O error occurs.

`UnknownServiceException`

If the protocol cannot support the content type.

Description

This method retrieves the content of the resource specified by the `URL` object associated with this `URLConnection`. If the connection for this object has not been established, the method implicitly establishes it.

The method returns an object that encapsulates the content of the connection. For example, for a connection that has content type `image/jpeg`, the method might return a object that belongs to subclass of `Image,` or for content type text/plain, it might return a String. The instanceof operator should determine the specific type of object that is returned.

The method first determines the content type of the connection by calling getContentType(). If this is the first time the content type has been seen, a content handler of type ContentHandler is created for the content type. Here are the steps that are taken to create a content handler:

1. If an application has set up a ContentHandlerFactory by calling setContentHandlerFactory(), the method calls the createContentHandler() method of that object to create the content handler. The content type is passed as a String argument to that method.

2. If no ContentHandlerFactory has been established, or the createContentHandler() method returns null, the method retrieves the value of the system property java.content.handler.pkgs. If this value is not null, it is interpreted as a list of packages separated by vertical bar (|) characters. The method then tries to load the class named *package*.*contentType*, where *package* is the name of the first package in the list and *contentType* is formed by taking the content-type string and replacing every slash (/) character with a period (.) and every other nonalphanumeric character with an underscore (_). If the class exists and is a subclass of ContentHandler, it is used as the ContentHandler for the content type. If the class does not exist, or if it exists but is not a subclass of ContentHandler, the method tries the next package in the list.

3. If the previous step fails to find an appropriate content handler, the method tries to load the class named sun.net.www.content.*contentType*, where *contentType* is formed by taking the content-type string and replacing every slash (/) character with a period (.) and every other nonalphanumeric character with an underscore (_). If the class exists and is a subclass of ContentHandler, it is used as the ContentHandler for the content type. If the class does not exist, or if it exists but is not a subclass of ContentHandler, a UnknownServiceException is thrown.

getContentEncoding

public String getContentEncoding()

Returns

The content encoding, or null if it is not known.

Description

This method retrieves the content encoding of the resource specified by the URL object associated with this URLConnection. In other words, the method returns the value of the content-encoding header field. If the connection for this object has not been established, the method implicitly establishes it.

getContentLength

public int getContentLength()

Returns

The content length or -1 if it is not known.

Description

> This method retrieves the content length of the resource specified by the `URL` object associated with this `URLConnection`. In other words, the method returns the value of the `content-length` header field. If the connection for this object has not been established, the method implicitly establishes it.

getContentType

public String getContentType()

Returns

> The content type, or `null` if it is not known.

Description

> This method retrieves the content type of the resource specified by the `URL` object associated with this `URLConnection`. In other words, the method returns the value of the `content-type` header field. This string is generally be a MIME type, such as `image/jpeg` or `text/html`. If the connection for this object has not been established, the method implicitly establishes it.

getDate

public long getDate()

Returns

> The content date, or 0 if it is not known.

Description

> This method retrieves the date of the resource specified by the `URL` object associated with this `URLConnection`. In other words, the method returns the value of the `date` header field. The date is returned as the number of seconds since January 1, 1970. If the connection for this object has not been established, the method implicitly establishes it.

getDefaultUseCaches

public boolean getDefaultUseCaches()

Returns

> `true` if the use of caches is allowed by default; `false` otherwise.

Description

> This method returns the default value of the `useCaches` variable for all instances of `URLConnection`. The default value is `true`, unless the `setDefaultUseCaches()` method has been called, in which case that method call controls the default value.

getDoInput

public boolean getDoInput()

Returns

> `true` if this `URLConnection` is to be used for input; `false` otherwise.

Description

> This method returns the value of the `doInput` variable for this `URLConnection`.

getDoOutput

public boolean getDoOutput()

Returns

> `true` if this `URLConnection` is to be used for output; `false` otherwise.

Description

> This method returns the value of the `doOutput` variable for this `URLConnection`.

getExpiration

public long getExpiration()

Returns

> The content expiration date, or if it is not known.

Description

> This method retrieves the expiration date of the resource specified by the `URL` object associated with this `URLConnection`. In other words, the method returns the value of the `expires` header field. The date is returned as the number of seconds since January 1, 1970. If the connection for this object has not been established, the method implicitly establishes it.

getHeaderField

public String getHeaderField(int n)

Parameters

> `n`

A header field index.

Returns

The value of the header field with the given index, or `null` if `n` is greater than the number of fields in the header.

Description

This method retrieves the value of the header field at index `n` of the resource specified by the `URL` object associated with this `URLConnection`. If the connection for this object has not been established, the method implicitly establishes it.

public String getHeaderField(String name)

Parameters

`name`

A header field name.

Returns

The value of the named header field, or `null` if the header field is not known or its value cannot be determined.

Description

This method retrieves the value of the named header field of the resource specified by the `URL` object associated with this `URLConnection`. This method is a helper for methods like `getContentEncoding()` and `getContentType()`. If the connection for this object has not been established, the method implicitly establishes it.

getHeaderFieldDate

public long getHeaderFieldDate(String name, long default)

Parameters

`name`

A header field name.

`default`

A default date value.

Returns

The value of the named header field parsed as a date value, or `default` if the field is missing or cannot be parsed.

Description

This method retrieves the value of the named header field of the resource specified by the `URL` object associated with this `URLConnection` and parses it as a date value. The date is returned as the number of seconds since January 1, 1970. If the value of the header field cannot be determined or it is not a properly formed date, the given default value is returned. If the connection for this object has not been established, the method implicitly establishes it.

getHeaderFieldInt

public int getHeaderFieldInt(String name, int default)

Parameters

`name`

A header field name.

`default`

A default value.

Returns

The value of the named header field parsed as a number, or `default` if the field is missing or cannot be parsed.

Description

This method retrieves the value of the named header field of the resource specified by the `URL` object associated with this `URLConnection` and parses it as a number. If the value of the header field cannot be determined or it is not a properly formed integer, the given default value is returned. If the connection for this object has not been established, the method implicitly establishes it.

getHeaderFieldKey

public String getHeaderFieldKey(int n)

Parameters

`n`

A header field index.

Returns

The name of the header field at the given index, or `null` if `n` is greater than the number of fields in the header.

Description

This method retrieves the name of the header field at index `n` of the resource specified by the `URL` object associated with this `URLConnection`. If the connection for this object has not been established, the method implicitly establishes it.

getIfModifiedSince

public long getIfModifiedSince()

Returns

The value of the `ifModifiedSince` variable.

Description

This method returns the value of the `ifModifiedSince` variable for this `URLConnection`.

getInputStream

`public InputStream getInputStream() throws IOException, UnknownServiceException`

Returns

An `InputStream` that reads data from this connection.

Throws

`IOException`

If any kind of I/O error occurs.

`UnknownServiceException`

If this protocol does not support input.

Description

This method returns an `InputStream` that reads the contents of the resource specified by the `URL` object associated with this `URLConnection`.

getLastModified

public long getLastModified()

Returns

The content last-modified date, or if it is not known.

Description

This method retrieves the last-modified date of the resource specified by the `URL` object associated with this

`URLConnection`. In other words, the method returns the value of the `last-modified` header field. The date is returned as the number of seconds since January 1, 1970. If the connection for this object has not been established, the method implicitly establishes it.

getOutputStream

`public OutputStream getOutputStream() throws IOException, UnknownServiceException`

Returns

An `OutputStream` that writes data to this connection.

Throws

`IOException`

If any kind of I/O error occurs.

`UnknownServiceException`

If this protocol does not support output.

Description

This method returns an `OutputStream` that writes to the resource specified by the `URL` object associated with this `URLConnection`.

getRequestProperty

public String getRequestProperty(String key)

Parameters

`key`

The name of a request property.

Returns

The value of the named request property.

Description

This method returns the value of the request property named by the `key` parameter.

getURL

public URL getURL(default.htm)

Returns

The `URL` that this connection accesses.

Description

This method returns a reference to the `URL` associated with this object. This is the value of the `url` variable for this `URLConnection`.

getUseCaches

public boolean getUseCaches()

Returns

`true` if this `URLConnection` uses caches; `false` otherwise.

Description

This method returns the value of the `useCaches` variable for this `URLConnection`.

setAllowUserInteraction

public void setAllowUserInteraction(boolean allowuserinteraction)

Parameters

`allowuserinteraction`

A `boolean` value that indicates whether user interaction is allowed or not.

Throws

`IllegalAccessError`

If this method is called after the connection has been established.

Description

This method sets the value of the `allowUserInteraction` variable for this `URLConnection`. This method must be called before this object establishes a connection.

setDefaultUseCaches

public void setDefaultUseCaches(boolean defaultusecaches)

Parameters

`defaultusecaches`

The new default value.

Description

This method sets the default value of the `useCaches` variable for all new instances of `URLConnection`.

setDoInput

public void setDoInput(boolean doinput)

Parameters

`doinput`

A `boolean` value that indicates if this connection is to be used for input.

Throws

`IllegalAccessError`

If this method is called after the connection has been established.

Description

This method sets the value of the `doInput` variable for this `URLConnection`. This method must be called before this object establishes a connection.

setDoOutput

public void setDoOutput(boolean dooutput)

Parameters

`dooutput`

A `boolean` value that indicates if this connection is to be used for output.

Throws

`IllegalAccessError`

If this method is called after the connection has been established.

Description

This method sets the value of the `doOutput` variable for this `URLConnection`. This method must be called before this object establishes a connection.

setIfModifiedSince

public void setIfModifiedSince(long ifmodifiedsince)

Parameters

`ifmodifiedsince`

A time value, specified as the number of seconds since January 1, 1970.

Throws

`IllegalAccessError`

If this method is called after the connection has been established.

Description

This method sets the value of the `ifModifiedSince` variable for this `URLConnection`. This method must be called before this object establishes a connection.

setRequestProperty

public void setRequestProperty(String key, String value)

Parameters

`key`

The name of a request property.

`value`

The new value.

Throws

`IllegalAccessError`

If this method is called after the connection has been established.

Description

This method sets the value of the request property named by the `key` parameter.

setUseCaches

public void setUseCaches(boolean defaultusecaches)

Parameters

`defaultusecaches`

A `boolean` value that indicates if this connection uses caches.

Throws

`IllegalAccessError`

If this method is called after the connection has been established.

Description

This method sets the value of the `useCaches` variable for this `URLConnection`. This method must be called before this object establishes a connection.

toString

public String toString()

Returns

A string representation of the `URLConnection`.

Overrides

`Object.toString()`

Description

This method returns a string representation of this `URLConnection`.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

ContentHandler, ContentHandlerFactory, Error, FileNameMap, HttpURLConnection, IllegalAccessError, InputStream, IOException, OutputStream, SecurityException, UnknownServiceException, URL, URLStreamHandler, URLStreamHandlerFactory

URLEncoder

Name

URLEncoder

Synopsis

Class Name:

```
java.net.URLEncoder
```

Superclass:

```
java.lang.Object
```

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The `URLEncoder` class defines a single `static` method that converts a `String` to its URL-encoded form. More precisely, the `String` is converted to a MIME type called `x-www-form-urlencoded`.

This is the format used when posting forms on the Web. The algorithm leaves letters, numbers, and the dash (`-`), underscore (`_`), period (`.`), and asterisk (`*`) characters unchanged. Space characters are converted to plus signs (`+`). All other characters are encoded with a percent sign (`%`) followed by the character code represented as a two-digit hexadecimal number. For example, consider the following string:

```
Jean-Louis Gassée
```

This string gets encoded as:

```
Jean-Louis+Gas%8ee
```

The point of the `URLEncoder` class is to provide a way to canonicalize a string into an extremely portable subset of ASCII that can be handled properly by computers around the world.

Class Summary

```
public class java.net.URLEncoder extends java.lang.Object {
  // Class Methods
  public static String encode(String s);
}
```

Class Methods

encode

public static String encode(String s)

Parameters

`s`

The string to encode.

Returns

A URL-encoded string.

Description

This method returns the contents of the `String` in the `x-www-form-urlencoded` format.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long timeout, int nanos)`	`Object`		

See Also

`String`, `URL`

PREVIOUS HOME NEXT
URLConnection BOOK INDEX URLStreamHandler

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

URLStreamHandler

Name

URLStreamHandler

Synopsis

Class Name:

`java.net.URLStreamHandler`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The `URLStreamHandler` class is an `abstract` class that defines methods that encapsulate protocol-specific behavior. A stream handler protocol knows how to establish a connection for a particular protocol and how to parse the protocol-specific portion of a URL. An application does not normally create a `URLStreamHandler` directly; the appropriate subclass of `URLStreamHandler` is created by a `URLStreamHandlerFactory`.

The main purpose of a subclass of `URLStreamHandler` is to create a `URLConnection` object for a given `URL`. The `URLStreamHandler` object creates an object of the appropriate subclass of `URLConnection` for the protocol type specified by the URL. In order for a `URL` object to handle a protocol type such as `http`, `ftp`, or `nntp`, it needs an object of the appropriate subclass of `URLStreamHandler` to handle the protocol-specific details.

Class Summary

```
public abstract class java.net.URLStreamHandler extends java.lang.Object {
  // Protected Instance Methods
  protected abstract URLConnection openConnection(URL u)
  protected void parseURL(URL u, String spec, int start, int limit);
  protected void setURL(URL u, String protocol, String host,
                        int port, String file, String ref);
  protected String toExternalForm(URL u);
}
```

Protected Instance Methods

openConnection

`protected abstract URLConnection openConnection(URL u) throws IOException`

Parameters

`u`

The `URL` being connected to.

Returns

The `URLConnection` object for the given `URL`.

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method handles the protocol-specific details of establishing a connection to a remote resource specified by the `URL`. The connection should be handled just up to the point where the resource data can be downloaded. A `ContentHandler` then takes care of downloading the data and creating an appropriate object.

A subclass of `URLStreamHandler` must implement this method.

parseURL

protected void parseURL(URL u, String spec, int start, int limit)

Parameters

`u`

A reference to a `URL` object that receives the results of parsing.

`spec`

The string representation of a URL to be parsed.

`start`

The offset at which to begin parsing the protocol-specific portion of the URL.

`limit`

The offset of the last character that is to be parsed.

Description

This method parses the string representation of a URL into a `URL` object.

Some parts of the `URL` object may already be specified if `spec` specifies a relative URL. However, values for those parts in `spec` can override the inherited context.

The method only parses the protocol-specific portion of the URL. In other words, `start` should specify the character immediately after the first colon (:), which marks the termination of the protocol type, and `limit` should either be the last character in the string or the first pound sign (#), which marks the beginning of a protocol-independent anchor. Rather than return a result, the method calls the `set()` method of the specified `URL` object to set its fields to the appropriate values.

The implementation of the `parseURL(default.htm)` method in `URLStreamHandler` parses the string representation as if it were an `http` specification. A subclass that implements a protocol stream handler for a different protocol must override this method to properly parse the URL.

setURL

protected void setURL(URL u, String protocol, String host, int port, String file, String ref)

Parameters

`u`

A reference to a URL object to be modified.

protocol

A protocol.

host

A hostname.

port

A port number.

file

A filename.

ref

A reference.

Description

This method sets the protocol, hostname, port number, filename, and reference of the given URL to the specified values by calling the set() method of the URL. Only subclasses of URLStreamHandler are allowed to call the set() method of a URL object.

toExternalForm

protected String toExternalForm(URL u)

Parameters

u

The URL object to convert to a string representation.

Returns

A string representation of the given URL.

Description

This method unparses a URL object and returns a string representation of the URL.

The implementation of the toExternalForm() method in URLStreamHandler returns a string representation that is appropriate for an `http` specification. A subclass that implements a protocol stream handler for a different protocol must override this method to create a correct string representation.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`ContentHandler`, `IOException`, `URL`, `URLConnection`, `URLStreamHandlerFactory`

PREVIOUS | HOME | NEXT
URLEncoder | BOOK INDEX | URLStreamHandlerFactory

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

URLStreamHandlerFactory

Name

URLStreamHandlerFactory

Synopsis

Interface Name:

`java.net.StreamHandlerFactory`

Super-interface:

None

Immediate Sub-interfaces:

None

Implemented By:

None

Availability:

JDK 1.0 or later

Description

The `URLStreamHandlerFactory` interface defines a method that creates a `URLStreamHandler` object for a specific protocol. The interface is implemented by classes that select `URLStreamHandler` subclasses to process particular protocol types.

The `URL` class uses a `URLStreamHandlerFactory` to create `URLStreamHandler` objects. The protocol type is determined by the portion of the URL up to the first colon. For example, given the following URL:

```
http://www.tolstoi.org/ilych.html
```

the protocol type is `http`. A `URLStreamHandlerFactory` that recognizes `http` returns a `URLStreamHandler` that can process the URL.

Interface Declaration

```
public abstract interface java.net.URLStreamHandlerFactory {
  // Methods
  public abstract URLStreamHandler createURLStreamHandler(String protocol);
}
```

Methods

createURLStreamHandler

public abstract URLStreamHandler createURLStreamHandler (String protocol)

Parameters

> `protocol`
>
> > A `String` that represents a protocol.

Description

> This method creates an object of the appropriate subclass of `URLStreamHandler` that can process a URL using the named protocol.

See Also

`URL`, `URLStreamHandler`

UnknownHostException

Name

UnknownHostException

Synopsis

Class Name:

    `java.net.UnknownHostException`

Superclass:

    `java.io.IOException`

Immediate Subclasses:

    None

Interfaces Implemented:

    None

Availability:

    JDK 1.0 or later

Description

A `UnknownHostException` is thrown when a hostname cannot be resolved to an IP address.

Class Summary

```
public class java.net.UnknownHostException extends java.io.IOException {
  // Constructors
  public UnknownHostException();
  public UnknownHostException(String host);
}
```

Constructors

UnknownHostException

public UnknownHostException()

Description

> This constructor `create`s an `UnknownHostException` with no associated detail message.

public UnknownHostException(String host)

Parameters

> `host`
>
> > The detail message.

Description

> This constructor creates an `UnknownHostException` with the specified detail message, which should be the hostname that cannot be resolved.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`

`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Throwable`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Exception`, `IOException`, `RuntimeException`

PREVIOUS HOME NEXT
URLStreamHandlerFactory BOOK INDEX UnknownServiceException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

UnknownServiceException

Name

UnknownServiceException

Synopsis

Class Name:

    java.net.UnknownServiceException

Superclass:

    java.io.IOException

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

A `UnknownServiceException` is thrown when an unrecognized service is requested, which can occur when the MIME type returned by a `URLConnection` does not match any recognized types.

Class Summary

```
public class java.net.UnknownServiceException
             extends java.io.IOException {
  // Constructors
  public UnknownServiceException();
  public UnknownServiceException(String msg);
}
```

Constructors

UnknownServiceException

public UnknownServiceException()

Description

This constructor `create`s an `UnknownServiceException` with no associated detail message.

public UnknownServiceException(String msg)

Parameters

`msg`

The detail message.

Description

This constructor creates an `UnknownServiceException` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`

`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Throwable`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Exception`, `IOException`, `RuntimeException`

PREVIOUS | HOME | NEXT
UnknownHostException | BOOK INDEX | BreakIterator

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

CharacterIterator

Name

CharacterIterator

Synopsis

Interface Name:

```
java.text.CharacterIterator
```

Super-interface:

```
java.lang.Cloneable
```

Immediate Sub-interfaces:

None

Implemented By:

```
java.text.StringCharacterIterator
```

Availability:

New as of JDK 1.1

Description

The `CharacterIterator` interface defines methods that support bidirectional movement through a sequence of text. The interface is implemented by classes that maintain a current position in a sequence of characters. The interface provides methods for moving to the first, last, next, and previous characters in the sequence. The `BreakIterator` classes uses this interface to locate boundaries in textual sequences.

Class Summary

```
public abstract interface java.text.CharacterIterator
                          extends java.lang.Cloneable {
  // Constants
  public static final char DONE;
  // Methods
  public abstract Object clone();
  public abstract char current();
  public abstract char first();
  public abstract int getBeginIndex();
  public abstract int getEndIndex();
  public abstract int getIndex();
  public abstract char last();
  public abstract char next();
  public abstract char previous();
  public abstract char setIndex(int position);
}
```

Constants

DONE

public final static char DONE

Description

> A constant that indicates that the beginning or end of the text has been reached. It can be returned by `next()` or `previous()`.

Methods

clone

public abstract Object clone()

Returns

> A copy of this `CharacterIterator`.

Overrides

> `Object.clone()`

Description

> This method creates a copy of this `CharacterIterator` and returns it.

current

public abstract char current()

Returns

> The character at the current position of this `CharacterIterator` or `DONE` if the current position is not within the text sequence.

Description

> This method returns the character at the current position of this `CharacterIterator`. The current position is returned by `getIndex()`.

first

public abstract char first()

Returns

> The first character in this `CharacterIterator`.

Description

This method returns the character at the first position in this `CharacterIterator`. The first position is returned by `getBeginIndex()`. The current position of the iterator is set to this position.

getBeginIndex

public abstract int getBeginIndex()

Returns

The index of the first character in this `CharacterIterator`.

Description

This method returns the index of the beginning of the text for this `CharacterIterator`.

getEndIndex

public abstract int getEndIndex()

Returns

The index after the last character in this `CharacterIterator`.

Description

This method returns the index of the character following the end of the text for this `CharacterIterator`.

getIndex

public abstract int getIndex()

Returns

The index of the current character in this `CharacterIterator`.

Description

This method returns the current position, or index, of this `CharacterIterator`.

last

public abstract char last()

Returns

The last character in this `CharacterIterator`.

Description

This method returns the character at the ending position of this `CharacterIterator`. The last position is the value of `getEndIndex()-1`. The current position of the iterator is set to this position.

next

public abstract char next()

Returns

The next character in this `CharacterIterator` or `DONE` if the current position is already at the end of the text.

Description

This method increments the current position of this `CharacterIterator` by one and returns the character at the new position. If the current position is already at `getEndIndex()`, the position is not changed, and `DONE` is returned.

previous

public abstract char previous()

Returns

The previous character in this `CharacterIterator` or `DONE` if the current position is already at the beginning of the text.

Description

This method decrements the current position of this `CharacterIterator` by one and returns the character at the new position. If the current position is already at `getBeginIndex()`, the position is not changed, and `DONE` is returned.

setIndex

public abstract char setIndex(int position)

Parameters

`position`

The new position.

Returns

The character at the specified position in this `CharacterIterator`.

Throws

`IllegalArgumentException`

If the given position is not between `getBeginIndex()` and `getEndIndex()-1`.

Description

This method sets the current position, or index, of this `CharacterIterator` to the given position.

See Also

`BreakIterator`, `StringCharacterIterator`

ChoiceFormat

Name

ChoiceFormat

Synopsis

Class Name:

`java.text.ChoiceFormat`

Superclass:

`java.text.NumberFormat`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `ChoiceFormat` class is a concrete subclass of `NumberFormat` that maps numerical ranges to strings, or formats. `ChoiceFormat` objects are used most often by `MessageFormat` objects to handle plurals, verb agreement, and other such issues.

The ranges in a `ChoiceFormat` are specified as an ascending array of `double` values, where each number is the bottom end of a range. A value is mapped to a format when it falls within the range for that format. If the value does not fall in any of the ranges, it is mapped to the first or the last format, depending on whether the value is too low or too high. For example, consider the following code:

```
double[] limits = {1, 10, 100};
String[] labels = {"small", "medium", "large"}
ChoiceFormat cf = new ChoiceFormat(limits, labels);
```

Any number greater than or equal to one and less than 10 is mapped to the format `"small"`. Any number greater than or equal to 10 and less than 100 is mapped to `"medium"`. Numbers greater than or equal to 100 are mapped to `"large"`. Furthermore, numbers less than one are also mapped to `"small"`.

The `nextDouble()` and `previousDouble()` methods can generate `double` values that are higher or lower than other `double` values. These methods provide another way to specify the limits used by a `ChoiceFormat` object.

As shown above, you can create a `ChoiceFormat` object by specifying the limits and formats in two separate arrays. You can also create a `ChoiceFormat` object using a pattern string that specifies the limits and formats. The string is of the form:

```
[limit1]#[format1]|[limit2]#[format2]|...
```

A `<` character can be used in place of the # to indicate that the next higher number, as determined by `nextDouble()`, should be used as the limit. The `toPattern()` method can be used to generate the pattern string for an existing `ChoiceFormat` object.

Note that you create `ChoiceFormat` objects directly, rather than through factory methods. This is because `ChoiceFormat` does not implement any locale-specific behavior. To produce properly internationalized output, the formats for a `ChoiceFormat` should come from a `ResourceBundle` instead of being embedded in the code.

Class Summary

```
public class java.text.ChoiceFormat extends java.text.NumberFormat {
  // Constructors
  public ChoiceFormat(String newPattern);
  public ChoiceFormat(double[] limits, String[] formats);
  // Class Methods
  public static final double nextDouble(double d);
  public static double nextDouble(double d, boolean positive);
  public static final double previousDouble(double d);
  // Instance Methods
  public void applyPattern(String newPattern);
  public Object clone();
  public boolean equals(Object obj);
  public StringBuffer format(double number,
                     StringBuffer toAppendTo, FieldPosition status);
  public StringBuffer format(long number,
                     StringBuffer toAppendTo, FieldPosition status);
  public Object[] getFormats();
  public double[] getLimits();
  public int hashCode();
  public Number parse(String text, ParsePosition status);
  public void setChoices(double[] limits, String[] formats);
  public String toPattern();
}
```

Constructors

ChoiceFormat

public ChoiceFormat(String newPattern)

Parameters

`newPattern`

The pattern string.

Description

This constructor creates a `ChoiceFormat` that uses the limits and formats represented by the given pattern string.

public ChoiceFormat(double[] limits, String[] formats)

Parameters

`limits`

An array of limits. Each element is the lower end of a range that runs up through, but not including, the next element.

`formats`

An array of format strings that correspond to the limit ranges.

Description

This constructor creates a `ChoiceFormat` that uses the given limits and format strings

Class Methods

nextDouble

public static final double nextDouble(double d)

Parameters

`d`

A `double` value.

Returns

The least `double` that is greater than `d`.

Description

This method returns the least `double` greater than `d`. Calling this method is equivalent to `nextDouble(d, true)`.

public static double nextDouble(double d, boolean positive)

Parameters

`d`

A `double` value.

`positive`

A `boolean` value that specifies whether to return the next higher or next lower value.

Returns

If `positive` is `true`, the least `double` that is greater than `d`. If `positive` is `false`, the greatest `double` that is less than `d`.

Description

This method finds the next higher or next lower `double` value from `d`, depending on the value of `positive`. If `positive` is `true`, the method returns the least `double` greater than `d`. Otherwise, the method returns the greatest `double` less than `d`.

previousDouble

public static final double previousDouble(double d)

Parameters

`d`

A `double` value.

Returns

The greatest `double` that is less than `d`.

Description

This method returns the greatest `double` less than `d`. Calling this method is equivalent to `nextDouble(d, false)`.

Instance Methods

applyPattern

public void applyPattern(String newPattern)

Parameters

`newPattern`

The pattern string.

Description

This method tells this `ChoiceFormat` to use the limits and formats represented by the given formatting pattern string. Pattern strings for `ChoiceFormat` objects are described above in the class description.

clone

public Object clone()

Returns

A copy of this `ChoiceFormat`.

Overrides

`NumberFormat.clone()`

Description

This method creates a copy of this `ChoiceFormat` and returns it.

equals

public boolean equals(Object obj)

Parameters

`obj`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Format.equals()`

Description

This method returns `true` if `obj` is an instance of `ChoiceFormat` and is equivalent to this `ChoiceFormat`.

format

`public StringBuffer format(double number, StringBuffer toAppendTo, FieldPosition status)`

Parameters

`number`

The `double` value to be formatted.

`toAppendTo`

A `StringBuffer` on which to append the formatted information.

`status`

Ignored.

Returns

The given `StringBuffer` with the `String` corresponding to the given number appended to it.

Overrides

`NumberFormat.format(double, StringBuffer, FieldPosition)`

Description

This method formats the given number and appends the result to the given `StringBuffer`.

`public StringBuffer format(long number, StringBuffer toAppendTo, FieldPosition status)`

Parameters

`number`

The `long` value to be formatted.

`toAppendTo`

A `StringBuffer` on which to append the formatted information.

`status`

Ignored.

Returns

The given `StringBuffer` with the `String` corresponding to the given number appended to it.

Overrides

`NumberFormat.format(long, StringBuffer, FieldPosition)`

Description

This method formats the given number and appends the result to the given `StringBuffer`.

getFormats

public Object[] getFormats()

Returns

An array that contains the format strings.

Description

This method returns an array containing the current set of format strings.

getLimits

public double[] getLimits()

Returns

An array that contains the limit values.

Description

This method returns an array that contains the current set of limits.

hashCode

public int hashCode()

Returns

A hashcode for this object.

Overrides

`NumberFormat.hashCode()`

Description

This method returns a hashcode for this `ChoiceFormat`.

parse

public Number parse(String text, ParsePosition status)

Parameters

`text`

The string to be parsed.

`status`

A `ParsePosition` object that can specify a position in the string.

Returns

A `Number` object that encapsulates the value that corresponds to the longest format string that matches the text that starts at the given position. If there is no matching format string, the value `Double.NaN` is returned.

Overrides

`NumberFormat.parse(String, ParsePosition)`

Description

This method parses a number from the given string, starting from the given position. The method returns a `Number` object that encapsulates the value that corresponds to the longest format string that matches the text starting at the given position. If there is no matching format string, the method returns the value `Double.NaN`.

If there is a matching format string, the index value of the given `ParsePosition` object is incremented by the length of that format string.

setChoices

public void setChoices(double[] limits, String[] formats)

Parameters

`limits`

An array of limits. Each element is the lower end of a range that runs up through, but not including, the next element.

`formats`

An array of format strings that correspond to the limit ranges.

Description

This method sets the limits and format strings that this `ChoiceFormat` uses.

toPattern

public String toPattern()

Returns

The pattern string of this `ChoiceFormat`.

Description

This method returns a string that represents the limits and format strings of this `ChoiceFormat`. Pattern strings for `ChoiceFormat` objects are described above in the class description.

Inherited Methods

Method	Inherited From	Method	Inherited From
`finalize()`	`Object`	`format(double)`	`NumberFormat`
`format(long number)`	`NumberFormat`	`format(Object, StringBuffer, FieldPosition)`	`NumberFormat`
`getClass()`	`Object`	`getMaximumFractionDigits()`	`NumberFormat`
`getMaximumIntegerDigits()`	`NumberFormat`	`getMinimumFractionDigits()`	`NumberFormat`
`getMinimumIntegerDigits()`	`NumberFormat`	`isGroupingUsed()`	`NumberFormat`
`isParseIntegerOnly()`	`NumberFormat`	`notify()`	`Object`
`notifyAll()`	`Object`	`parse(String)`	`NumberFormat`

`parseObject(String)`	`Format`	`parseObject(String, ParsePosition)`	`NumberFormat`
`setGroupingUsed(boolean)`	`NumberFormat`	`setMaximumFractionDigits(int)`	`NumberFormat`
`setMaximumIntegerDigits(int)`	`NumberFormat`	`setMinimumFractionDigits(int)`	`NumberFormat`
`setMinimumIntegerDigits(int)`	`NumberFormat`	`setParseIntegerOnly(boolean)`	`NumberFormat`
`toString()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`FieldPosition`, `MessageFormat`, `Number`, `NumberFormat`, `ParsePosition`, `ResourceBundle`, `String`, `StringBuffer`

PREVIOUS	HOME	NEXT
CharacterIterator	BOOK INDEX	CollationElementIterator

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

JAVA
Fundamental Classes Reference

CollationElementIterator

Name

CollationElementIterator

Synopsis

Class Name:

`java.text.CollationElementIterator`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

A `RuleBasedCollator` object creates an instance of the `CollationElementIterator` class to iterate through the characters of a string and determine their relative collation sequence. A `CollationElementIterator` object performs callbacks to the `RuleBasedCollator` that created it to get the information it needs to recognize groups of characters that are treated as single collation characters. For example, a `RuleBasedCollator` for a Spanish language locale would be set up to treat `ch' as a single letter. A `CollationElementIterator` object also gets information from its `RuleBasedCollator` that is used to determine the collation ordering priority for characters.

A collation-ordering priority of a character is a composite integer value that determines how the character is collated. This priority is comprised of:

- A primary order that distinguishes between different letters. Characters that are considered to be different letters, such as `e' and `f', have different primary orders. Different forms of the same letter, such as `e' and `E', or an accented form of `e', have the same primary order. Primary orders are short values.

- A secondary order that distinguishes between accented forms of the same letter. An unaccented `e' has a different secondary order than forms of `e' that have different accents. `E' and `e' have the same secondary order, as do upper- and lowercase forms of `e' that have the same accent. Secondary orders are byte values.

- A tertiary order that distinguishes between case differences. `E' and `e' have different tertiary orders. Tertiary orders are byte values.

The `next()` method returns the collation-ordering priority of the next logical character. Primary, secondary, and tertiary orders are extracted from an ordering priority with the `primaryOrder()`, `secondaryOrder()`, and `tertiaryOrder()` methods.

Class Summary

```
public final class java.text.CollationElementIterator
                   extends java.lang.Object {
   // Constants
   public static final int NULLORDER;

  // Class Methods
   public static final int primaryOrder(int order);
   public static final short secondaryOrder(int order);
   public static final short tertiaryOrder(int order);
  // Instance Methods
   public int next();
   public void reset();
}
```

Constants

NULLORDER

public final static int NULLORDER

Description

A constant that is returned by `next()` if the end of the string has been reached.

Class Methods

primaryOrder

public static final int primaryOrder(int order)

Returns

The primary order component of the given order key.

Description

This method extracts the primary order value from the given order key.

secondaryOrder

public static final short secondaryOrder(int order)

Returns

The secondary order component of the given order key.

Description

This method extracts the secondary order value from the given order key.

tertiaryOrder

public static final short tertiaryOrder(int order)

Returns

The tertiary order component of the given order key.

Description

This method extracts the tertiary order value from the given order key.

Instance Methods

next

public int next()

Returns

The order value of the next character in the string.

Description

This method returns the order key for the next character in the string. The returned value can be broken apart using the `primaryOrder()`, `secondaryOrder()`, and `tertiaryOrder()` methods.

reset

public void reset()

Description

This method resets the position of this `CollationElementIterator` to the beginning of the string.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Collator`, `RuleBasedCollator`, `String`

PREVIOUS
ChoiceFormat

HOME
BOOK INDEX

NEXT
CollationKey

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

CollationKey

Name

CollationKey

Synopsis

Class Name:

`java.text.CollationKey`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `CollationKey` class optimizes the sorting of many strings. The easiest way to compare strings is using `Collator.compare()`, but this can be inefficient, especially if the same strings are compared many times. Instead, you can create a `CollationKey` for each of your strings and compare the `CollationKey` objects to each other using the `compareTo()` method. A `CollationKey` is essentially a bit representation of a `String` object. Two `CollationKey` objects can be compared bitwise, which allows for a fast comparison.

You cannot create `CollationKey` objects directly; you must create them through a specific `Collator` object using `Collator.getCollationKey()`. You can only compare `CollationKey` objects that have been generated from the same `Collator`.

Class Summary

```
public final class java.text.CollationKey extends java.lang.Object {
  // Instance Methods
  public int compareTo(CollationKey target);
  public boolean equals(Object target);
  public String getSourceString();
  public int hashCode();
  public byte[] toByteArray();
}
```

Instance Methods

compareTo

public int compareTo(CollationKey target)

Parameters

`target`

The key to compare with this `CollationKey`.

Returns

`-1` if this `CollationKey` is less than `target`, `0` if this `CollationKey` is equal to `target`, or `1` if this `CollationKey` is greater than `target`.

Description

This method returns an integer that indicates the ordering of this `CollationKey` and the given `CollationKey`. Only `CollationKey` objects generated by the same `Collator` should be compared.

equals

public boolean equals(Object target)

Parameters

`target`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `obj` is an instance of `CollationKey` and is equivalent to this `CollationKey`.

getSourceString

public String getSourceString()

Returns

The string that generated this `CollationKey`.

Description

This method returns the string that was passed to `Collator.getCollationKey()` to create this `CollationKey`.

hashCode

public int hashCode()

Returns

A hashcode for this object.

Overrides

`Object.hashCode()`

Description

This method returns a hashcode for this `CollationKey`.

toByteArray

public byte[] toByteArray()

Returns

A byte array that represents this `CollationKey`.

Description

This method returns a byte array that represents the value of this `CollationKey`, with the most significant byte first.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`

`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Collator`, `RuleBasedCollator`, `String`

PREVIOUS
CollationElementIterator

HOME
BOOK INDEX

NEXT
Collator

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Collator

Name

Collator

Synopsis

Class Name:

`java.text.Collator`

Superclass:

`java.lang.Object`

Immediate Subclasses:

`java.text.RuleBasedCollator`

Interfaces Implemented:

`java.lang.Cloneable`, `java.io.Serializable`

Availability:

New as of JDK 1.1

Description

The Collator class compares strings in a manner that is appropriate for a particular locale. Although Collator is an abstract class, the getInstance() factory methods can be used to get a usable instance of a Collator subclass that implements a particular collation strategy. One subclass, RuleBasedCollator, is provided as part of the JDK.

A Collator object has a *strength* property that controls the level of difference that is considered significant for comparison purposes. The Collator class provides four strength values: PRIMARY, SECONDARY, TERTIARY, and IDENTICAL. Although the interpretation of these strengths is locale-dependent, they generally have the following meanings:

PRIMARY

> The comparison considers letter differences, but ignores case and diacriticals.

SECONDARY

> The comparison considers letter differences and diacriticals, but ignores case.

TERTIARY

> The comparison considers letter differences, case, and diacriticals.

IDENTICAL

> The comparison considers all differences.

The default comparison strength is TERTIARY.

If you only need to compare two String objects once, the compare() method of the Collator class provides the best performance. However, if you need to compare the same String objects multiple times, such as when you are sorting, you should use CollationKey objects instead. A CollationKey object contains a String that has been converted into a series of bits that can be compared in a bitwise fashion against other CollationKey objects. You use a Collator object to create a CollationKey for a given String.

Class Summary

```
public abstract class java.text.Collator extends java.lang.Object
                      implements java.io.Serializable,
                                 java.lang.Cloneable {
  // Constants
  public static final int CANONICAL_DECOMPOSITION;
  public static final int FULL_DECOMPOSITION;
  public static final int IDENTICAL;
  public static final int NO_DECOMPOSITION;
  public static final int PRIMARY;
  public static final int SECONDARY;
  public static final int TERTIARY;
  // Constructors
  protected Collator();
  // Class Methods
  public static synchronized Locale[] getAvailableLocales();
  public static synchronized Collator getInstance();
  public static synchronized Collator getInstance(Locale desiredLocale);
  // Instance Methods
  public Object clone();
  public abstract int compare(String source, String target);
  public boolean equals(Object that);
  public boolean equals(String source, String target);
  public abstract CollationKey getCollationKey(String source);
  public synchronized int getDecomposition();
  public synchronized int getStrength();
  public abstract synchronized int hashCode();
  public synchronized void setDecomposition(int decompositionMode);
  public synchronized void setStrength(int newStrength);
}
```

Constants

CANONICAL_DECOMPOSITION

public final static int CANONICAL_DECOMPOSITION

Description

> A decomposition constant that specifies that Unicode 2.0 characters which are canonical variants are decomposed for collation. This is the default decomposition setting.

FULL_DECOMPOSITION

public final static int FULL_DECOMPOSITION

Description

A decomposition constant that specifies that Unicode 2.0 canonical variants and compatibility variants are decomposed for collation. This is the most complete decomposition setting, and thus the slowest setting.

IDENTICAL

public final static int IDENTICAL

Description

A strength constant that specifies that all differences are considered significant for comparison purposes.

NO_DECOMPOSITION

public final static int NO_DECOMPOSITION

Description

A decomposition setting that specifies that no Unicode characters are decomposed for collation. This is the least complete decomposition setting, and thus the fastest setting. It only works correctly for languages that do not use diacriticals.

PRIMARY

public final static int PRIMARY

Description

A strength constant that specifies that only primary differences are considered significant for comparison purposes. Primary differences are typically letter differences.

SECONDARY

public final static int SECONDARY

Description

A strength constant that specifies that only secondary differences and above are considered significant for comparison purposes. Secondary differences are typically differences in diacriticals, or accents.

TERTIARY

public final static int TERTIARY

Description

> A strength constant that specifies that only tertiary differences and above are considered significant for comparison purposes. Tertiary differences are typically differences in case. This is the default strength setting.

Constructors

Collator

protected Collator()

Description

> This constructor creates a `Collator` with the default strength of `TERTIARY` and default decomposition mode of `CANONICAL_DECOMPOSITION`.

Class Methods

getAvailableLocales

public static synchronized Locale[] getAvailableLocales()

Returns

> An array of `Locale` objects.

Description

> This method returns an array of the `Locale` objects for which this class can create `Collator` objects.

getInstance

public static synchronized Collator getInstance()

Returns

A `Collator` appropriate for the default `Locale`.

Description

This method creates a `Collator` that compares strings in the default `Locale`.

`public static synchronized Collator getInstance( Locale desiredLocale)`

Parameters

`desiredLocale`

The `Locale` to use.

Returns

A `Collator` appropriate for the given `Locale`.

Description

This method creates a `Collator` that compares strings in the given `Locale`.

Instance Methods

clone

public Object clone()

Returns

A copy of this `Collator`.

Overrides

`Object.clone()`

Description

This method creates a copy of this `Collator` and returns it.

compare

public abstract int compare(String source, String target)

Parameters

`source`

The source string.

`target`

The target string.

Returns

`-1` if `source` is less than `target`, `0` if the strings are equal, or `1` if `source` is greater than `target`.

Description

This method compares the given strings according to the collation rules for this `Collator` and returns a value that indicates their relationship. If either of the strings are compared more than once, a `CollationKey` should be used instead.

equals

public boolean equals(Object that)

Parameters

`that`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `obj` is an instance of `Collator` and is equivalent to this `Collator`.

public boolean equals(String source, Source target)

Parameters

`source`

The source string.

`target`

The target string.

Returns

`true` if the given strings are equal; `false` otherwise.

Description

This method compares the given strings for equality using the collation rules for this `Collator`. Note that this method applies locale-specific rules and is thus not the same as `String.equals()`.

getCollationKey

public abstract CollationKey getCollationKey(String source)

Parameters

`source`

The string to use when generating the `CollationKey`.

Returns

A `CollationKey` for the given string.

Description

This method generates a `CollationKey` for the given string. The returned object can be compared with other `CollationKey` objects using `CollationKey.compareTo()`. This comparison is faster than using `Collator.compare()`, so if the same string is used for many comparisons, you should use `CollationKey` objects.

getDecomposition

public synchronized int getDecomposition()

Returns

The decomposition mode for this `Collator`.

Description

This method returns the current decomposition mode for this `Collator`. The decomposition mode specifies how composed Unicode characters are handled during collation. You can adjust the decomposition mode to choose between faster and more complete collation. The returned value is one of the following values: `NO_DECOMPOSITION`, `CANONICAL_DECOMPOSITION`, or `FULL_DECOMPOSITION`.

getStrength

public synchronized int getStrength()

Returns

The strength setting for this `Collator`.

Description

This method returns the current strength setting for this `Collator`. The strength specifies the minimum level of difference that is considered significant during collation. The returned value is one of the following values: `PRIMARY`, `SECONDARY`, `TERTIARY`, or `IDENTICAL`.

hashCode

public abstract synchronized int hashCode()

Returns

A hashcode for this object.

Overrides

`Object.hashCode()`

Description

This method returns a hashcode for this `Collator`.

setDecomposition

public synchronized void setDecomposition(int decompositionMode)

Parameters

`decompositionMode`

The decomposition mode: `NO_DECOMPOSITION`, `CANONICAL_DECOMPOSITION`, or `FULL_DECOMPOSITION`.

Description

This method sets the decomposition mode for this `Collator`. The decomposition mode specifies how composed Unicode characters are handled during collation. You can adjust the decomposition mode to choose between faster and more complete collation.

setStrength

public synchronized void setStrength(int newStrength)

Parameters

`newStrength`

The new strength setting: `PRIMARY`, `SECONDARY`, `TERTIARY`, or `IDENTICAL`.

Description

This method sets the strength of this `Collator`. The strength specifies the minimum level of difference that is considered significant during collation.

Inherited Methods

Method	Inherited From	Method	Inherited From
`finalize()`	`Object`	`getClass()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`toString()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`CollationKey`, `Locale`, `RuleBasedCollator`, `String`

PREVIOUS
CollationKey

HOME
BOOK INDEX

NEXT
DateFormat

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

DateFormat

Name

DateFormat

Synopsis

Class Name:

`java.text.DateFormat`

Superclass:

`java.text.Format`

Immediate Subclasses:

`java.text.SimpleDateFormat`

Interfaces Implemented:

`java.lang.Cloneable`

Availability:

New as of JDK 1.1

Description

The `DateFormat` class formats and parses dates and times in a locale-specific manner. `DateFormat` is an `abstract` class, but it provides factory methods that return useful instances of `DateFormat` subclasses. These factory methods come in three groups:

- The `getDateInstance()` methods return objects that format and parse only dates.

- The `getDateTimeInstance()` methods return objects that format and parse date and time combinations.
- The `getTimeInstance()` methods return objects that format only times.

Certain of these factory methods allow you to specify the style, or length, of the resulting date and time strings. The interpretation of the style parameter is locale-specific. For the locale `Locale.US`, the styles and their results are as follows:

`FULL`

```
Tuesday, March 04, 1997 12:00:00 o'clock AM EST
```

`LONG`

```
March 04, 1997 12:00:00 AM EST
```

`MEDIUM`

```
04-Mar-97 12:00:00 AM
```

`SHORT`

```
3/4/97 12:00 AM
```

There is also a `DEFAULT` style, which is equivalent to `MEDIUM`.

The `DateFormat` class defines a number of field constants that represent the various fields in formatted date and time strings. These field constants can create `FieldPosition` objects.

Class Summary

```
public abstract class java.text.DateFormat extends java.text.Format
                      implements java.lang.Cloneable {
  // Constants
  public static final int AM_PM_FIELD;
  public static final int DATE_FIELD;
  public static final int DAY_OF_WEEK_FIELD;
  public static final int DAY_OF_WEEK_IN_MONTH_FIELD;
  public static final int DAY_OF_YEAR_FIELD;
  public static final int DEFAULT;
  public static final int ERA_FIELD;
  public static final int FULL;
  public static final int HOUR0_FIELD;
  public static final int HOUR1_FIELD;
  public static final int HOUR_OF_DAY0_FIELD;
  public static final int HOUR_OF_DAY1_FIELD;
  public static final int LONG;
  public static final int MEDIUM;
  public static final int MILLISECOND_FIELD;
  public static final int MINUTE_FIELD;
  public static final int MONTH_FIELD;
  public static final int SECOND_FIELD;
  public static final int SHORT;
  public static final int TIMEZONE_FIELD;
  public static final int WEEK_OF_MONTH_FIELD;
  public static final int WEEK_OF_YEAR_FIELD;
  public static final int YEAR_FIELD;
  // Variables
  protected Calendar calendar;
  protected NumberFormat numberFormat;
  // Constructors
  protected DateFormat();
  // Class Methods
  public static Locale[] getAvailableLocales();
  public static final DateFormat getDateInstance();
  public static final DateFormat getDateInstance(int style);
  public static final DateFormat getDateInstance(int style, Locale aLocale);
  public static final DateFormat getDateTimeInstance();
  public static final DateFormat getDateTimeInstance(int dateStyle,
                                 int timeStyle);
  public static final DateFormat getDateTimeInstance(int dateStyle,
                                 int timeStyle, Locale aLocale);
  public static final DateFormat getInstance();
  public static final DateFormat getTimeInstance();
  public static final DateFormat getTimeInstance(int style);
  public static final DateFormat getTimeInstance(int style, Locale aLocale);
  // Instance Methods
  public Object clone();
  public boolean equals(Object obj);
  public final String format(Date date);
  public final StringBuffer format(Object obj, StringBuffer toAppendTo,
                          FieldPosition fieldPosition);
  public abstract StringBuffer format(Date date, StringBuffer toAppendTo,
                             FieldPosition fieldPosition);
  public Calendar getCalendar();
  public NumberFormat getNumberFormat();
  public TimeZone getTimeZone();
  public int hashCode();
  public boolean isLenient();
  public Date parse(String text);
  public abstract Date parse(String text, ParsePosition pos);
  public Object parseObject(String source, ParsePosition pos);
  public void setCalendar(Calendar newCalendar);
  public void setLenient(boolean lenient);
  public void setNumberFormat(NumberFormat newNumberFormat);
  public void setTimeZone(TimeZone zone);
}
```

Constants

AM_PM_FIELD

public final static int AM_PM_FIELD

Description

A field constant that represents the A.M./P.M. field.

DATE_FIELD

public final static int DATE_FIELD

Description

A field constant that represents the date (day of month) field.

DAY_OF_WEEK_FIELD

public final static int DAY_OF_WEEK_FIELD

Description

A field constant that represents the day-of-the-week field.

DAY_OF_WEEK_IN_MONTH_FIELD

public final static int DAY_OF_WEEK_IN_MONTH_FIELD

Description

A field constant that represents the day of the week in the current month field.

DAY_OF_YEAR_FIELD

public final static int DAY_OF_YEAR_FIELD

Description

A field constant that represents the day-of-the-year field.

DEFAULT

public final static int DEFAULT

Description

A constant that specifies the default style.

ERA_FIELD

public final static int ERA_FIELD

Description

A field constant that represents the era field.

FULL

public final static int FULL

Description

A constant that specifies the most complete style.

HOUR0_FIELD

public final static int HOUR0_FIELD

Description

A field constant that represents the zero-based hour field.

HOUR1_FIELD

public final static int HOUR1_FIELD

Description

A field constant that represents the one-based hour field.

HOUR_OF_DAY0_FIELD

public final static int HOUR_OF_DAY0_FIELD

Description

A field constant that represents the zero-based hour of the day field.

HOUR_OF_DAY1_FIELD

public final static int HOUR_OF_DAY1_FIELD

Description

A field constant that represents the one-based hour of the day field.

LONG

public final static int LONG

Description

A constant that specifies the long style.

MEDIUM

public final static int MEDIUM

Description

A constant that specifies the medium style.

MILLISECOND_FIELD

public final static int MILLISECOND_FIELD

Description

A field constant that represents the millisecond field.

MINUTE_FIELD

public final static int MINUTE_FIELD

Description

A field constant that represents the minute field.

MONTH_FIELD

public final static int MONTH_FIELD

Description

A field constant that represents the month field.

SECOND_FIELD

public final static int SECOND_FIELD

Description

A field constant that represents the second field.

SHORT

public final static int SHORT

Description

A constant that specifies the short style.

TIMEZONE_FIELD

public final static int TIMEZONE_FIELD

Description

A field constant that represents the time-zone field.

WEEK_OF_MONTH_FIELD

public final static int WEEK_OF_MONTH_FIELD

Description

A field constant that represents the week-of-the-month field.

WEEK_OF_YEAR_FIELD

public final static int WEEK_OF_YEAR_FIELD

Description

A field constant that represents the week-of-the-year field.

YEAR_FIELD

public final static int YEAR_FIELD

Description

A field constant that represents the year field.

Variables

calendar

protected Calendar calendar

Description

> A `Calendar` object that internally generates the field values for formatting dates and times.

numberFormat

protected NumberFormat numberFormat

Description

> A `NumberFormat` object that internally formats the numbers in dates and times.

Constructors

DateFormat

protected DateFormat()

Description

> This constructor creates a `DateFormat`.

Class Methods

getAvailableLocales

public static Locale[] getAvailableLocales()

Returns

> An array of `Locale` objects.

Description

> This method returns an array of the `Locale` objects for which this class can create `DateFormat` objects.

getDateInstance

public static final DateFormat getDateInstance()

Returns

> A `DateFormat` appropriate for the default `Locale` that uses the default style.

Description

This method creates a `DateFormat` that formats and parses dates in the default locale with the default style.

public static final DateFormat getDateInstance(int style)

Parameters

`style`

A style constant.

Returns

A `DateFormat` appropriate for the default `Locale` that uses the given style.

Description

This method creates a `DateFormat` that formats and parses dates in the default locale with the given style.

`public static final DateFormat getDateInstance(int style, Locale aLocale)`

Parameters

`style`

A style constant.

`aLocale`

The `Locale` to use.

Returns

A `DateFormat` appropriate for the given `Locale` that uses the given style.

Description

This method creates a `DateFormat` that formats and parses dates in the given locale with the given style.

getDateTimeInstance

public static final DateFormat getDateTimeInstance()

Returns

A `DateFormat` appropriate for the default `Locale` that uses the default date and time styles.

Description

This method creates a `DateFormat` that formats and parses dates and times in the default locale with the default date and time styles.

`public static final DateFormat getDateTimeInstance(int dateStyle, int timeStyle)`

Parameters

`dateStyle`

A style constant.

`timeStyle`

A style constant.

Returns

A `DateFormat` appropriate for the default `Locale` that uses the given data and time styles.

Description

This method creates a `DateFormat` that formats and parses dates and times in the default locale with the given date and time styles.

`public static final DateFormat getDateTimeInstance(int dateStyle, int timeStyle, Locale aLocale)`

Parameters

`dateStyle`

A style constant.

`timeStyle`

A style constant.

`aLocale`

The `Locale` to use.

Returns

A `DateFormat` appropriate for the given `Locale` that uses the given date and time styles.

Description

This method creates a `DateFormat` that formats and parses dates and times in the given locale with the given date and time styles.

getInstance

public static final DateFormat getInstance()

Returns

A `DateFormat` appropriate for the default `Locale`.

Description

This method creates a general purpose `DateFormat` by calling `getDateTimeInstance(DateFormat.SHORT, DateFormat.SHORT)`.

getTimeInstance

public static final DateFormat getTimeInstance()

Returns

A `DateFormat` appropriate for the default `Locale` that uses the default style.

Description

This method creates a `DateFormat` that formats and parses times in the default locale with the default style.

public static final DateFormat getTimeInstance(int style)

Parameters

`style`

A style constant.

Returns

A `DateFormat` appropriate for the default `Locale` that uses the given style.

Description

This method creates a `DateFormat` that formats and parses times in the default locale with the given style.

`public static final DateFormat getTimeInstance(int style, Locale aLocale)`

Parameters

`style`

A style constant.

`aLocale`

The `Locale` to use.

Returns

A `DateFormat` appropriate for the given `Locale` that uses the given style.

Description

This method creates a `DateFormat` that formats and parses times in the given locale with the given style.

Instance Methods

clone

public Object clone()

Returns

A copy of this `DateFormat`.

Overrides

`Format.clone()`

Description

This method creates a copy of this `DateFormat` and returns it.

equals

public boolean equals(Object obj)

Parameters

`obj`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `obj` is an instance of `DateFormat` and is equivalent to this `DateFormat`.

format

public final String format(Date date)

Parameters

`date`

The `Date` object to be formatted.

Returns

A string that contains a formatted representation of the date.

Description

This method formats the given date and returns the result as a string.

public final StringBuffer format(Object obj, StringBuffer toAppendTo, FieldPosition fieldPosition)

Parameters

`obj`

The object to be formatted.

`toAppendTo`

A `StringBuffer` on which to append the formatted information.

`fieldPosition`

A date or time field.

Returns

The given buffer `toAppendTo` with the formatted representation of the object appended to it.

Overrides

`Format.format(Object, StringBuffer, FieldPosition)`

Description

This method formats the given object and appends the result to the given `StringBuffer`. If `fieldPosition` refers to one of the time or date fields, its beginning and ending indices are filled with the beginning and ending positions of the given field in the resulting formatted string.

`public abstract StringBuffer format(Date date, StringBuffer toAppendTo, FieldPosition fieldPosition)`

Parameters

`date`

The `Date` object to be formatted.

`toAppendTo`

A `StringBuffer` on which to append the formatted information.

`fieldPosition`

A date or time field.

Returns

The given buffer `toAppendTo` with the formatted representation of the date appended to it.

Description

This method formats the given date and appends the result to the given `StringBuffer`. If `fieldPosition` refers to one of the time or date fields, its beginning and ending indices are filled with the beginning and ending positions of the given field in the resulting formatted string.

getCalendar

public Calendar getCalendar()

Returns

The internal `Calendar` object of this `DateFormat`.

Description

This method returns the `Calendar` object that this `DateFormat` uses internally.

getNumberFormat

public NumberFormat getNumberFormat()

Returns

> The internal `NumberFormat` object of this `DateFormat`.

Description

> This method returns the `NumberFormat` object that this `DateFormat` uses internally.

getTimeZone

public TimeZone getTimeZone()

Returns

> The internal `TimeZone` object of this `DateFormat`.

Description

> This method returns the `TimeZone` object that this `DateFormat` uses internally.

hashCode

public int hashCode()

Returns

> A hashcode for this object.

Overrides

> `Object.hashCode()`

Description

> This method returns a hashcode for this `DateFormat`.

isLenient

public boolean isLenient()

Returns

> A `boolean` value that indicates the leniency of this `DateFormat`.

Description

This method returns the current leniency of this `DateFormat`. A value of `false` indicates that the `DateFormat` throws exceptions when it tries to parse questionable data, while a value of `true` indicates that the `DateFormat` makes its best guess to interpret questionable data. For example, if the `DateFormat` is being lenient, a date such as March 135, 1997 is interpreted as the 135th day after March 1, 1997.

parse

public Date parse(String text) throws ParseException

Parameters

`text`

The string to be parsed.

Returns

The `Date` object represented by the given string.

Throws

`ParseException`

If the text cannot be parsed as a date.

Description

This method parses a date from the given string, starting from the beginning of the string.

public abstract Date parse(String text, ParsePosition pos)

Parameters

`text`

The string to be parsed.

`pos`

A `ParsePosition` object that can specify a position in the string.

Returns

The `Date` object represented by the text starting at the given position.

Description

This method parses a date from the given string, starting from the given position. After the string has been parsed, the given `ParsePosition` object is updated so that its index is after the parsed text.

parseObject

public Object parseObject(String source, ParsePosition pos)

Parameters

`source`

The string to be parsed.

`pos`

A `ParsePosition` object that can specify a position in the string.

Returns

The object represented by the text starting at the given position.

Overrides

`Format.parseObject(String, ParsePosition)`

Description

This method parses a date from the given string, starting from the given position. After the string has been parsed, the given `ParsePosition` object is updated so that its index is after the parsed text.

setCalendar

public void setCalendar(Calendar newCalendar)

Parameters

`newCalendar`

The new `Calendar` to use.

Description

This method sets the `Calendar` that this `DateFormat` uses internally.

setLenient

public void setLenient(boolean lenient)

Parameters

`lenient`

A `boolean` value that specifies the leniency of this `DateFormat`.

Description

This method sets the leniency of this `DateFormat`. A value of `false` specifies that the `DateFormat` throws exceptions when it tries to parse questionable data, while a value of `true` indicates that the `DateFormat` makes its best guess to interpret questionable data. For example, if the `Calendar` is being lenient, a date such as March 135, 1997 is interpreted as the 135th day after March 1, 1997.

setNumberFormat

public void setNumberFormat(NumberFormat newNumberFormat)

Parameters

`newNumberFormat`

The new `NumberFormat` to use.

Description

This method sets the `NumberFormat` that this `DateFormat` uses internally.

setTimeZone

public void setTimeZone(TimeZone zone)

Parameters

`zone`

The new `TimeZone` to use.

Description

This method sets the `TimeZone` that this `DateFormat` uses internally.

Inherited Methods

Method	Inherited From	Method	Inherited From
`finalize()`	`Object`	`format(Object)`	`Format`
`getClass()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`parseObject(String)`	`Format`
`toString()`	`Object`	`wait()`	`Object`

`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`Calendar`, `Cloneable`, `Date`, `FieldPosition`, `Format`, `Locale`, `NumberFormat`, `ParsePosition`, `String`, `StringBuffer`, `TimeZone`

DateFormatSymbols

Name

DateFormatSymbols

Synopsis

Class Name:

`java.text.DateFormatSymbols`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

`java.lang.Cloneable`, `java.io.Serializable`

Availability:

New as of JDK 1.1

Description

The `DateFormatSymbols` class encapsulates date and time formatting data that is locale-specific, like the names of the days of the week and the names of the months. Typically, you do not need to instantiate `DateFormatSymbols` yourself. Instead, an instance is automatically created for you, behind the scenes, when you use one of the factory methods in `DateFormat` to create a `DateFormat` object. You can retrieve a `DateFormatSymbols` object by calling the `getDateFormatSymbols()` method of `SimpleDateFormat`. Once you have a `DateFormatSymbols` object, you can modify the strings it uses if you want to change them.

Class Summary

```
public class java.text.DateFormatSymbols extends java.lang.Object
             implements java.io.Serializable, java.lang.Cloneable {
  // Constructors
  public DateFormatSymbols();
  public DateFormatSymbols(Locale locale);
  // Instance Methods
  public Object clone();
  public boolean equals(Object obj);
  public String[] getAmPmStrings();
  public String[] getEras();
  public String getLocalPatternChars();
  public String[] getMonths();
  public String[] getShortMonths();
  public String[] getShortWeekdays();
  public String[] getWeekdays();
  public String[][] getZoneStrings();
  public int hashCode();
  public void setAmPmStrings(String[] newAmpms);
  public void setEras(String[] newEras);
  public void setLocalPatternChars(String newLocalPatternChars);
  public void setMonths(String[] newMonths);
  public void setShortMonths(String[] newShortMonths);
  public void setShortWeekdays(String[] newShortWeekdays);
  public void setWeekdays(String[] newWeekdays);
  public void setZoneStrings(String[][] newZoneStrings);
}
```

Constructors

DateFormatSymbols

public DateFormatSymbols()

Throws

`MissingResourceException`

If the resources for the default locale cannot be found or loaded.

Description

This constructor creates a `DateFormatSymbols` object for the default locale.

public DateFormatSymbols(Locale locale)

Parameters

`locale`

The `Locale` to use.

Throws

`MissingResourceException`

If the resources for the given locale cannot be found or loaded.

Description

This constructor creates a `DateFormatSymbols` object for the given locale.

Instance Methods

clone

public Object clone()

Returns

A copy of this `DateFormatSymbols`.

Overrides

`Object.clone()`

Description

This method creates a copy of this `DateFormatSymbols` and returns it.

equals

public boolean equals(Object obj)

Parameters

`obj`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `obj` is an instance of `DecimalFormatSymbols` and is equivalent to this `DateFormatSymbols`.

getAmPmStrings

public String[] getAmPmStrings()

Returns

An array of strings used for the A.M./P.M. field for this `DateFormatSymbols`.

Description

This method returns the strings that are used for the A.M./P.M. field (e.g., "AM", "PM").

getEras

public String[] getEras()

Returns

An array of strings used for the era field for this `DateFormatSymbols`.

Description

This method returns the strings that are used for the era field (e.g., "BC", "AD").

getLocalPatternChars

public String getLocalPatternChars()

Returns

A string that contains the data-time pattern characters for this `DateFormatSymbols`.

Description

This method returns the data-time pattern characters for the locale of this object.

getMonths

public String[] getMonths()

Returns

An array of strings used for the month field for this `DateFormatSymbols`.

Description

This method returns the strings that are used for the month field (e.g., "January", "February").

getShortMonths

public String[] getShortMonths()

Returns

An array of strings used for the short month field for this `DateFormatSymbols`.

Description

This method returns the strings that are used for the short (i.e., three-character) month field (e.g., "Jan", "Feb").

getShortWeekdays

public String[] getShortWeekdays()

Returns

An array ofstrings used for the short weekday field for this `DateFormatSymbols`.

Description

This method returns the strings that are used for the short (i.e., three-character) weekday field (e.g., "Mon", "Tue").

getWeekdays

public String[] getWeekdays()

Returns

An array ofstrings used for the weekday field for this `DateFormatSymbols`.

Description

This method returns the strings that are used for the weekday field (e.g., "Monday", "Tuesday").

getZoneStrings

public String[][] getZoneStrings()

Returns

> An array of arrays of strings used for the time zones for this `DateFormatSymbols`.

Description

> This method returns the time-zone strings. Each subarray is an array of six strings that specify a time-zone ID, its long name, its short name, its daylight-savings-time name, its short daylight-savings-time name, and a major city in the time zone. For example, an entry for Mountain Standard Time is:

```
{"MST", "Mountain Standard Time", "MST",
 "Mountain Daylight Time", "MDT", "Denver"}
```

hashCode

public int hashCode()

Returns

> A hashcode for this object.

Overrides

> `Object.hashCode()`

Description

> This method returns a hashcode for this `DateFormatSymbols` object.

setAmPmStrings

public void setAmPmStrings(String[] newAmpms)

Parameters

`newAmpms`

The new strings.

Description

This method sets the strings that are used for the A.M./P.M. field for this `DateFormatSymbols`.

setEras

public void setEras(String[] newEras)

Parameters

`newEras`

The new strings.

Description

This method sets the strings that are used for the era field for this `DateFormatSymbols`.

setLocalPatternChars

public void setLocalPatternChars(String newLocalPatternChars)

Parameters

`newLocalPatternChars`

The new date-time pattern characters.

Description

This method sets the date-time pattern characters of this `DateFormatSymbols` object.

setMonths

public void setMonths(String[] newMonths)

Parameters

`newMonths`

The new strings.

Description

This method sets the strings that are used for the month field for this `DateFormatSymbols`.

setShortMonths

public void setShortMonths(String[] newShortMonths)

Parameters

`newShortMonths`

The new strings.

Description

This method sets the strings that are used for the short (i.e., three-character) month field for this `DateFormatSymbols`.

setShortWeekdays

public void setShortWeekdays(String[] newShortWeekdays)

Parameters

`newShortWeekdays`

The new strings.

Description

This method sets the strings that are used for the short (i.e., three-character) weekday field for this `DateFormatSymbols`.

setWeekdays

public void setWeekdays(String[] newWeekdays)

Parameters

`newWeekdays`

The new strings.

Description

This method sets the strings that are used for the weekday field for this `DateFormatSymbols`.

setZones

public void setZones(String[][] newZoneStrings)

Parameters

`newZoneStrings`

The new strings.

Description

This method sets the strings that are used for the time-zone field for this `DateFormatSymbols`.

Inherited Methods

Method	Inherited From	Method	Inherited From
`finalize()`	`Object`	`getClass()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`toString()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`Calendar`, `DateFormat`, `Locale`, `SimpleDateFormat`, `TimeZone`

DecimalFormat

Name

DecimalFormat

Synopsis

Class Name:

> `java.text.DecimalFormat`

Superclass:

> `java.text.NumberFormat`

Immediate Subclasses:

> None

Interfaces Implemented:

> None

Availability:

> New as of JDK 1.1

Description

The `DecimalFormat` class is a concrete subclass of `NumberFormat` that formats and parses numbers using a formatting pattern. Typically, you do not need to instantiate `DecimalFormat` yourself. Instead, the factory methods of `NumberFormat` return instances of `DecimalFormat` that are appropriate for particular locales.

However, if you need a specialized number format, you can instantiate your own `DecimalFormat` using a pattern string. You can also modify the formatting pattern of an existing `DecimalFormat` object using the `applyPattern()` method.

A pattern string has the following form:

```
positive-pattern[;negative-pattern]
```

If the negative pattern is omitted, negative numbers are formatted using the positive pattern with a `-` character prepended to the result. Each pattern has the following form:

```
[prefix]integer[.fraction][suffix]
```

The following symbols are significant in the pattern string.

Symbol	Description
`0`	A digit
`#`	A digit where 0 is not shown
`.`	A placeholder for a decimal separator
`,`	A placeholder for a grouping separator
`;`	The format separator
`-`	The default negative prefix
`%`	Divides value by 100 and shows as a percentage

Any characters other than these special characters can appear in the prefix or the suffix. A single quote can be used to escape a special character, if you need to use one of these symbols in a prefix or a suffix.

For example, the pattern string for U.S. currency values is:

```
$#,##0.00;($#,##0.00)
```

This indicates that a `$` character is prepended to all formatted values. The grouping separator character `,` is inserted every three digits. Exactly two digits after the decimal place are always shown. Negative values are shown in parentheses. Thus, the value `-1234.56` produces output like:

```
($1,234.56)
```

Internally, the `DecimalFormat` class uses a `DecimalFormatSymbols` object to get the numerical strings that are appropriate for a particular locale. If you want to modify these strings, you can get the `DecimalFormatSymbols` object by calling `getDecimalFormatSymbols()`.

Class Summary

```
public class java.text.DecimalFormat extends java.text.NumberFormat {
  // Constructors
  public DecimalFormat();
  public DecimalFormat(String pattern);
  public DecimalFormat(String pattern, DecimalFormatSymbols symbols);
  // Instance Methods
  public void applyLocalizedPattern(String pattern);
  public void applyPattern(String pattern);
  public Object clone();
  public boolean equals(Object obj);
  public StringBuffer format(double number, StringBuffer result,
                      FieldPosition fieldPosition);
  public StringBuffer format(long number, StringBuffer result,
                      FieldPosition fieldPosition);
  public DecimalFormatSymbols getDecimalFormatSymbols();
  public int getGroupingSize();
  public int getMultiplier();
  public String getNegativePrefix();
  public String getNegativeSuffix();
  public String getPositivePrefix();
  public String getPositiveSuffix();
  public int hashCode();
  public boolean isDecimalSeparatorAlwaysShown();
  public Number parse(String text, ParsePosition status);
  public void setDecimalFormatSymbols(DecimalFormatSymbols newSymbols);
  public void setDecimalSeparatorAlwaysShown(boolean newValue);
  public void setGroupingSize(int newValue);
  public void setMultiplier(int newValue);
  public void setNegativePrefix(String newValue);
  public void setNegativeSuffix(String newValue);
  public void setPositivePrefix(String newValue);
  public void setPositiveSuffix(String newValue);
  public String toLocalizedPattern();
  public String toPattern();
}
```

Constructors

DecimalFormat

public DecimalFormat()

Description

> This constructor creates a `DecimalFormat` that uses the default formatting pattern and `DecimalFormatSymbols` that are appropriate for the default locale.

public DecimalFormat(String pattern)

Parameters

> `pattern`
>
> > The pattern string.

Description

This constructor creates a `DecimalFormat` that uses the given formatting pattern and a `DecimalFormatSymbols` that is appropriate for the default locale.

public DecimalFormat(String pattern, DecimalFormatSymbols symbols)

Parameters

`pattern`

The pattern string.

`symbols`

The `DecimalFormatSymbols` to use.

Description

This constructor creates a `DecimalFormat` that uses the given formatting pattern and `DecimalFormatSymbols` object.

Instance Methods

applyLocalizedPattern

public void applyLocalizedPattern(String pattern)

Parameters

`pattern`

The pattern string.

Description

This method tells this `DecimalFormat` to use the given formatting pattern to format and parse numbers. The pattern string is assumed to have been localized to the `DecimalFormatSymbols` object this `DecimalFormat` uses.

applyPattern

public void applyPattern(String pattern)

Parameters

`pattern`

The pattern string.

Description

This method tells this `DecimalFormat` to use the given formatting pattern to format and parse numbers. The pattern string is localized to the `DecimalFormatSymbols` object this `DecimalFormat` uses.

clone

public Object clone()

Returns

A copy of this `DecimalFormat`.

Overrides

`NumberFormat.clone()`

Description

This method creates a copy of this `DecimalFormat` and then returns it.

equals

public boolean equals(Object obj)

Parameters

`obj`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`NumberFormat.equals()`

Description

This method returns `true` if `obj` is an instance of `DecimalFormat` and is equivalent to this `DecimalFormat`.

format

`public StringBuffer format(double number, StringBuffer result, FieldPosition fieldPosition)`

Parameters

`number`

The `double` value to be formatted.

`result`

A `StringBuffer` on which to append the formatted information.

`fieldPosition`

A number field.

Returns

The given buffer `result` with the formatted representation of the number appended to it.

Overrides

`NumberFormat.format(double, StringBuffer, FieldPosition)`

Description

This method formats the given number and appends the result to the given `StringBuffer`. If `fieldPosition` refers to one of the number fields, its beginning and ending indices are filled with the beginning and ending positions of the given field in the resulting formatted string.

`public StringBuffer format(long number, StringBuffer result, FieldPosition fieldPosition)`

Parameters

`number`

The `long` value to be formatted.

`result`

A `StringBuffer` on which to append the formatted information.

`fieldPosition`

A number field.

Returns

The given buffer `result` with the formatted representation of the number appended to it.

Overrides

`NumberFormat.format(double, StringBuffer, FieldPosition)`

Description

This method formats the given number and appends the result to the given `StringBuffer`. If `fieldPosition` refers to one of the number fields, its beginning and ending indices are filled with the beginning and ending positions of the given field in the resulting formatted string.

getDecimalFormatSymbols

public DecimalFormatSymbols getDecimalFormatSymbols()

Returns

The `DecimalFormatSymbols` object used by this `DecimalFormat`.

Description

This method returns the `DecimalFormatSymbols` object that this `DecimalFormat` uses internally.

getGroupingSize

public int getGroupingSize()

Returns

The grouping size of this `DecimalFormat`.

Description

This method returns the grouping size of this `DecimalFormat`. The grouping size is the number of digits between grouping separators in the integer portion of a number. For example, in the number `1,234.56`, the grouping size is `3` (and the grouping symbol is ",").

getMultiplier

public int getMultiplier()

Returns

The multiplier of this `DecimalFormat`.

Description

This method returns the multiplier of this `DecimalFormat`. The multiplier is used to adjust a number before it is formatted or after it is parsed. For example, a percent format has a multiplier of 100 and a suffix of `` `%' ``. Thus, a value of `.42` could be formatted as `42%`.

getNegativePrefix

public String getNegativePrefix()

Returns

The string that is prepended to negative values.

Description

This method returns the prefix string for negative numbers.

getNegativeSuffix

public String getNegativeSuffix()

Returns

The string that is appended to negative values.

Description

This method returns the suffix string for negative numbers.

getPositivePrefix

public String getPositivePrefix()

Returns

The string that is prepended to positive values.

Description

This method returns the prefix string for positive numbers.

getPositiveSuffix

public String getPositiveSuffix()

Returns

The string that is appended to positive values.

Description

This method returns the suffix string for positive numbers.

hashCode

public int hashCode()

Returns

A hashcode for this object.

Overrides

`NumberFormat.hashCode()`

Description

This method returns a hashcode for this `DecimalFormat`.

isDecimalSeparatorAlwaysShown

public boolean isDecimalSeparatorAlwaysShown()

Returns

A `boolean` value that indicates whether or not the decimal separator symbol is always shown.

Description

This method returns `true` if this `DecimalFormat` always shows the decimal separator. The method returns `false` if the decimal separator is only shown if there is a fractional portion of the number being formatted.

parse

public Number parse(String text, ParsePosition status)

Parameters

`text`

The string to be parsed.

`status`

A `ParsePosition` object that specifies a position in the string.

Returns

The `Number` object represented by the text starting at the given position.

Overrides

`NumberFormat.parse(String, ParsePosition)`

Description

This method parses a number from the given string, starting from the given position. After the string has been parsed, the given `ParsePosition` object is updated so that its index is after the parsed text.

setDecimalFormatSymbols

`public void setDecimalFormatSymbols( DecimalFormatSymbols newSymbols)`

Parameters

`newSymbols`

The new `DecimalFormatSymbols` object to use.

Description

This method sets the `DecimalFormatSymbols` object that this `DecimalFormat` uses internally.

setDecimalSeparatorAlwaysShown

public void setDecimalSeparatorAlwaysShown(boolean newValue)

Parameters

`newValue`

The new decimal separator value.

Description

This method specifies whether or not the decimal separator symbol is always shown in formatted numbers. If `newValue` is `false`, the separator is only shown for numbers that have a fractional part. Otherwise, the separator is always shown.

setGroupingSize

public void setGroupingSize(int newValue)

Parameters

`newValue`

The new grouping size.

Description

This method sets the grouping size of this `DecimalFormat`. The grouping size is the number of digits between grouping separators in the integer portion of a number. For example, in the number `1,234.56`, the grouping size is `3` (and the grouping symbol is ",").

setMultiplier

public void setMultiplier(int newValue)

Parameters

`newValue`

The new multiplier.

Description

This method sets the multiplier of this `DecimalFormat`. The multiplier is used to adjust a number before it is formatted or after it is parsed. For example, a percent format has a multiplier of 100 and a suffix of `%`. Thus, a value of `.42` could be formatted as `42%`.

setNegativePrefix

public void setNegativePrefix(String newValue)

Parameters

`newValue`

The new prefix.

Description

This method sets the prefix string for negative values.

setNegativeSuffix

public void setNegativeSuffix(String newValue)

Parameters

`newValue`

The new suffix.

Description

This method sets the suffix string for negative values.

setPositivePrefix

public void setPositivePrefix(String newValue)

Parameters

`newValue`

The new prefix.

Description

This method sets the prefix string for positive values.

setPositiveSuffix

public void setPositiveSuffix(String newValue)

Parameters

`newValue`

The new suffix.

Description

This method sets the suffix string for positive values.

toLocalizedPattern

public String toLocalizedPattern()

Returns

The pattern string of this `DecimalFormat`.

Description

This method returns the pattern string of this `DecimalFormat`, localized with the `DecimalFormatSymbols` object of this `DecimalFormat`.

toPattern

public String toPattern()

Returns

The pattern string of this `DecimalFormat`.

Description

This method returns the pattern string of this `DecimalFormat`.

Inherited Methods

Method	Inherited From	Method	Inherited From
`finalize()`	`Object`	`format(double)`	`NumberFormat`
`format(long)`	`NumberFormat`	`format(Object, StringBuffer, FieldPosition)`	`NumberFormat`
`getClass()`	`Object`	`getMaximumFractionDigits()`	`NumberFormat`
`getMaximumIntegerDigits()`	`NumberFormat`	`getMinimumFractionDigits()`	`NumberFormat`
`getMinimumIntegerDigits()`	`NumberFormat`	`isGroupingUsed()`	`NumberFormat`
`isParseIntegerOnly()`	`NumberFormat`	`notify()`	`Object`
`notifyAll()`	`Object`	`parse(String)`	`NumberFormat`
`parseObject(String)`	`Format`	`parseObject(String, ParsePosition)`	`NumberFormat`
`setGroupingUsed(boolean)`	`NumberFormat`	`setMaximumFractionDigits(int)`	`NumberFormat`
`setMaximumIntegerDigits(int)`	`NumberFormat`	`setMinimumFractionDigits(int)`	`NumberFormat`
`setMinimumIntegerDigits(int)`	`NumberFormat`	`setParseIntegerOnly(boolean)`	`NumberFormat`
`toString()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`DecimalFormatSymbols`, `FieldPosition`, `Number`, `NumberFormat`, `ParsePosition`, `String`, `StringBuffer`

PREVIOUS HOME NEXT
DateFormatSymbols BOOK INDEX DecimalFormatSymbols

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

DecimalFormatSymbols

Name

DecimalFormatSymbols

Synopsis

Class Name:

`java.text.DecimalFormatSymbols`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

`java.lang.Cloneable`, `java.io.Serializable`

Availability:

New as of JDK 1.1

Description

The `DecimalFormatSymbols` class encapsulates number-formatting data that is locale-specific, like grouping separators and decimal separators. Typically, you do not need to instantiate `DecimalFormatSymbols` yourself. Instead, an instance is automatically created for you, behind the scenes, when you use one of the factory methods in `NumberFormat` to create a `DecimalFormat` object. You can retrieve a `DecimalFormatSymbols` object by calling the `getDecimalFormatSymbols()` method of `DecimalFormat`. Once you have a `DecimalFormatSymbols` object, you can modify the strings it uses if you want to change them.

Class Summary

```
public final class java.text.DecimalFormatSymbols extends java.lang.Object
                   implements java.io.Serializable, java.lang.Cloneable {
  // Constructors
  public DecimalFormatSymbols();
  public DecimalFormatSymbols(Locale locale);
  // Instance Methods
  public Object clone();
  public boolean equals(Object obj);
  public char getDecimalSeparator();
  public char getDigit();
  public char getGroupingSeparator();
  public String getInfinity();
  public char getMinusSign();
  public String getNaN();
  public char getPatternSeparator();
  public char getPerMill();
  public char getPercent();
  public char getZeroDigit();
  public int hashCode();
  public void setDecimalSeparator(char decimalSeparator);
  public void setDigit(char digit);
  public void setGroupingSeparator(char groupingSeparator);
  public void setInfinity(String infinity);
  public void setMinusSign(char minusSign);
  public void setNaN(String NaN);
  public void setPatternSeparator(char patternSeparator);
  public void setPerMill(char perMill);
  public void setPercent(char percent);
  public void setZeroDigit(char zeroDigit);
}
```

Constructors

DecimalFormatSymbols

public DecimalFormatSymbols()

Description

This constructor creates a `DecimalFormatSymbols` object for the default locale.

public DecimalFormatSymbols(Locale locale)

Parameters

`locale`

The `Locale` to use.

Description

This constructor creates a `DecimalFormatSymbols` object for the given locale.

Instance Methods

clone

public Object clone()

Returns

A copy of this `DecimalFormatSymbols`.

Overrides

`Object.clone()`

Description

This method creates a copy of this `DecimalFormatSymbols` and returns it.

equals

public boolean equals(Object obj)

Parameters

`obj`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `obj` is an instance of `DateFormatSymbols` and is equivalent to this `DecimalFormatSymbols`.

getDecimalSeparator

public char getDecimalSeparator()

Returns

The character used to separate the integer and fractional parts of a number for this `DecimalFormatSymbols`.

Description

This method returns the decimal separator character (e.g., ".", ",").

getDigit

public char getDigit()

Returns

The character used to represent a digit in a pattern string for this `DecimalFormatSymbols`.

Description

This method returns the digit pattern character, which represents a digit that is not shown if it is zero.

getGroupingSeparator

public char getGroupingSeparator()

Returns

The character used to separate long numbers for this `DecimalFormatSymbols`.

Description

This method returns the grouping separator character (e.g., ",", ".").

getInfinity

public String getInfinity()

Returns

The string used to represent infinity for this `DecimalFormatSymbols`.

Description

This method returns the string that represents infinity.

getMinusSign

public char getMinusSign()

Returns

The character used to signify negative numbers for this `DecimalFormatSymbols`.

Description

This method returns the character that signifies negative numbers.

getNaN

public String getNaN()

Returns

The string used to represent the value not-a-number for this `DecimalFormatSymbols`.

Description

This method returns the string that represents not-a-number.

getPatternSeparator

public char getPatternSeparator()

Returns

The pattern separator character for this `DecimalFormatSymbols`.

Description

This method returns the character used in pattern strings to separate the positive subpattern and negative subpattern.

getPerMill

public char getPerMill()

Returns

The character used to represent the per mille sign for this `DecimalFormatSymbols`.

Description

This method returns the character that represents a per mille value.

getPercent

public char getPercent()

Returns

The character used to represent the percent sign for this `DecimalFormatSymbols`.

Description

This method returns the character that represents a percent value (e.g., %).

getZeroDigit

public char getZeroDigit()

Returns

The character used to represent a digit in a pattern string for this `DecimalFormatSymbols`.

Description

This method returns the zero-digit pattern character, which represents a digit that is shown even if it is zero.

hashCode

public int hashCode()

Returns

A hashcode for this object.

Overrides

`Object.hashCode()`

Description

This method returns a hashcode for this `DecimalFormatSymbols` object.

setDecimalSeparator

public void setDecimalSeparator(char decimalSeparator)

Parameters

`decimalSeparator`

The new decimal separator.

Description

This method sets the decimal separator character for this `DecimalFormatSymbols`.

setDigit

public void setDigit(char digit)

Parameters

`digit`

The new digit pattern character.

Description

This method sets the digit pattern character, which represents a digit that is not shown if it is zero, for this `DecimalFormatSymbols`.

setGroupingSeparator

public void setGroupingSeparator(char groupingSeparator)

Parameters

`groupingSeparator`

The new grouping separator.

Description

This method sets the grouping separator character for this `DecimalFormatSymbols`.

setInfinity

public Void setInfinity(String infinity)

Parameters

`infinity`

The new infinity string.

Description

This method sets the string that represents infinity for this `DecimalFormatSymbols`.

setMinusSign

public void setMinusSign(char minusSign)

Parameters

`minusSign`

The new minus sign.

Description

This method sets the character that signifies negative numbers for this `DecimalFormatSymbols`.

setNaN

public Void setNaN(String NaN)

Parameters

`NaN`

The new non-a-number string.

Description

This method sets the string that represents not-a-number for this `DecimalFormatSymbols`.

setPatternSeparator

public void setPatternSeparator(char patternSeparator)

Parameters

`patternSeparator`

The new pattern separator.

Description

This method sets the character that is used in pattern strings to separate the positive subpattern and negative subpattern for this `DecimalFormatSymbols`.

setPerMill

public void setPerMill(char perMill)

Parameters

`perMill`

The new per mille sign.

Description

This method sets the character that represents the per mille sign for this `DecimalFormatSymbols`.

setPercent

public void setPercent(char percent)

Parameters

`percent`

The new percent sign.

Description

This method sets the character that represents the percent sign for this `DecimalFormatSymbols`.

setZeroDigit

public void setZeroDigit(char zeroDigit)

Parameters

`zeroDigit`

The new zero-digit pattern character.

Description

This method sets the zero-digit pattern character, which represents a digit that is shown even if it is zero, for this `DecimalFormatSymbols`.

Inherited Methods

Method	Inherited From	Method	Inherited From
`finalize()`	`Object`	`getClass()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`toString()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`DecimalFormat`, `NumberFormat`, `Locale`

PREVIOUS
DecimalFormat

HOME
BOOK INDEX

NEXT
FieldPosition

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Format

Name

Format

Synopsis

Class Name:

`java.text.Format`

Superclass:

`java.lang.Object`

Immediate Subclasses:

`java.text.DateFormat`, `java.text.MessageFormat`,

`java.text.NumberFormat`

Interfaces Implemented:

`java.lang.Cloneable`, `java.io.Serializable`

Availability:

New as of JDK 1.1

Description

The `Format` class is an `abstract` class that is the superclass of all classes that handle locale-sensitive parsing and formatting of dates, numbers, and messages. The two `format()` methods take the information in a supplied object and convert it to a string. The two `parseObject()` methods do the reverse; they take the information from a string and construct an appropriate object.

Class Summary

```
public abstract class java.text.Format extends java.lang.Object
                      implements java.io.Serializable, java.lang.Cloneable {
  // Instance Methods
  public Object clone();
  public final String format(Object obj);
  public abstract StringBuffer format(Object obj, StringBuffer toAppendTo,
                                       FieldPosition pos);
  public Object parseObject(String source);
  public abstract Object parseObject(String source, ParsePosition status);
}
```

Instance Methods

clone

public Object clone()

Returns

> A copy of this `Format`.

Overrides

> `Object.clone()`

Description

> This method creates a copy of this `Format` and returns it.

format

public final String format(Object obj)

Parameters

`obj`

The object to be formatted.

Returns

A string that contains a formatted representation of the object.

Throws

`IllegalArgumentException`

If the given object cannot be formatted.

Description

This method formats the given object by calling `format(Object, StringBuffer, FieldPosition)` with an empty `StringBuffer` and a `FieldPosition` that has a value of 0.

`public abstract StringBuffer format(Object obj, StringBuffer toAppendTo, FieldPosition pos)`

Parameters

`obj`

The object to be formatted.

`toAppendTo`

A `StringBuffer` on which to append the formatted information.

`pos`

A field.

Returns

The given buffer `toAppendTo` with the formatted representation of the object appended to it.

Throws

`IllegalArgumentException`

If the given object cannot be formatted.

Description

This method formats the given object and appends the result to the given `StringBuffer`. After the object has been formatted, the beginning and ending indices of the given `FieldPosition` are updated to reflect the field's position in the formatted output.

A subclass of `Format` must implement this method.

parseObject

public Object parseObject(String source) throws ParseException

Parameters

`source`

The string to be parsed.

Returns

The object represented by the given string.

Throws

`ParseException`

If the text cannot be parsed by this `Format`.

Description

This method parses the given text and returns an appropriate object. It does this by calling `parseObject(String, ParsePosition)` with a `ParsePosition` that has an index of 0.

`public abstract Object parseObject(String source, ParsePosition status)`

Parameters

`source`

The string to be parsed.

`status`

A `ParsePosition` object that specifies a position in the string.

Returns

The object represented by the text starting at the given position.

Description

This method parses the given text, starting at the specified position, and returns an object created from the data. After the string has been parsed, the given `ParsePosition` object is updated so that its index is after the parsed text.

A subclass of `Format` must implement this method.

Inherited Methods

Method	Inherited From	Method	Inherited From
`equals(Object)`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`toString()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`DateFormat`, `FieldPosition`, `MessageFormat`, `NumberFormat`, `ParseException`, `ParsePosition`, `String`, `StringBuffer`

MessageFormat

Name

MessageFormat

Synopsis

Class Name:

    java.text.MessageFormat

Superclass:

    java.text.Format

Immediate Subclasses:

    None

Interfaces Implemented:

    None

Availability:

    New as of JDK 1.1

Description

The `MessageFormat` class constructs textual messages using a formatting pattern string. Conceptually, the class functions much like `printf()` in C. Syntactically, however, it is quite different. A `MessageFormat` object uses a pattern string; formatted arguments are placed into the pattern string to produce a resulting string. Arguments are delimited by matching sets of curly braces and may include additional information about how that data should be formatted. For example, consider the following code:

```
String message =
  "Boot of server {0}began at {1, time}on {1, date, full}.";
MessageFormat boot = new MessageFormat(message);
Date now = new Date();
Object[] arguments = {"luna3", now};
System.out.println(boot.format(arguments));
```

This code produces the following output:

```
Boot of server luna3 began at 11:13:22 AM on Monday, March 03, 1997.
```

Each of the arguments is numbered and includes an optional type and an optional style. In the example above, `{1, date, full}` indicates that the argument at index `1` in the argument array should be formatted using a `DateFormat` object with the `FULL` style. The allowed types and styles are:

Type	Object	Styles
`choice`	`ChoiceFormat`	*pattern*
`date`	`DateFormat`	`short`, `medium`, `long`, `full`, *pattern*
`number`	`NumberFormat`	`integer`, `percent`, `currency`, *pattern*
`time`	`DateFormat`	`short`, `medium`, `long`, `full`, *pattern*

For the `date` and `time` types, the styles correspond to the styles, or lengths, of the resulting date and time strings. You can also specify a date or time pattern string as you would for creating a `SimpleDateFormat` object. For the `number` type, the styles correspond to formatting normal numbers, percentage values, and currency values. You can also specify a number pattern string as you would for creating a `DecimalFormat` object. For the `choice` type, you can specify a choice pattern as you would for creating a `ChoiceFormat` object. If no type is specified, the argument should be a string.

The following example shows how to use a `choice` format pattern with a `MessageFormat`:

```
Object[] arguments = {new Integer(1)};
String grammar = "At last count, {0}server{0, choice, 0#s|1#|1<s}
 {0, choice, 0#were|1#was|1<were}booted.";
MessageFormat booted = new MessageFormat(grammar);
System.out.println(booted.format(arguments));
arguments[0] = new Integer(2);
System.out.println(booted.format(arguments));
```

This example produces the following output:

```
At last count, 1 server was booted.
At last count, 2 servers were booted.
```

As an alternative to specifying all of the formatting in the pattern string, you can use an array of `Format` objects to format the arguments. You can specify this array using `setFormats()`.

Note that you create `MessageFormat` objects directly, rather than through factory methods. This is because `MessageFormat` does not implement any locale-specific behavior. To produce properly internationalized output, the pattern string that is used to construct a `MessageFormat` should come from a `ResourceBundle` instead of being embedded in the code.

Class Summary

```
public class java.text.MessageFormat extends java.text.Format {
  // Constructors
  public MessageFormat(String pattern);
  // Class Methods
  public static String format(String pattern, Object[] arguments);
  // Instance Methods
  public void applyPattern(String newPattern);
  public Object clone();
  public boolean equals(Object obj);
  public final StringBuffer format(Object source, StringBuffer result,
                             FieldPosition ignore);
  public final StringBuffer format(Object[] source, StringBuffer result,
                             FieldPosition ignore);
  public Format[] getFormats();
  public Locale getLocale();
  public int hashCode();
  public Object[] parse(String source);
  public Object[] parse(String source, ParsePosition status);
  public Object parseObject(String text, ParsePosition status);
  public void setFormat(int variable, Format newFormat);
  public void setFormats(Format[] newFormats);
  public void setLocale(Locale theLocale);
  public String toPattern();
}
```

Constructors

MessageFormat

public MessageFormat(String pattern)

Parameters

`pattern`

The pattern string.

Description

This constructor creates a `MessageFormat` with the given formatting pattern string.

Class Methods

format

public static String format(String pattern, Object[] arguments)

Parameters

`pattern`

The pattern string.

`arguments`

An array of arguments.

Description

Calling this `static` method is equivalent to constructing a `MessageFormat` using the given formatting pattern string and asking it to format the given arguments with the `format()` method.

Instance Methods

applyPattern

public void applyPattern(String pattern)

Parameters

`pattern`

The pattern string.

Description

This method tells this `MessageFormat` to use the given formatting pattern to format and parse arguments.

clone

public Object clone()

Returns

A copy of this `MessageFormat`.

Overrides

`Format.clone()`

Description

This method creates a copy of this `MessageFormat` and then returns it.

equals

public boolean equals(Object obj)

Parameters

`obj`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Format.equals()`

Description

This method returns `true` if `obj` is an instance of `MessageFormat` and is equivalent to this `MessageFormat`.

format

`public StringBuffer format(Object source, StringBuffer result, FieldPosition ignore)`

Parameters

`source`

The object to be formatted.

`result`

A `StringBuffer` on which to append the formatted information.

`ignore`

Ignored.

Returns

The given buffer `result` with the formatted representation of the object appended to it.

Overrides

`Format.format(Object, StringBuffer, FieldPosition)`

Description

This method formats the given object and appends the result to the given `StringBuffer`. The method assumes that the given object is an array of arguments.

`public StringBuffer format(Object[] source, StringBuffer result, FieldPosition ignore)`

Parameters

`source`

The object array to be formatted.

`result`

A `StringBuffer` on which to append the formatted information.

`ignore`

Ignored.

Returns

The given buffer `result` with the formatted representation of the object array appended to it.

Description

This method formats the given arguments in the object array and appends the result to the given `StringBuffer`.

getFormats

public Format[] getFormats()

Returns

An array of the formats used by this `MessageFormat`.

Description

This method returns the format objects that this `MessageFormat` uses. Note that formats are numbered according to the order in which they appear in the formatting pattern string, not according to their specified argument numbers.

getLocale

public Locale getLocale()

Returns

`The Locale` of this `MessageFormat`.

Description

This method returns the locate for this `MessageFormat`. This locale is used to get default date, time, and number formatters.

hashCode

public int hashCode()

Returns

A hashcode for this object.

Overrides

`Object.hashCode()`

Description

This method returns a hashcode for this `MessageFormat`.

parse

public Object[] parse(String source) throws ParseException

Parameters

`source`

The string to be parsed.

Returns

An array of objects represented by the given string.

Throws

`ParseException`

If the text cannot be parsed.

Description

This method parses arguments from the given string, which should be in the format given by the formatting pattern string of this `MessageFormat`. If the string is not correctly formatted, an exception is thrown.

public Object[] parse(String source, ParsePosition status)

Parameters

`source`

The string to be parsed.

`status`

A `ParsePosition` object that specifies a position in the string.

Returns

An array of objects represented by the test starting at the given position.

Description

This method parses arguments from the given string, starting at the specified position. The string should be in the format given by the formatting pattern string of this `MessageFormat`.

parseObject

public Object parseObject(String text, ParsePosition status)

Parameters

`text`

The string to be parsed.

`status`

A `ParsePosition` object that specifies a position in the string.

Returns

The object represented by the test starting at the given position.

Overrides

`Format.parseObject(String, ParsePosition)`

Description

This method parses arguments from the given string, starting at the specified position. The string should be in the format given by the formatting pattern string of this `MessageFormat`.

setFormat

public void setFormat(int variable, Format newFormat)

Parameters

`variable`

The index of an argument in the pattern string.

`newFormat`

The format object to use.

Description

This method sets the `Format` object that is used for the given argument in the formatting pattern string.

setFormats

public void setFormats(Format[] newFormats)

Parameters

`newFormats`

The format objects to use.

Description

This method sets the `Format` objects that format the arguments of this `MessageFormat`. Note that formats are numbered according to the order in which they appear in the formatting pattern string, not according to their specified argument numbers.

setLocale

public void setLocale(Locale theLocale)

Parameters

`theLocale`

The new locale.

Description

This method sets the `Locale` object that generates the default date, time, and number format objects.

toPattern

public String toPattern()

Returns

> The pattern string of this `MessageFormat`.

Description

> This method returns the pattern string of this `MessageFormat`.

Inherited Methods

Method	Inherited From	Method	Inherited From
`finalize()`	`Object`	`format(Object)`	`Format`
`getClass()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`parseObject(String)`	`Format`
`toString()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`ChoiceFormat`, `DateFormat`, `FieldPosition`, `Format`, `Locale`, `NumberFormat`, `ParseException`, `ParsePosition`, `ResourceBundle`, `String`, `StringBuffer`

PREVIOUS HOME NEXT
Format BOOK INDEX NumberFormat

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

NumberFormat

Name

NumberFormat

Synopsis

Class Name:

`java.text.NumberFormat`

Superclass:

`java.text.Format`

Immediate Subclasses:

`java.text.ChoiceFormat`, `java.text.DecimalFormat`

Interfaces Implemented:

`java.lang.Cloneable`

Availability:

New as of JDK 1.1

Description

The `NumberFormat` class formats and parses numbers in a locale-specific manner. NumberFormat is an abstract class, but it provides factory methods that return useful instances of NumberFormat subclasses. These factory methods come in three groups:

- The `getCurrencyInstance()` methods return objects that format and parse currency values.
- The `getNumberInstance()` methods return objects that format and parse normal numbers.
- The `getPercentInstance()` methods return objects that format percentage values.

For example, to format a number as an Italian currency value, you can use this code:

```
double salary = 1234.56;
System.out.println
  (NumberFormat.getCurrencyInstance(Locale.ITALY).format(salary));
```

This produces the following output:

```
L 1.234,56
```

The `NumberFormat` class defines two field constants that represent the integer and fractional fields in a formatted number. These field constants create `FieldPosition` objects.

Class Summary

```
public abstract class java.text.NumberFormat extends java.text.Format
                      implements java.lang.Cloneable {
  // Constants
  public static final int FRACTION_FIELD;
  public static final int INTEGER_FIELD;
  // Class Methods
  public static Locale[] getAvailableLocales();
  public static final NumberFormat getCurrencyInstance();
  public static NumberFormat getCurrencyInstance(Locale inLocale);
  public static final NumberFormat getInstance();
  public static NumberFormat getInstance(Locale inLocale);
  public static final NumberFormat getNumberInstance();
  public static NumberFormat getNumberInstance(Locale inLocale);
  public static final NumberFormat getPercentInstance();
  public static NumberFormat getPercentInstance(Locale inLocale);
  // Instance Methods
  public Object clone();
  public boolean equals(Object obj);
  public final String format(double number);
  public final String format(long number);
  public final StringBuffer format(Object number, StringBuffer toAppendTo,
                              FieldPosition pos);
  public abstract StringBuffer format(double number,
                                 StringBuffer toAppendTo, FieldPosition pos);
  public abstract StringBuffer format(long number, StringBuffer toAppendTo,
                                 FieldPosition pos);
  public int getMaximumFractionDigits();
  public int getMaximumIntegerDigits();
  public int getMinimumFractionDigits();
  public int getMinimumIntegerDigits();
  public int hashCode();
  public boolean isGroupingUsed();
  public boolean isParseIntegerOnly();
  public Number parse(String text);
  public abstract Number parse(String text, ParsePosition parsePosition);
  public final Object parseObject(String source,
                        ParsePosition parsePosition);
  public void setGroupingUsed(boolean newValue);
  public void setMaximumFractionDigits(int newValue);
  public void setMaximumIntegerDigits(int newValue);
  public void setMinimumFractionDigits(int newValue);
  public void setMinimumIntegerDigits(int newValue);
  public void setParseIntegerOnly(boolean value);
}
```

Constants

FRACTION_FIELD

public final static int FRACTION_FIELD

Description

A field constant that represents the fractional part of the number.

INTEGER_FIELD

public final static int INTEGER_FIELD

Description

A field constant that represents the integer part of the number.

Class Methods

getAvailableLocales

public static Locale[] getAvailableLocales()

Returns

> An array of `Locale` objects.

Description

> This method returns an array of the `Locale` objects for which this class can create `NumberFormat` objects.

getCurrencyInstance

public static final NumberFormat getCurrencyInstance()

Returns

> A `NumberFormat` appropriate for the default `Locale` that formats currency values.

Description

> This method creates a `NumberFormat` that formats and parses currency values in the default locale.

public static NumberFormat getCurrencyInstance(Locale inLocale)

Parameters

> `inLocale`
>
> > The `Locale` to use.

Returns

> A `NumberFormat` appropriate for the given `Locale` that formats currency values.

Description

> This method creates a `NumberFormat` that formats and parses currency values in the given locale.

getInstance

public static final NumberFormat getInstance()

Returns

A default `NumberFormat` appropriate for the default `Locale`.

Description

This method creates a default `NumberFormat` that formats and parses values in the default locale.

public static NumberFormat getInstance(Locale inLocale)

Parameters

`inLocale`

The `Locale` to use.

Returns

A default `NumberFormat` appropriate for the given `Locale`.

Description

This method creates a `NumberFormat` that formats and parses values in the given locale.

getNumberInstance

public static final NumberFormat getNumberInstance()

Returns

A `NumberFormat` appropriate for the default `Locale` that formats normal numbers.

Description

This method creates a `NumberFormat` that formats and parses number values in the default locale.

public static NumberFormat getNumberInstance(Locale inLocale)

Parameters

`inLocale`

The `Locale` to use.

Returns

A `NumberFormat` appropriate for the given `Locale` that formats normal numbers.

Description

This method creates a `NumberFormat` that formats and parses number values in the given locale.

getPercentInstance

public static final NumberFormat getPercentInstance()

Returns

A `NumberFormat` appropriate for the default `Locale` that formats percentage values.

Description

This method creates a `NumberFormat` that formats and parses percent values in the default locale.

public static NumberFormat getPercentInstance(Locale inLocale)

Parameters

`inLocale`

The `Locale` to use.

Returns

A `NumberFormat` appropriate for the given `Locale` that formats percentage values.

Description

This method creates a `NumberFormat` that formats and parses percent values in the given locale.

Instance Methods

clone

public Object clone()

Returns

A copy of this `NumberFormat`.

Overrides

`Format.clone()`

Description

This method creates a copy of this `NumberFormat` and returns it.

equals

public boolean equals(Object obj)

Parameters

`obj`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `obj` is an instance of `NumberFormat` and is equivalent to this `NumberFormat`.

format

public final String format(double number)

Parameters

`number`

The `double` value to be formatted.

Returns

A string that contains a formatted representation of the value.

Description

This method formats the given number and returns the result as a string.

public final String format(long number)

Parameters

`number`

The `long` value to be formatted.

Returns

A string that contains a formatted representation of the value.

Description

This method formats the given number and returns the result as a string.

`public final StringBuffer format(Object number, StringBuffer toAppendTo, FieldPosition pos)`

Parameters

`number`

The object to be formatted.

`toAppendTo`

A `StringBuffer` on which to append the formatted information.

`pos`

A number field.

Returns

The given buffer `toAppendTo` with the formatted representation of the object appended to it.

Overrides

`Format.format(Object, StringBuffer, FieldPosition)`

Description

This method formats the given object and appends the result to the given `StringBuffer`. If `pos` refers to one of the number fields, its beginning and ending indices are filled with the beginning and ending positions of the given field in the resulting formatted string.

`public abstract StringBuffer format(double number, StringBuffer toAppendTo, FieldPosition pos)`

Parameters

`number`

The `double` value to be formatted.

`toAppendTo`

A `StringBuffer` on which to append the formatted information.

`pos`

A number field.

Returns

The given buffer `toAppendTo` with the formatted representation of the object appended to it.

Description

This method formats the given number and appends the result to the given `StringBuffer`. If `pos` refers to one of the number fields, its beginning and ending indices are filled with the beginning and ending positions of the given field in the resulting formatted string.

`public abstract StringBuffer format(long number, StringBuffer toAppendTo, FieldPosition pos)`

Parameters

`number`

The `long` value to be formatted.

`toAppendTo`

A `StringBuffer` on which to append the formatted information.

`pos`

A number field.

Returns

The given buffer `toAppendTo` with the formatted representation of the object appended to it.

Description

This method formats the given number and appends the result to the given `StringBuffer`. If `pos` refers to one of the number fields, its beginning and ending indices are filled with the beginning and ending positions of the given field in the resulting formatted string.

getMaximumFractionDigits

public int getMaximumFractionDigits()

Returns

The maximum number of digits allowed in the fraction portion.

Description

This method returns the maximum number of digits that can be in the fraction part of the number.

getMaximumIntegerDigits

public int getMaximumIntegerDigits()

Returns

The maximum number of digits allowed in the integer portion.

Description

This method returns the maximum number of digits that can be in the integer part of the number.

getMinimumFractionDigits

public int getMinimumFractionDigits()

Returns

The minimum number of digits allowed in the fraction portion.

Description

This method returns the minimum number of digits that can be in the fraction part of the number.

getMinimumIntegerDigits

public int getMinimumIntegerDigits()

Returns

The minimum number of digits allowed in the integer portion.

Description

This method returns the minimum number of digits that can be in the integer part of the number.

hashCode

public int hashCode()

Returns

A hashcode for this object.

Overrides

`Object.hashCode()`

Description

This method returns a hashcode for this `NumberFormat`.

isGroupingUsed

public boolean isGroupingUsed()

Returns

A `boolean` value that indicates whether or not this `NumberFormat` uses a grouping character to break up long sequences of digits in the integer part of a number.

Description

This method returns `true` if this `NumberFormat` uses a grouping character. For example, it is common in the United States to use a comma as a grouping character: `1,234.56`.

isParseIntegerOnly

public boolean isParseIntegerOnly()

Returns

A `boolean` value that indicates whether or not this `NumberFormat` parses only integers.

Description

This method returns `true` if this `NumberFormat` parses only integers.

parse

public Number parse(String text) throws ParseException

Parameters

`text`

The string to be parsed.

Returns

The `Number` object represented by the given string.

Throws

`ParseException`

If the text cannot be parsed as a number.

Description

This method parses a number from the given string, starting from the beginning of the string.

`public abstract Number parse(String text, ParsePosition parsePosition)`

Parameters

`text`

The string to be parsed.

`parsePosition`

A `ParsePosition` object that specifies a position in the string.

Returns

The `Number` object represented by the text starting at the given position.

Description

This method parses a number from the given string, starting from the given position. After the string has been parsed, the given `ParsePosition` object is updated so that its index is after the parsed text.

parseObject

`public final Object parseObject(String source, ParsePosition parsePosition)`

Parameters

`source`

The string to be parsed.

`parsePosition`

A `ParsePosition` object that specifies a position in the string.

Returns

The object represented by the text starting at the given position.

Overrides

`Format.parseObject(String, ParsePosition)`

Description

This method parses a number from the given string, starting from the given position. After the string has been parsed, the given `ParsePosition` object is updated so that its index is after the parsed text.

setGroupingUsed

public void setGroupingUsed(boolean newValue)

Parameters

`newValue`

The new grouping flag.

Description

This method sets whether or not this `NumberFormat` uses a grouping character to break up long sequences of digits in the integer part of a number. For example, it is common in the United States to use a comma as a grouping character: `1,234.56`.

setMaximumFractionDigits

public void setMaximumFractionDigits(int newValue)

Parameters

`newValue`

The new maximum number of fraction digits.

Description

This method sets the maximum number of digits that may be present in the fraction part of the number. The maximum value must be greater than the minimum number of fraction digits allowed. If the value is less than the current minimum, the minimum is also set to this value.

setMaximumIntegerDigits

public void setMaximumIntegerDigits(int newValue)

Parameters

`newValue`

The new maximum number of integer digits.

Description

This method sets the maximum number of digits that may be present in the integer part of the number. The maximum value must be greater than the minimum number of integer digits allowed. If the value is less than the current minimum, the minimum is also set to this value.

setMinimumFractionDigits

public void setMinimumFractionDigits(int newValue)

Parameters

`newValue`

The new minimum number of fraction digits.

Description

This method sets the minimum number of digits that may be present in the fraction part of the number. The minimum value must be less than the maximum number of fraction digits allowed. If the value is greater than the current maximum, the maximum is also set to this value.

setMinimumIntegerDigits

public void setMinimumIntegerDigits(int newValue)

Parameters

`newValue`

The new minimum number of integer digits.

Description

This method sets the minimum number of digits that may be present in the integer part of the number. The minimum value must be less than the maximum number of integer digits allowed. If the value is greater than the current maximum, the maximum is also set to this value.

setParseIntegerOnly

public void setParseIntegerOnly(boolean value)

Parameters

`value`

The new parsing flag.

Description

This method sets whether or not this `NumberFormat` parses only integers. If the value is `true`, this `NumberFormat` only parse integers. Otherwise it parses both the integer and fractional parts of numbers.

Inherited Methods

Method	Inherited From	Method	Inherited From
`finalize()`	`Object`	`format(Object)`	`Format`
`getClass()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`parseObject(String)`	`Format`
`toString()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`ChoiceFormat`, `DecimalFormat`, `FieldPosition`, `Format`, `Number`, `ParseException`, `ParsePosition`, `String`, `StringBuffer`

RuleBasedCollator

Name

RuleBasedCollator

Synopsis

Class Name:

```
java.text.RuleBasedCollator
```

Superclass:

```
java.text.Collator
```

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `RuleBasedCollator` class is a concrete subclass of `Collator` that can compare strings using a table of rules. The rules for many locales already exist. To get a useful `Collator` for a given locale, use the `getInstance(Locale)` factory method of `Collator`.

If you need a special-purpose `Collator` or a `Collator` for a new locale, you can create your own `RuleBasedCollator` from a string that represents the rules. The rules are expressed in three primary forms:

```
[relation] [text]
[reset] [text]
[modifier]
```

The rules can be chained together. The only modifier is the `@` character, which specifies that all diacriticals are sorted backwards, as in French.

The valid relations are:

`>`

Greater than as a primary difference

`;`

Greater than as a secondary difference or difference in accent

`,`

Greater than as a tertiary difference or difference in case

`=`

Equal

For example `"<a<b"` is two chained rules that state that `'a'` is greater than all ignorable characters, and `'b'` is greater than `'a'`. To expand this rule to include capitals, use `"<a,A<b,B"`.

A reset, specified by the & character, is used to insert rules in an existing list of chained rules. For example, we can add a rule to sort `'e'` with an umlaut (Unicode 0308) after plain `'e'`. The existing rules might look like `"<a<b<c<d<e<f"`. We can add the following reset, `"&e;e\u0308"`, so that the complete rule table looks like `"<a<b<c<d<e<f&e;e\u0308"`.

Class Summary

```
public class java.text.RuleBasedCollator extends java.text.Collator {
  // Constructors
  public RuleBasedCollator(String rules);
  // Instance Methods
  public Object clone();
  public int compare(String source, String target);
  public boolean equals(Object obj);
  public CollationElementIterator
         getCollationElementIterator( String source);
  public CollationKey getCollationKey(String source);
  public String getRules();
  public int hashCode();
}
```

Constructors

RuleBasedCollator

public RuleBasedCollator(String rules) throws ParseException

Parameters

`rules`

A string that contains the rules.

Throws

`ParseException`

If the given rules are incorrectly formed.

Description

This constructor creates a `RuleBasedCollator` with the given rules.

Instance Methods

clone

public Object clone()

Returns

A copy of this `RuledBasedCollator`.

Overrides

`Collator.clone()`

Description

This method creates a copy of this `RuledBasedCollator` and returns it.

compare

public int compare(String source, String target)

Parameters

`source`

The source string.

`target`

The target string.

Returns

`-1` if `source` is less than `target`, `0` if the strings are equal, or `1` if `source` is greater than `target`.

Overrides

`Collator.compare()`

Description

This method compares the given strings according to the rules for this `RuleBasedCollator` and returns a value that indicates their relationship. If either of the strings are compared more than once, a `CollationKey` should be used instead.

equals

public boolean equals(Object obj)

Parameters

`obj`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Collator.equals()`

Description

This method returns `true` if `obj` is an instance of `RuleBasedCollator` and is equivalent to this `RuleBasedCollator`.

getCollationElementIterator

```
public CollationElementIterator getCollationElementIterator( String source)
```

Parameters

`source`

The source string.

Returns

A `CollationElementIterator` for the given string.

Description

This method generates a `CollationElementIterator` for the given string.

getCollationKey

public CollationKey getCollationKey(String source)

Parameters

`source`

The source string.

Returns

A `CollationKey` for the given string.

Overrides

`Collator.getCollationKey()`

Description

This method generates a `CollationKey` for the given string. The returned object can be compared with other `CollationKey` objects using `CollationKey.compareTo()`. This comparison is faster than using `RuleBasedCollator.compare()`, so if the same string is used for many comparisons, you should use `CollationKey` objects.

getRules

public String getRules()

Returns

The rules string for this `RuleBasedCollator`.

Description

This method returns a string that contains the rules that this `RuleBasedCollator` is using.

hashCode

public int hashCode()

Returns

A hashcode for this object.

Overrides

`Collator.hashCode()`

Description

This method returns a hashcode for this `RuleBasedCollator`.

Inherited Methods

Method	Inherited From	Method	Inherited From
`equals(String, String)`	`Collator`	`finalize()`	`Object`
`getClass()`	`Object`	`getDecomposition()`	`Collator`
`getStrength()`	`Collator`	`notify()`	`Object`
`notifyAll()`	`Object`	`setDecomposition(int)`	`Collator`
`setStrength(int)`	`Collator`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`CollationKey`, `CollationElementIterator`, `Collator`, `Locale`, `ParseException`, `String`

SimpleDateFormat

Name

SimpleDateFormat

Synopsis

Class Name:

`java.text.SimpleDateFormat`

Superclass:

`java.text.DateFormat`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `SimpleDateFormat` class is a concrete subclass of `DateFormat` that formats and parses dates and times using a formatting pattern. Typically, you do not need to instantiate `SimpleDateFormat` yourself. Instead, the factory methods of `DateFormat` return instances of `SimpleDateFormat` that are appropriate for particular locales.

However, if you need a specialized date and time format, you can instantiate your own `SimpleDateFormat` using a pattern string. You can also modify the formatting pattern of an existing `SimpleDateFormat` object using the `applyPattern()` method. The following symbols are significant in the pattern string.

Symbol	**Description**	**Example**	**Type**
`G`	Era	AD	Text
`y`	Year	1997	Numeric
`M`	Month in year	3 or March	Text or numeric
`d`	Day in month	4	Numeric
`h`	Hour in A.M./P.M. (1-12)	2	Numeric
`H`	Hour in day (0-23)	14	Numeric
`m`	Minute in hour	33	Numeric
`s`	Second in minute	21	Numeric
`S`	Milliseconds	333	Numeric
`E`	Day of week	Thursday	Text
`D`	Day in year	63	Numeric
`F`	Day of week of month	1	Numeric
`w`	Week in year	9	Numeric
`W`	Week in month	1	Numeric
`a`	A.M./P.M.	P.M.	Text
`k`	Hour in day (1-24)	14	Numeric
`K`	Hour in A.M./P.M. (0-11)	2	Numeric
`z`	Time zone	EST	Text

Symbols that are numeric can be repeated to specify a minimum number of digits. For example, `"hh"` produces an hour field that is always at least two digits, like `"02"`. Symbols that are textual can be repeated to specify whether the short form or the long form of the text string is used, if there are both short and long forms. If four or more symbols are specified, the long form is used; otherwise the short form is used. For example, `"E"` produces a short form of the day of the week, such as `"Tue"`, while `"EEEE"` produces the long form, such as `"Tuesday"`. For the month of the year, if one or two `"M"` symbols are used, the field is numeric. If three or more `"M"` symbols are used, the field is textual.

Single quotes can be used to specify literal text that should be included in the formatted output, and any unrecognized symbol is treated as literal text. For example, the following pattern:

```
hh:mm a 'in the' zzzz 'zone.'
```

produces output like:

```
02:33 PM in the Eastern Standard Time zone.
```

Internally, the `SimpleDataFormat` class uses a `DateFormatSymbols` object to get the date and time strings that are appropriate for a particular locale. If you want to modify these strings, you can get the `DateFormatSymbols` object by calling `getDateFormatSymbols()`.

Class Summary

```
public class java.text.SimpleDateFormat  extends java.text.DateFormat {
  // Constructors
  public SimpleDateFormat();
  public SimpleDateFormat(String pattern);
  public SimpleDateFormat(String pattern, Locale loc);
  public SimpleDateFormat(String pattern, DateFormatSymbols formatData);
  // Instance Methods
  public void applyLocalizedPattern(String pattern);
  public void applyPattern(String pattern);
  public Object clone();
  public boolean equals(Object obj);
  public StringBuffer format(Date date, StringBuffer toAppendTo,
                      FieldPosition pos);
  public DateFormatSymbols getDateFormatSymbols();
  public int hashCode();
  public Date parse(String text, ParsePosition pos);
  public void setDateFormatSymbols(DateFormatSymbols newFormatSymbols);
  public String toLocalizedPattern();
  public String toPattern();
}
```

Constructors

SimpleDateFormat

public SimpleDateFormat()

Description

> This constructor creates a `SimpleDateFormat` that uses a default formatting pattern and `DateFormatSymbols` that are appropriate for the default locale. It produces the same result as calling `DateFormat.getDateTimeInstance(DateFormat.SHORT, DateFormat.SHORT)`.

public SimpleDateFormat(String pattern)

Parameters

> `pattern`
>
> > The pattern string.

Description

> This constructor creates a `SimpleDateFormat` that uses the given formatting pattern and a `DateFormatSymbols` object that is appropriate for the default locale.

public SimpleDateFormat(String pattern, Locale loc)

Parameters

`pattern`

The pattern string.

`loc`

The `Locale` to use.

Description

This constructor creates a `SimpleDateFormat` that uses the given formatting pattern and a `DateFormatSymbols` object that is appropriate for the given locale.

`public SimpleDateFormat(String pattern, DateFormatSymbols formatData)`

Parameters

`pattern`

The pattern string.

`formatData`

The `DateFormatSymbols` to use.

Description

This constructor creates a `SimpleDateFormat` that uses the given formatting pattern and `DateFormatSymbols` object.

Instance Methods

applyLocalizedPattern

public void applyLocalizedPattern(String pattern)

Parameters

`pattern`

The pattern string.

Description

This method tells this `SimpleDateFormat` to use the given formating pattern to format and parse dates and times. The pattern string is assumed to have been localized to the `DateFormatSymbols` object this `SimpleDateFormat` uses.

applyPattern

public void applyPattern(String pattern)

Parameters

`pattern`

The pattern string.

Description

This method tells this `SimpleDateFormat` to use the given formatting pattern to format and parse dates and times. The pattern string is localized to the `DateFormatSymbols` object this `SimpleDateFormat` uses.

clone

public Object clone()

Returns

A copy of this `SimpleDateFormat`.

Overrides

`DateFormat.clone()`

Description

This method creates a copy of this `SimpleDateFormat` and returns it.

equals

public boolean equals(Object obj)

Parameters

`obj`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`DateFormat.equals()`

Description

This method returns `true` if `obj` is an instance of `SimpleDateFormat` and is equivalent to this `SimpleDateFormat`.

format

```
public StringBuffer format(Date date, StringBuffer toAppendTo, FieldPosition pos)
```

Parameters

`date`

The `Date` object to be formatted.

`toAppendTo`

A `StringBuffer` on which to append the formatted information.

`pos`

A date or time field.

Returns

The given buffer `toAppendTo` with the formatted representation of the object appended to it.

Overrides

`DateFormat.format(Date, StringBuffer, FieldPosition)`

Description

This method formats the given date and appends the result to the given `StringBuffer`. If `pos` refers to one of the time or date fields, its beginning and ending indexes are filled with the beginning and ending positions of the given field in the resulting formatted string.

getDateFormatSymbols

public DateFormatSymbols getDateFormatSymbols()

Returns

The `DateFormatSymbols` object used by this `SimpleDateFormat`.

Description

This method returns the `DateFormatSymbols` object that this `SimpleDateFormat` uses internally.

hashCode

public int hashCode()

Returns

A hashcode for this object.

Overrides

`DateFormat.hashCode()`

Description

This method returns a hashcode for this `SimpleDateFormat`.

parse

public Date parse(String text, ParsePosition pos)

Parameters

`text`

The string to be parsed.

`pos`

A `ParsePosition` object that specifies a position in the string.

Returns

The `Date` object represented by the text starting at the given position.

Overrides

`DateFormat.parse(String, ParsePosition)`

Description

This method parses a date from the given string, starting from the given position. After the string has been parsed, the given `ParsePosition` object is updated so that its index is after the parsed text.

setDateFormatSymbols

```
public void setDateFormatSymbols( DateFormatSymbols newFormatSymbols)
```

Parameters

`newFormatSymbols`

The new `DateFormatSymbols` object to use.

Description

This method sets the `DateFormatSymbols` object that this `SimpleDateFormat` uses internally.

toLocalizedPattern

public String toLocalizedPattern()

Returns

The pattern string of this `SimpleDateFormat`.

Description

This method returns the pattern string of this `SimpleDateFormat`, localized with the `DateFormatSymbols` object of this `SimpleDateFormat`.

toPattern

public String toPattern()

Returns

The pattern string of this `SimpleDateFormat`.

Description

This method returns the pattern string of this `SimpleDateFormat`.

Inherited Methods

Method	Inherited From Method	Inherited From

`finalize()`	`Object`	`format(Object)`	`Format`
`format(Date)`	`DateFormat`	`format(Object, StringBuffer, FieldPosition)`	`DateFormat`
`getCalendar()`	`DateFormat`	`getClass()`	`Object`
`getNumberFormat()`	`DateFormat`	`getTimeZone()`	`DateFormat`
`isLenient()`	`DateFormat`	`notify()`	`Object`
`notifyAll()`	`Object`	`parse(String)`	`DateFormat`
`parseObject(String)`	`Format`	`parseObject(String, ParsePosition)`	`DateFormat`
`setCalendar(Calendar)`	`DateFormat`	`setLenient(boolean)`	`DateFormat`
`setNumberFormat(NumberFormat)`	`DateFormat`	`setTimeZone(TimeZone)`	`DateFormat`
`toString()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`Calendar`, `Date`, `DateFormat`, `DateFormatSymbols`, `FieldPosition`, `Format`, `Locale`, `ParsePosition`, `String`, `StringBuffer`, `TimeZone`

PREVIOUS HOME NEXT
RuleBasedCollator BOOK INDEX StringCharacterIterator

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

StringCharacterIterator

Name

StringCharacterIterator

Synopsis

Class Name:

```
java.text.StringCharacterIterator
```

Superclass:

```
java.lang.Object
```

Immediate Subclasses:

None

Interfaces Implemented:

```
java.text.CharacterIterator
```

Availability:

New as of JDK 1.1

Description

The `StringCharacterIterator` class can move bidirectionally through a character string. In other words, the class iterates through the characters in a `String`. The class implements the `CharacterIterator` interface. The class is used by `BreakIterator` to find boundaries in text strings.

Class Summary

```
public final class java.text.StringCharacterIterator
                    extends java.lang.Object
                    implements java.text.CharacterIterator {
  // Constructors
  public StringCharacterIterator(String text);
  public StringCharacterIterator(String text, int pos);
  public StringCharacterIterator(String text, int begin, int end, int pos);
  // Instance Methods
  public Object clone();
  public char current();
  public boolean equals(Object obj);
  public char first();
  public int getBeginIndex();
  public int getEndIndex();
  public int getIndex();
  public int hashCode();
  public char last();
  public char next();
  public char previous();
  public char setIndex(int p);
}
```

Constructors

StringCharacterIterator

public StringCharacterIterator(String text)

Parameters

`text`

The `String` to use.

Description

This constructor creates a `StringCharacterIterator` that uses the given string. The initial index of the iterator is at the beginning of the string, or in other words, at index `0`.

public StringCharacterIterator(String text, int pos)

Parameters

`text`

The `String` to use.

`pos`

The initial position.

Description

This constructor creates a `StringCharacterIterator` that uses the given string. The initial index of the iterator is set to the given initial position.

`public StringCharacterIterator(String text, int begin, int end, int pos)`

Parameters

`text`

The `String` to use.

`begin`

The beginning index.

`end`

The ending index.

`pos`

The initial position.

Description

This constructor creates a `StringCharacterIterator` that uses the specified range of the given string. In other words, the iterator uses the sequence of text from the specified beginning index to the specified ending index. The initial index of the iterator is set to the given initial position.

Instance Methods

clone

public Object clone()

Returns

> A copy of this `StringCharacterIterator`.

Implements

> `CharacterIterator.clone()`

Overrides

> `Object.clone()`

Description

> This method creates a copy of this `StringCharacterIterator` and returns it.

current

public char current()

Returns

> The character at the current position of this `StringCharacterIterator` or `DONE` if the current position is not within the text sequence.

Implements

> `CharacterIterator.current()`

Description

> This method returns the character at the current position of this `CharacterIterator`. The current position is returned by `getIndex()`.

equals

public boolean equals(Object obj)

Parameters

`obj`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `obj` is an instance of `StringCharacterIterator` and is equivalent to this `StringCharacterIterator`.

first

public char first()

Returns

The first character in this `StringCharacterIterator`.

Implements

`CharacterIterator.first()`

Description

This method returns the character at the first position in this `StringCharacterIterator`. The first position is returned by `getBeginIndex()`. The current position of the iterator is set to this position.

getBeginIndex

public int getBeginIndex()

Returns

The index of the first character in this `StringCharacterIterator`.

Implements

`CharacterIterator.getBeginIndex()`

Description

This method returns the index of the beginning of the text for this `StringCharacterIterator`.

getEndIndex

public int getEndIndex()

Returns

The index after the last character in this `StringCharacterIterator`.

Implements

`CharacterIterator.getEndIndex()`

Description

This method returns the index of the character following the end of the text for this `StringCharacterIterator`.

getIndex

public int getIndex()

Returns

The index of the current character in this `StringCharacterIterator`.

Description

This method returns the current position, or index, of this `StringCharacterIterator`.

hashCode

public int hashCode()

Returns

A hashcode for this object.

Overrides

`Object.hashCode()`

Description

This method returns a hashcode for this `StringCharacterIterator`.

last

public char last()

Returns

The last character in this `StringCharacterIterator`.

Implements

`CharacterIterator.last()`

Description

This method returns the character at the ending position of this `StringCharacterIterator`. The last position is the value of `getEndIndex()-1`. The current position of the iterator is set to this position.

next

public char next()

Returns

The next character in this `StringCharacterIterator` or `DONE` if the current position is already at the end of the text.

Implements

`CharacterIterator.next()`

Description

This method increments the current position of this `StringCharacterIterator` by 1 and returns the character at the new position. If the current position is already at `getEndIndex()`, the position is not changed and `DONE` is returned.

previous

public char previous()

Returns

The previous character in this `StringCharacterIterator` or `DONE` if the current position is already at the beginning of the text.

Implements

`CharacterIterator.previous()`

Description

This method decrements the current position of this `StringCharacterIterator` by 1 and returns the character at the new position. If the current position is already at `getBeginIndex()`, the position is not changed and `DONE` is returned.

setIndex

public char setIndex(int p)

Parameters

`p`

The new position.

Returns

The character at the specified position in this `StringCharacterIterator`.

Throws

`IllegalArgumentException`

If the given position is not between `getBeginIndex()` and `getEndIndex()-1`.

Implements

`CharacterIterator.setIndex()`

Description

This method sets the current position, or index, of this `StringCharacterIterator` to the given position.

Inherited Methods

Method	**Inherited From**	**Method**	**Inherited From**
`finalize()`	`Object`	`getClass()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`toString()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`BreakIterator`, `CharacterIterator`, `String`

PREVIOUS SimpleDateFormat | HOME BOOK INDEX | NEXT BitSet

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Calendar

Name

Calendar

Synopsis

Class Name:

`java.util.Calendar`

Superclass:

`java.lang.Object`

Immediate Subclasses:

`java.util.GregorianCalendar`

Interfaces Implemented:

`java.lang.Cloneable`, `java.io.Serializable`

Availability:

New as of JDK 1.1

Description

The `Calendar` class is an `abstract`class that is used to convert between `Date` objects, which represent points in time, and calendar fields, like months or days of the week. The JDK 1.0 `Date` class included calendar and text-formatting methods. As of JDK 1.1, both of these functions have been split off from `Date` in order to support internationalization. As of JDK 1.1, `Date` represents only a point in time, measured in milliseconds. A subclass of `Calendar` examines the `Date` in the context of a particular calendar system; a `Calendar` instance is a locale-sensitive object. Also as of JDK 1.1, the `java.text.DateFormat` class generates and parses strings representing points in time.

`Calendar` defines a number of symbolic constants. They represent either fields or values. For example, `MONTH` is a field constant. It can be passed to `get()` and `set()` to retrieve and adjust the month. `AUGUST`, on the other hand, represents a particular month value. Calling `get(Calendar.MONTH)` could return `Calendar.AUGUST`.

Internally, `Calendar` keeps track of a point in time in two ways. First, a "raw" value is maintained, which is simply a count of milliseconds since midnight, January 1, 1970 GMT, or, in other words, a `Date` object. Second, the calendar keeps track of a number of fields, which are the values that are specific to the `Calendar` type. These are values such as day of the week, day of the month, and month. The raw millisecond value can be calculated from the field values, or vice versa.

When a `Date` object is computed from the time fields, there may be insufficient information to compute the raw millisecond value. For example, the year and the month could be set, but not the day of the month. In this case, `Calendar` uses default information to fill in the missing fields. For `GregorianCalendar`, the default field values are taken from the date of the epoch, or midnight, January 1, 1970 GMT.

Another problem that can arise when computing a `Date` object from the time fields is that of inconsistent information in the fields. For example, the time fields could specify "Sunday, March 8, 1997" when in fact March 8, 1997 is a Saturday. If the time fields contain inconsistent information, `Calendar` gives preference to the combinations of fields in the following order:

1. month and day of the week
2. month, week of the month, and day of the week
3. month, day of the week in the month, and day of the week
4. day of the year
5. day of the week and week of the year
6. hour of the day
7. A.M./P.M. and hour of A.M./P.M.

There is also the possibility of ambiguity for certain points in time, so the following rules apply. The time 24:00:00 belongs to the next day and midnight is an A.M. time, while noon is a P.M. time.

Class Summary

```
public abstract class java.util.Calendar extends java.lang.Object
                      implements java.lang.Cloneable, java.io.Serializable {
```

```
// Constants
public final static int AM;
public final static int AM_PM;
public final static int APRIL;
public final static int AUGUST;
public final static int DATE;
public final static int DAY_OF_MONTH;
public final static int DAY_OF_WEEK;
public final static int DAY_OF_WEEK_IN_MONTH;
public final static int DAY_OF_YEAR;
public final static int DECEMBER;
public final static int DST_OFFSET;
public final static int ERA;
public final static int FEBRUARY;
public final static int FIELD_COUNT;
public final static int FRIDAY;
public final static int HOUR;
public final static int HOUR_OF_DAY;
public final static int JANUARY;
public final static int JULY;
public final static int JUNE;
public final static int MARCH;
public final static int MAY;
public final static int MILLISECOND;
public final static int MINUTE;
public final static int MONDAY;
public final static int MONTH;
public final static int NOVEMBER;
public final static int OCTOBER;
public final static int PM;
public final static int SATURDAY;
public final static int SECOND;
public final static int SEPTEMBER;
public final static int SUNDAY;
public final static int THURSDAY;
public final static int TUESDAY;
public final static int UNDECIMBER;
public final static int WEDNESDAY;
public final static int WEEK_OF_MONTH;
public final static int WEEK_OF_YEAR;
public final static int YEAR;
public final static int ZONE_OFFSET;
// Variables
protected boolean areFieldsSet;
protected int[] fields;
protected boolean[] isSet;
protected boolean isTimeSet;
protected long time;
// Constructors
```

```
  protected Calendar();
  protected Calendar(TimeZone zone, Locale aLocale);
  // Class Methods
  public static synchronized Locale[] getAvailableLocales();
  public static synchronized Calendar getInstance();
  public static synchronized Calendar getInstance(TimeZone zone);
  public static synchronized Calendar getInstance(Locale aLocale);
  public static synchronized Calendar getInstance(TimeZone zone,
                                      Locale aLocale);
  // Instance Methods
  public abstract void add(int field, int amount);
  public abstract boolean after(Object when);
  public abstract boolean before(Object when);
  public final void clear();
  public final void clear(int field);
  public Object clone();
  public abstract boolean equals(Object when);
  public final int get(int field);
  public int getFirstDayOfWeek();
  public abstract int getGreatestMinimum(int field);
  public abstract int getLeastMaximum(int field);
  public abstract int getMaximum(int field);
  public int getMinimalDaysInFirstWeek();
  public abstract int getMinimum(int field);
  public final Date getTime();
  public TimeZone getTimeZone();
  public boolean isLenient();
  public final boolean isSet(int field);
  public abstract void roll(int field, boolean up);
  public final void set(int field, int value);
  public final void set(int year, int month, int date);
  public final void set(int year, int month, int date,
                     int hour, int minute);
  public final void set(int year, int month, int date, int hour,
                     int minute, int second);
  public void setFirstDayOfWeek(int value);
  public void setLenient(boolean lenient);
  public void setMinimalDaysInFirstWeek(int value);
  public final void setTime(Date date);
  public void setTimeZone(TimeZone value);
  // Protected Instance Methods
  protected void complete();
  protected abstract void computeFields();
  protected abstract void computeTime();
  protected long getTimeInMillis();
  protected final int internalGet(int field);
  protected void setTimeInMillis(long millis);
}
```

Constants

AM

public final static int AM

Description

A constant value that represents morning times.

AM_PM

public final static int AM_PM

Description

A field constant that represents the A.M./P.M. flag of this object.

APRIL

public final static int APRIL

Description

A constant value that represents the month of April.

AUGUST

public final static int AUGUST

Description

A constant value that represents the month of August.

DATE

public final static int DATE

Description

A field constant that represents the day of the month of this object.

DAY_OF_MONTH

public final static int DAY_OF_MONTH

Description

> A field constant that represents the day of the month of this object. This field is synonymous with `DATE`.

DAY_OF_WEEK

public final static int DAY_OF_WEEK

Description

> A field constant that represents the day of the week of this object.

DAY_OF_WEEK_IN_MONTH

public final static int DAY_OF_WEEK_IN_MONTH

Description

> A field constant that represents the day of the week in the current month. For example, February 10, 1997, has a `DAY_OF_WEEK_IN_MONTH` value of `2` because it is the second Monday in February for that year.

DAY_OF_YEAR

public final static int DAY_OF_YEAR

Description

> A field constant that represents the day of the year of this object. January 1 is the first day of the year.

DECEMBER

public final static int DECEMBER

Description

> A constant value that represents the month of December.

DST_OFFSET

public final static int DST_OFFSET

Description

A field constant that represents the offset due to daylight savings time, in milliseconds, of this object.

ERA

public final static int ERA

Description

A field constant that represents the era of this object. A Gregorian calendar has two eras, BC and AD.

FEBRUARY

public final static int FEBRUARY

Description

A constant value that represents the month of February.

FIELD_COUNT

public final static int FIELD_COUNT

Description

A constant that represents the number of attribute fields for `Calendar` objects.

FRIDAY

public final static int FRIDAY

Description

A constant value that represents the day Friday.

HOUR

public final static int HOUR

Description

A field constant that represents the hour of this object.

HOUR_OF_DAY

public final static int HOUR_OF_DAY

Description

> A field constant that represents the hour of the day of this object. A time of 1:00 P.M. has an `HOUR` value of 1, but an `HOUR_OF_DAY` value of 13.

JANUARY

public final static int JANUARY

Description

> A constant value that represents the month of January.

JULY

public final static int JULY

Description

> A constant value that represents the month of July.

JUNE

public final static int JUNE

Description

> A constant value that represents the month of June.

MARCH

public final static int MARCH

Description

> A constant value that represents the month of March.

MAY

public final static int MAY

Description

A constant value that represents the month of May.

MILLISECOND

public final static int MILLISECOND

Description

A field constant that represents the milliseconds of this object.

MINUTE

public final static int MINUTE

Description

A field constant that represents the minutes of this object.

MONDAY

public final static int MONDAY

Description

A constant value that represents the day Monday.

MONTH

public final static int MONTH

Description

A field constant that represents the month of this object.

NOVEMBER

public final static int NOVEMBER

Description

A constant value that represents the month of November.

OCTOBER

public final static int OCTOBER

Description

A constant value that represents the month of October.

PM

public final static int PM

Description

A constant value that represents afternoon and evening times.

SATURDAY

public final static int SATURDAY

Description

A constant value that represents the day Saturday.

SECOND

public final static int SECOND

Description

A field constant that represents the seconds of this object.

SEPTEMBER

public final static int SEPTEMBER

Description

A constant value that represents the month of September.

SUNDAY

public final static int SUNDAY

Description

A constant value that represents the day Sunday.

THURSDAY

public final static int THURSDAY

Description

A constant value that represents the day Thursday.

TUESDAY

public final static int TUESDAY

Description

A constant value that represents the day Tuesday.

UNDECIMBER

public final static int UNDECIMBER

Description

A constant value that represents the thirteenth month used in lunar calendars.

WEDNESDAY

public final static int WEDNESDAY

Description

A constant value that represents the day Wednesday.

WEEK_OF_MONTH

public final static int WEEK_OF_MONTH

Description

A field constant that represents the week of the month of this object.

WEEK_OF_YEAR

public final static int WEEK_OF_YEAR

Description

> A field constant that represents the week of the year of this object.

YEAR

public final static int YEAR

Description

> A field constant that represents the year of this object.

ZONE_OFFSET

public final static int ZONE_OFFSET

Description

> A field constant that represents the raw time zone offset, in milliseconds, of this object. The value should be added to GMT to get local time.

Variables

areFieldsSet

protected boolean areFieldsSet

Description

> A `boolean` value that indicates if the time fields of this `Calendar` have been set. These fields can be computed from the raw millisecond time value.

fields

protected int[] fields

Description

> An array that stores the time field values for this `Calendar`.

isSet

protected boolean[] isSet

Description

An array that contains a flag for each entry in the `fields` array. The value of each flag indicates if the corresponding entry in `fields` has been set for this `Calendar`.

isTimeSet

protected boolean isTimeSet

Description

A `boolean` value that indicates if the raw millisecond time value of this `Calendar` has been set. The value can be computed from the time fields.

time

protected long time

Description

The raw time value for this `Calendar`. The value is the number of milliseconds since midnight, January 1, 1970 GMT.

Constructors

Calendar

protected Calendar()

Description

This constructor creates `a Calendar` that uses the system's default time zone and locale. The default time zone is that returned by `TimeZone.getDefault()`. The default locale is that returned by `Locale.getDefault()`.

protected Calendar(TimeZone zone, Locale aLocale)

Parameters

`zone`

The `TimeZone` to use.

aLocale

The Locale to use.

Description

This constructor creates a Calendar that uses the supplied time zone and locale.

Class Methods

getAvailableLocales

public static synchronized Locale[] getAvailableLocales()

Returns

An array of Locale objects for which Calendar objects are installed.

Description

This method returns an array of locales that have corresponding Calendar objects.

getInstance

public static synchronized Calendar getInstance()

Returns

A Calendar for the default time zone and locale.

Description

This method returns a newly constructed Calendar for the default time zone and locale. Future implementations of this method may infer the subclass of Calendar to instantiate based on the default locale. However, the current implementation always returns a GregorianCalendar. The default time zone is that returned by TimeZone.getDefault(). The default locale is that returned by Locale.getDefault().

public static synchronized Calendar getInstance(TimeZone zone)

Parameters

zone

The `TimeZone` to use.

Returns

A `Calendar` for the given time zone and the default locale.

Description

This method returns a newly constructed `Calendar` for the given time zone and the default locale. Future implementations of this method may infer the subclass of `Calendar` to instantiate based on the default locale. However, the current implementation always returns a `GregorianCalendar`. The default locale is that returned by `Locale.getDefault()`.

public static synchronized Calendar getInstance(Locale aLocale)

Parameters

`aLocale`

The `Locale` to use.

Returns

A `Calendar` for the given locale and the default time zone.

Description

This method returns a newly constructed `Calendar` for the given locale and the default time zone. Future implementations of this method may infer the subclass of `Calendar` to instantiate based on the given locale. However, the current implementation always returns a `GregorianCalendar`. The default time zone is that returned by `TimeZone.getDefault()`.

public static synchronized Calendar getInstance(TimeZone zone, Locale aLocale)

Parameters

`zone`

The `TimeZone` to use.

`aLocale`

The `Locale` to use.

Returns

A `Calendar` for the given time zone and locale.

Description

This method returns a newly constructed `Calendar` for the given time zone and locale. Future implementations of this method may infer the subclass of `Calendar` to instantiate based on the given locale. However, the current implementation always returns a `GregorianCalendar`.

Instance Methods

add

public abstract void add(int field, int amount)

Parameters

`field`

The time field to be modified.

`amount`

The amount to add to the specified field value. This value can be negative.

Description

This method adds the given amount to the specified time field. For example, you can compute a date 90 days beyond the current date of this `Calendar` by calling `add(Calendar.DATE, 90)`.

after

public abstract boolean after(Object when)

Parameters

`when`

The object to compare to this `Calendar`.

Returns

`true` if this object is after `when`; `false` otherwise.

Description

This method returns `true` if `when` is a `Calendar` object whose value falls before the value of this `Calendar`.

before

public abstract boolean before(Object when)

Parameters

`when`

The object to compare to this `Calendar`.

Returns

`true` if this object is before `when`; `false` otherwise.

Description

This method returns `true` if `when` is a `Calendar` object whose value falls after the value of this `Calendar`.

clear

public final void clear()

Description

This method clears the values of all of the time fields of this `Calendar`.

public final void clear(int field)

Parameters

`field`

The time field to be cleared.

Description

This method clears the specified time field by setting its value to 0.

clone

public Object clone()

Returns

A copy of this `Calendar`.

Overrides

`Object.clone()`

Description

This method creates a copy of this `Calendar` and returns it. In other words, the returned `Calendar` has the same time field values and raw time value as this `Calendar`.

equals

public abstract boolean equals(Object when)

Parameters

`when`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `when` is an instance of `Calendar` and it contains the same value as the object this method is associated with.

get

public final int get(int field)

Parameters

`field`

The time field to be retrieved.

Returns

The value of the given time field.

Description

This method returns the value of the specified time field. If the fields of this `Calendar` have not been set, they are set from the raw time value before the requested field is returned.

getFirstDayOfWeek

public int getFirstDayOfWeek()

Returns

The first day of the week for this Calendar.

Description

This method returns the day that is considered the beginning of the week for this `Calendar`. This value is determined by the `Locale` of this `Calendar`. For example, the first day of the week in the United States is Sunday, while in France it is Monday.

getGreatestMinimum

public abstract int getGreatestMinimum(int field)

Parameters

`field`

A time field constant.

Returns

The highest minimum value for the given time field.

Description

This method returns the highest minimum value for the given time field, if the field has a range of minimum values. If the field does not have a range of minimum values, this method is equivalent to `getMinimum()`.

getLeastMaximum

public abstract int getLeastMaximum(int field)

Parameters

`field`

A time field constant.

Returns

The lowest maximum value for the given time field.

Description

This method returns the lowest maximum value for the given time field, if the field has a range of maximum values. If the field does not have a range of maximum values, this method is equivalent to `getMaximum()`. For example, for a `GregorianCalendar`, the lowest maximum value of `DATE_OF_MONTH` is 28.

getMaximum

public abstract int getMaximum(int field)

Parameters

`field`

A time field constant.

Returns

The maximum value for the given time field.

Description

This method returns the maximum value for the given time field. For example, for a `GregorianCalendar`, the maximum value of `DATE_OF_MONTH` is 31.

getMinimalDaysInFirstWeek

public int getMinimalDaysInFirstWeek()

Returns

The number of days that must be in the first week of the year.

Description

This method returns the number of days that must be in the first week of the year. For example, a value of 7 indicates that the first week of the year must be a full week, while a value of 1 indicates that the first week of the year can contain a single day. This value is determined by the `Locale` of this `Calendar`.

getMinimum

public abstract int getMinimum(int field)

Parameters

`field`

A time field constant.

Returns

The minimum value for the given time field.

Description

This method returns the minimum value for the given time field. For example, for a `GregorianCalendar`, the minimum value of `DATE_OF_MONTH` is 1.

getTime

public final Date getTime()

Returns

A `Date` object that represents the point in time represented by this `Calendar`.

Description

This method returns a newly created `Date` object that is constructed from the value returned by `getTimeInMillis()`.

getTimeZone

public TimeZone getTimeZone()

Returns

The `TimeZone` of this `Calendar`.

Description

This method returns the `TimeZone` object for this `Calendar`.

isLenient

public boolean isLenient()

Returns

A `boolean` value that indicates the leniency of this `Calendar`.

Description

This method returns the current leniency of this `Calendar`. A value of `false` indicates that the `Calendar` throws exceptions when questionable data is passed to it, while a value of `true` indicates that the `Calendar` makes its best guess to interpret questionable data. For example, if the `Calendar` is being lenient, a date such as March 135, 1997 is interpreted as the 134th day after March 1, 1997.

isSet

public final boolean isSet(int field)

Parameters

`field`

A time field constant.

Returns

`true` if the time field has been set; `false` otherwise.

Description

This method returns a `boolean` value that indicates whether or not the specified time field has been set.

roll

public abstract void roll(int field, boolean up)

Parameters

`field`

The time field to be adjusted.

`up`

A `boolean` value that indicates if the given field should be incremented.

Description

This method adds or subtracts one time unit from the given time field. For example, to increase the current date by one day, you can call `roll(Calendar.DATE, true)`.

The method maintains the field being rolled within its valid range. For example, in a calendar system that uses hours and minutes to measure time, rolling the minutes up from 59 sets that field to 0. By the same token, rolling that field down from 0 sets it to 59.

The `roll()` method does not adjust the value of any other field than the one specified by its `field` argument. In particular, for calendar systems that have months with different numbers of days, it may be necessary to adjust the month and also year when the day of the month is rolled up. It is the responsibility of the caller of `roll()` to perform that adjustment.

set

public final void set(int field, int value)

Parameters

`field`

The time field to be set.

`value`

The new value.

Description

This method sets the value of the specified time field.

public final void set(int year, int month, int date)

Parameters

`year`

The value for the year field.

`month`

The value for the month field, where 0 represents the first month.

`date`

The value for the day-of-the-month field.

Description

This method sets the values of the year, month, and day-of-the-month fields of this `Calendar`.

`public final void set(int year, int month, int date, int hour, int minute)`

Parameters

`year`

The value for the year field.

`month`

The value for the month field, where 0 represents the first month.

`date`

The value for the day-of-the-month field.

`hour`

The value for the hour field.

`minute`

The value for the minute field.

Description

This method sets the values of the year, month, day-of-the-month, hour, and minute fields of this `Calendar`.

`public final void set(int year, int month, int date, int hour, int minute, int second)`

Parameters

`year`

The value for the year field.

`month`

The value for the month field, where 0 represents the first month.

`date`

The value for the day-of-the-month field.

`hour`

The value for the hour field.

`minute`

The value for the minute field.

`second`

The value for the second field.

Description

This method sets the values of the year, month, day-of-the-month, hour, minute, and second fields of this `Calendar`.

setFirstDayOfWeek

public void setFirstDayofWeek(int value)

Parameters

`value`

The value for the first day of the week.

Description

This method sets the day that is considered the beginning of the week for this `Calendar`. This value should be determined by the `Locale` of this `Calendar`. For example, the first day of the week in the United States is Sunday; in France it's Monday.

setLenient

public void setLenient(boolean lenient)

Parameters

`lenient`

A `boolean` value that specifies the leniency of this `Calendar`.

Description

This method sets the leniency of this `Calendar`. A value of `false` specifies that the `Calendar` throws exceptions when questionable data is passed to it, while a value of `true` indicates that the `Calendar` makes its best guess to interpret questionable data. For example, if the `Calendar` is being lenient, a date such as March 135, 1997 is interpreted as the 135th day after March 1, 1997.

setMinimalDaysInFirstWeek

public void setMinimalDaysInFirstWeek(int value)

Parameters

`value`

The value for the minimum number of days in the first week of the year.

Description

This method sets the minimum number of days in the first week of the year. For example, a value of 7 indicates the first week of the year must be a full week, while a value of 1 indicates the first week of the year can contain a single day. This value should be determined by the `Locale` of this `Calendar`.

setTime

public final void setTime(Date date)

Parameters

`date`

A `Date` object that represents the new time value.

Description

This method sets the point in time that is represented by this `Calendar`.

setTimeZone

public void setTimeZone(TimeZone value)

Parameters

> `value`
>
> > A `TimeZone` object that represents the new time zone.

Description

> This method is used to set the time zone of this `Calendar`.

Protected Instance Methods

complete

protected void complete()

Description

> This method fills out the fields of this `Calendar` as much as possible by calling `computeTime()` and `computeFields()`.

computeFields

protected abstract void computeFields()

Description

> This method calculates the time fields of this `Calendar` from its raw time value.

computeTime

protected abstract void computeTime()

Description

> This method calculates the raw time value of this `Calendar` from its time field values.

getTimeInMillis

protected long getTimeInMillis()

Returns

The raw time value of this `Calendar`.

Description

This method returns the raw time value of this `Calendar`. The value is measured as the number of milliseconds since midnight, January 1, 1970 GMT.

internalGet

protected final int internalGet(int field)

Parameters

`field`

A time field constant.

Returns

The value of the given time field.

Description

This method returns the value of the specified time field without first checking to see if it needs to be computed from the raw time value.

setTimeInMillis

protected void setTimeInMillis(long millis)

Parameters

`millis`

The new raw time value for this `Calendar`.

Description

This method sets the raw time value of this `Calendar`. The value is measured as the number of milliseconds since midnight, January 1, 1970 GMT.

Inherited Methods

Method	Inherited From	Method	Inherited From
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Cloneable`, `Date`, `DateFormat`, `GregorianCalendar`, `Locale`, `Serializable`, `TimeZone`

PREVIOUS HOME NEXT
BitSet BOOK INDEX Date

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Date

Name

Date

Synopsis

Class Name:

`java.util.Date`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

`java.lang.Cloneable`, `java.io.Serializable`

Availability:

JDK 1.0 or later

Description

The `Date` class encapsulates a point in time with millisecond precision. The value of a `Date` is represented internally by a `long` value that contains the number of milliseconds since midnight, January 1, 1970 GMT.

Prior to JDK 1.1, the `Date` class was used for two purposes that are now encapsulated by other classes. First, the `Date` class included methods for calculating calendar values, like months and days of the week. This functionality is now embedded in the `Calendar` class. Second, the `Date` class included methods for generating and parsing a string representation of a date. This functionality is now provided by `java.text.DateFormat`. Thus, as of JDK 1.1, most of the methods of `Date` are deprecated; the class is used only to represent a point in time.

The accurate measurement of time is a subject of considerable complexity and multifarious acronyms. There are two main methods of measuring time, atomic and astronomical. The U.S. Naval Observatory (*http://tycho.usno.navy.mil*) maintains a set of atomic clocks that provide the basis for Coordinated Universal Time (UTC). These clocks adhere to precise definitions of the second based on atomic decay.

Outside of the U.S. Navy, people tend to measure time in terms of Greenwich Mean Time (GMT). In the scientific community, GMT is called UT, which is a system of time predicated on the assumption that each rotation of the earth is exactly 24 * 60 * 60 seconds long. Because the earth's rotation is gradually slowing down, the seconds in UT are a little bit longer than the seconds in UTC. Now and then a "leap second" is added in UTC to keep it close to UT. Because the `Date` class simply measures milliseconds since a point in time, without regard for leap seconds, it is a good representation of UT or GMT.

Class Summary

```
public class java.util.Date extends java.lang.Object
             implements java.lang.Cloneable, java.io.Serializable {
  // Constructors
  public Date();
  public Date(long date);
  public Date(int year, int month, int date);       // Deprecated in 1.1
  public Date(int year, int month, int date,
              int hrs, int min);                    // Deprecated in 1.1
  public Date(int year, int month, int date,
              int hrs, int min, int sec);           // Deprecated in 1.1
  public Date(String s);                            // Deprecated in 1.1
  // Class Methods
  public static long parse(String s);               // Deprecated in 1.1
  public static long UTC(int year, int month,
                         int date, int hrs,
                         int min, int sec);         // Deprecated in 1.1
  // Instance Methods
  public boolean after(Date when);
  public boolean before(Date when);
  public boolean equals(Object obj);
  public int getDate();                             // Deprecated in 1.1
  public int getDay();                              // Deprecated in 1.1
  public int getHours();                            // Deprecated in 1.1
  public int getMinutes();                          // Deprecated in 1.1
  public int getMonth();                            // Deprecated in 1.1
  public int getSeconds();                          // Deprecated in 1.1
  public long getTime();
  public int getTimezoneOffset();                   // Deprecated in 1.1
  public int getYear();                             // Deprecated in 1.1
  public int hashCode();
  public void setDate(int date);                    // Deprecated in 1.1
  public void setHours(int hours);                  // Deprecated in 1.1
  public void setMinutes(int minutes);              // Deprecated in 1.1
  public void setMonth(int month);                  // Deprecated in 1.1
  public void setSeconds(int seconds);              // Deprecated in 1.1
  public void setTime(long time);
  public void setYear(int year);                    // Deprecated in 1.1
  public String toGMTString();                      // Deprecated in 1.1
  public String toLocaleString();                   // Deprecated in 1.1
  public String toString();
}
```

Constructors

Date

public Date()

Description

This constructor creates a `Date` object that is initialized to the current time.

public Date(long date)

Parameters

`date`

A time value, measured as the number of milliseconds since midnight, January 1, 1970 GMT.

Description

This constructor creates a `Date` object that represents the given time.

public Date(int year, int month, int day)

Availability

Deprecated as of JDK 1.1

Parameters

`year`

The year specified as a value that is added to 1900 to get the actual year.

`month`

The month specified in the range 0 to 11.

`day`

The day of the month specified in the range 1 to 31.

Description

This constructor creates a `Date` that represents midnight local time on the specified date.

public Date(int year, int month, int day, int hrs, int min)

Availability

Deprecated as of JDK 1.1

Parameters

`year`

The year specified as a value that is added to 1900 to get the actual year.

`month`

The month specified in the range 0 to 11.

`day`

The day of the month specified in the range 1 to 31.

`hrs`

The hours specified in the range 0 to 23.

`min`

The minutes specified in the range 0 to 59.

Description

This constructor creates a `Date` that represents the given date and time.

`public Date(int year, int month, int day, int hrs, int min, int sec)`

Availability

Deprecated as of JDK 1.1

Parameters

`year`

The year specified as a value that is added to 1900 to get the actual year.

`month`

The month specified in the range 0 to 11.

`day`

The day of the month specified in the range 1 to 31.

`hrs`

The hours specified in the range 0 to 23.

`min`

The minutes specified in the range 0 to 59.

`sec`

The seconds specified in the range 0 to 59.

Description

This constructor creates a `Date` that represents the given date and time.

public Date(String s)

Availability

Deprecated as of JDK 1.1

Parameters

`s`

The string to parse.

Description

This constructor creates a `Date` that represents the date and time specified by the given string. The syntax of the date in the string must satisfy the requirements of the `parse()` method. The following is an example of a string that this constructor can understand:

```
Sat, 8 Feb 1997 13:30:00 GMT
```

Class Methods

parse

public static long parse(String s)

Availability

Deprecated as of JDK 1.1

Parameters

`s`

The string to parse.

Returns

A time value represented as the number of milliseconds since midnight, January 1, 1970 GMT.

Throws

`IllegalArgumentException`

If the string cannot be parsed.

Description

This method returns the raw time value specified by the given string. This method understands a number of different formats. The following are examples of strings that this method can understand:

```
Sat, 8 Feb 1997 13:30:00 GMT
4/6/97
4/6/1997
January 5, 1997
2/4/97 11:03 AM
2/4/97 10:25 PM
2/4/97 17:03 GMT-6
2/4/97 17:03:24
March 16, 97 17:03 EST
March (comment)16, 97 (comment) 17:03 EST
16 march 1996 17:03 pdt
```

```
Sat 16 march 97 17:03 cst
```

The JDK 1.0.2 implementation of `parse()` has a serious bug. It incorrectly interprets date formats that specify the month as a number by making the month one greater than it should be. So `2/4/97` is incorrectly interpreted as March 4, 1997.

For the purposes of this method, UTC and GMT are considered equivalent.

UTC

`public static long UTC(int year, int month, int date, int hrs, int min, int sec)`

Availability

Deprecated as of JDK 1.1

Parameters

`year`

The year specified as a value that is added to 1900 to get the actual year.

`month`

The month specified in the range 0 to 11.

`day`

The day of the month specified in the range 1 to 31.

`hrs`

The hours specified in the range 0 to 23.

`min`

The minutes specified in the range 0 to 59.

`sec`

The seconds specified in the range 0 to 59.

Returns

A time value represented as the number of milliseconds since midnight, January 1, 1970 GMT.

Description

This method returns a raw time value that corresponds to the given parameters. Computations are based on GMT, not the local time zone.

Instance Methods

after

public boolean after(Date when)

Parameters

`when`

The object to compare to this `Date`.

Returns

`true` if this object is after `when`; `false` otherwise.

Description

This method returns `true` if the value of `when` falls before the value of this `Date`.

before

public boolean before(Date when)

Parameters

`when`

The object to compare to this `Date`.

Returns

`true` if this object is before `when`; `false` otherwise.

Description

This method returns `true` if the value of `when` falls after the value of this `Date`.

equals

public boolean equals(Object obj)

Parameters

`obj`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `when` is an instance of `Date` and it contains the same value as the object this method is associated with. In other words, the two `Date` objects are equal only if they both represent the same point in time, to the millisecond.

getDate

public int getDate()

Availability

Deprecated as of JDK 1.1

Returns

The day of the month of this `Date`.

Description

This method returns the day of the month represented by this `Date` object. The value is in the range 1 to 31.

getDay

public int getDay()

Availability

Deprecated as of JDK 1.1

Returns

The day of the week of this `Date`.

Description

This method returns the day of the week represented by this `Date` object. The value is in the range 0 to 6, where 0 means Sunday.

getHours

public int getHours()

Availability

Deprecated as of JDK 1.1

Returns

The hour value of this `Date`.

Description

This method returns the hour represented by this `Date` object. The value is in the range 0 to 23, where 0 means midnight.

getMinutes

public int getMinutes()

Availability

Deprecated as of JDK 1.1

Returns

The minute value of this `Date`.

Description

This method returns the number of minutes after the hour represented by this `Date` object. The value is in the range 0 to 59.

getMonth

public int getMonth()

Availability

Deprecated as of JDK 1.1

Returns

The month of this `Date`.

Description

This method returns the month represented by this `Date` object. The value is in the range 0 to 11, where 0 means January.

getSeconds

public int getSeconds()

Availability

Deprecated as of JDK 1.1

Returns

The second value of this `Date`.

Description

This method returns the number of seconds after the minute represented by this `Date` object. The value is in the range 0 to 59.

getTime

public long getTime()

Returns

The raw time value of this `Date`.

Description

This method returns the date and time of this `Date` as the number of milliseconds since midnight, January 1, 1970 GMT.

getTimezoneOffset

public int getTimezoneOffset()

Availability

Deprecated as of JDK 1.1

Returns

The time zone offset for this `Date`.

Description

This method returns the number of minutes between the local time zone and GMT for this `Date` object.

getYear

public int getYear()

Availability

Deprecated as of JDK 1.1

Returns

The year of this `Date`.

Description

This method returns the year represented by this `Date` object. The value is the number of years since 1990.

hashCode

public int hashCode()

Returns

The hashcode for this `Date`.

Overrides

`Object.hashCode()`

Description

This method returns a hashcode for this object.

setDate

public void setDate(int date)

Availability

Deprecated as of JDK 1.1

Parameters

`date`

The day of the month specified in the range 1 to 31.

Description

This method sets the day of the month of this `Date` object.

setHours

public void setHours(int hours)

Availability

Deprecated as of JDK 1.1

Parameters

`hours`

The hours specified in the range 0 to 23.

Description

This method sets the hour of this `Date` object.

setMinutes

public void setMinutes(int minutes)

Availability

Deprecated as of JDK 1.1

Parameters

`minutes`

The minutes specified in the range 0 to 59.

Description

This method sets the minute value of this `Date` object.

setMonth

public void setMonth(int month)

Availability

Deprecated as of JDK 1.1

Parameters

`month`

The month specified in the range 0 to 11.

Description

This method sets the month of this `Date` object

setSeconds

public void setSeconds(int seconds)

Availability

Deprecated as of JDK 1.1

Parameters

`seconds`

The seconds specified in the range 0 to 59.

Description

This method sets the second value of this `Date` object.

setTime

public void setTime(long time)

Parameters

`time`

A time value specified as the number of milliseconds since midnight, January 1, 1970 GMT.

Description

This method sets the date and time represented by this `Date` to the given raw time value.

setYear

public void setYear(int year)

Availability

Deprecated as of JDK 1.1

Parameters

`year`

The year specified as a value that is added to 1900 to get the actual year.

Description

This method sets the year of this `Date` object.

toGMTString

public String toGMTString()

Availability

Deprecated as of JDK 1.1

Returns

A string that represents this `Date`.

Description

The method returns a string representation of this `Date` object based on Internet GMT conventions. The string is of the form:

```
Sat, 8 Feb 1997 13:30:00 GMT
```

The date is the string is either one or two digits; the rest of the fields always have the width shown. The time zone is always GMT.

toLocaleString

public String toLocaleString()

Availability

Deprecated as of JDK 1.1

Returns

A string that represents this `Date`.

Description

The method returns a string representation of this `Date` based on the conventions of the current locale.

toString

public String toString()

Returns

A string that represents this `Date`.

Overrides

`Object.toString()`

Description

This method returns a string representation of this `Date`. The string is of the form:

```
Sat Feb 8 2:30:00 MST 1997
```

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`Calendar`, `Cloneable`, `DateFormat`, `GregorianCalendar`, `IllegalArgumentException`, `Serializable`, `TimeZone`

PREVIOUS | HOME | NEXT
Calendar | BOOK INDEX | Dictionary

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

PREVIOUS

NEXT

Dictionary

Name

Dictionary

Synopsis

Class Name:

```
java.util.Dictionary
```

Superclass:

```
java.lang.Object
```

Immediate Subclasses:

```
java.util.Hashtable
```

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The `Dictionary` class is an `abstract` class that associates keys with values. Any non-`null` object can be used as a key or as a value. Key/value pairs can be stored in a `Dictionary`, and values can be retrieved or removed using their associated keys.

A subclass of `Dictionary` should use the `equals()` method to decide if two keys are equivalent.

Class Summary

```
public abstract class java.util.Dictionary extends java.lang.Object {
  // Instance Methods
  public abstract Enumeration elements();
  public abstract Object get(Object key);
  public abstract boolean isEmpty();
  public abstract Enumeration keys();
  public abstract Object put(Object key, Object value);
  public abstract Object remove(Object key);
  public abstract int size();
}
```

Instance Methods

elements

public abstract Enumeration elements()

Returns

The values in the dictionary as an `Enumeration`.

Description

This method returns an `Enumeration` that iterates through the values in this `Dictionary`.

get

public abstract Object get(Object key)

Parameters

key

The key of the value to retrieve.

Returns

The value that corresponds to this key.

Description

This method returns the value that is associated with the given key.

isEmpty

public abstract boolean isEmpty()

Returns

`true` if there are no values in the `Dictionary`7thinsp;; `false` otherwise.

Description

This method returns a `boolean` value that indicates whether or not the `Dictionary` is empty.

keys

public abstract Enumeration keys()

Returns

The keys in the dictionary as an `Enumeration`.

Description

This method returns an `Enumeration` that iterates through the keys in this `Dictionary`.

put

public abstract Object put(Object key, Object value)

Parameters

`key`

A key object.

`value`

A value object.

Returns

The previous value associated with the given key or `null` if `key` has not previously been associated with a value.

Throws

`NullPointerException`

If either the key or the value is `null`.

Description

This method associates the given key with the given value in this `Dictionary`.

remove

public abstract Object remove(Object key)

Parameters

`key`

The key of the value to remove.

Returns

The value associated with the given key or `null` if `key` is not associated with a value.

Description

> This method removes a key/value pair from this `Dictionary`. If the given key is not in the `Dictionary`, the method does nothing.

size

public abstract int size()

Returns

> The number of keys in the `Dictionary`.

Description

> This method returns the number of key/value pairs in this `Dictionary`.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Enumeration`, `Hashtable`, `NullPointerException`

GregorianCalendar

Name

GregorianCalendar

Synopsis

Class Name:

`java.util.GregorianCalendar`

Superclass:

`java.util.Calendar`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `GregorianCalendar` class is a subclass of the `abstract Calendar` class. `GregorianCalendar` provides an implementation of the calendar that much of the world uses. `GregorianCalendar` has two eras, `BC` and `AD`.

`GregorianCalendar` provides both Gregorian and Julian dates, depending on the date that is represented by the object. The Gregorian calendar was instituted in October 15, 1582, so any dates before this cut-off time are represented as

Julian dates. Some countries switched from the Julian and the Gregorian calendar after that date, however. The cutoff date can be changed using the `setGregorianChange()` method. When using Julian dates, be aware that this class does not account for the fact that the Julian calendar used March 25 as the beginning of the year. You will have to adjust the year on Julian dates that fall between January 1 and March 24.

You can find a fascinating discussion of the history of Western calendars at *http://barroom.visionsystems.com/serendipity/date/jul_greg.html*.

Class Summary

```
public class java.util.GregorianCalendar extends java.util.Calendar {
  // Constants
  public final static int AD;
  public final static int BC;
  // Constructors
  public GregorianCalendar();
  public GregorianCalendar(TimeZone zone);
  public GregorianCalendar(Locale aLocale);
  public GregorianCalendar(TimeZone zone, Locale aLocale);
  public GregorianCalendar(int year, int month, int date);
  public GregorianCalendar(int year, int month, int date,
                           int hour, int minute);
  public GregorianCalendar(int year, int month, int date,
                           int hour, int minute, int second);
  // Instance Methods
  public void add(int field, int amount);
  public boolean after(Object when);
  public boolean before(Object when);
  public Object clone();
  public boolean equals(Object obj);
  public int getGreatestMinimum(int field);
  public final Date getGregorianChange();
  public int getLeastMaximum(int field);
  public int getMaximum(int field);
  public int getMinimum(int field);
  public synchronized int hashCode();
  public boolean isLeapYear(int year);
  public void roll(int field, boolean up);
  public void setGregorianChange(Date date);
  // Protected Instance Methods
  protected void computeFields();
  protected void computeTime();
}
```

Constants

AD

public final static int AD

Description

A `constant` value that represents the AD era, which stands for *anno Domini*, Latin for "the year of the Lord". People who do not want to measure years with a Christian connotation call this era CE the Common Era.

BC

public final static int BC

Description

A `constant` value that represents the BC era, which stands for *before Christ*, before the birth of Christ. People who do not want to measure years with a Christian connotation call this era BCE, which stands for Before the Common Era.

Constructors

GregorianCalendar

public GregorianCalendar()

Description

This constructor creates `a GregorianCalendar` that represents the current time using the system's default time zone and locale. The default time zone is that returned by `TimeZone.getDefault()`. The default locale is that returned by `Locale.getDefault()`.

public GregorianCalendar(TimeZone zone)

Parameters

`zone`

The `TimeZone` to use.

Description

This constructor creates a `GregorianCalendar` that represents the current time using the supplied time zone and the default locale. The default locale is that returned by `Locale.getDefault()`.

public GregorianCalendar(Locale aLocale)

Parameters

`aLocale`

The `Locale` to use.

Description

This constructor creates a `GregorianCalendar` that represents the current time using the supplied locale and the default time zone. The default time zone is that returned by `TimeZone.getDefault()`.

public GregorianCalendar(TimeZone zone, Locale aLocale)

Parameters

`zone`

The `TimeZone` to use.

`aLocale`

The `Locale` to use.

Description

This constructor creates a `GregorianCalendar` that represents the current time using the supplied time zone and locale.

public GregorianCalendar(int year, int month, int date)

Parameters

`year`

The value for the year field.

`month`

The value for the month field, where 0 represents the first month.

`date`

The value for the day-of-the-month field.

Description

This constructor creates a `GregorianCalendar` that represents the given date in the default time zone and locale. The default time zone is that returned by `TimeZone.getDefault()`. The default locale is that returned by `Locale.getDefault()`.

`public GregorianCalendar(int year, int month, int date, int hour, int minute)`

Parameters

`year`

The value for the year field.

`month`

The value for the month field, where 0 represents the first month.

`date`

The value for the day-of-the-month field.

`hour`

The value for the hour field.

`minute`

The value for the minute field.

Description

This constructor creates a `GregorianCalendar` that represents the given date and time in the default time zone and locale. The default time zone is that returned by `TimeZone.getDefault()`. The default locale is that returned by `Locale.getDefault()`.

`public GregorianCalendar(int year, int month, int date, int hour, int minute, int second)`

Parameters

`year`

The value for the year field.

`month`

The value for the month field, where 0 represents the first month.

`date`

The value for the day-of-the-month field.

`hour`

The value for the hour field.

`minute`

The value for the minute field.

`second`

The value for the second field.

Description

This constructor creates a `GregorianCalendar` that represents the given data and time in the default time zone and locale. The default time zone is that returned by `TimeZone.getDefault()`. The default locale is that returned by `Locale.getDefault()`.

Instance Methods

add

public void add(int field, int amount)

Parameters

`field`

The time field to be modified.

`amount`

The amount to add to the specified field value. This value can be negative.

Throws

`IllegalArgumentException`

If `field` is not a valid time field.

Overrides

`Calendar.add()`

Description

This method adds the given amount to the specified time field. For example, you can compute a date 90 days beyond the current date of this `GregorianCalendar` by calling `add(Calendar.DATE, 90)`.

after

public boolean after(Object when)

Parameters

`when`

The object to compare to this `GregorianCalendar`.

Returns

`true` if this object is after `when`; `false` otherwise.

Overrides

`Calendar.after()`

Description

This method returns `true` if `when` is a `GregorianCalendar` whose value falls before the value of this `GregorianCalendar`.

before

public boolean before(Object when)

Parameters

`when`

The object to compare to this `GregorianCalendar`.

Returns

`true` if this object is before `when`; `false` otherwise.

Overrides

`Calendar.before()`

Description

This method returns `true` if `when` is a `GregorianCalendar` whose value falls after the value of this `GregorianCalendar`.

clone

public Object clone()

Returns

A copy of this `GregorianCalendar`.

Overrides

`Calendar.clone()`

Description

This method creates a copy of this `GregorianCalendar` and returns it. In other words, the returned `GregorianCalendar` has the same time field values and raw time value as this `GregorianCalendar`.

equals

public boolean equals(Object when)

Parameters

`when`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Calendar.equals()`

Description

This method returns `true` if `when` is an instance of `GregorianCalendar`, and it contains the same value as the object this method is associated with.

getGreatestMinimum

public int getGreatestMinimum(int field)

Parameters

`field`

A time field constant.

Returns

The highest minimum value for the given time field.

Overrides

`Calendar.getGreatestMinimum()`

Description

This method returns the highest minimum value for the given time field, if the field has a range of minimum values. If the field has only one minimum value, this method is equivalent to `getMinimum()`. All of the fields in `GregorianCalendar` have only one minimum value.

getGregorianChange

public final Date getGregorianChange()

Returns

The date this `GregorianCalendar` uses as the change date between the Julian and Gregorian calendars.

Description

By default, `GregorianCalendar` considers midnight local time, October 15, 1582, to be the date when the Gregorian calendar was adopted. This value can be changed using `setGregorianChange()`.

getLeastMaximum

public int getLeastMaximum(int field)

Parameters

`field`

A time field constant.

Returns

The lowest maximum value for the given time field.

Overrides

`Calendar.getLeastMaximum()`

Description

This method returns the lowest maximum value for the given time field, if the field has a range of maximum values. If the field has only one maximum value, this method is equivalent to `getMaximum()`. For example, for a `GregorianCalendar`, the lowest maximum value of `DATE_OF_MONTH` is 28.

getMaximum

public int getMaximum(int field)

Parameters

`field`

A time field constant.

Returns

The maximum value for the given time field.

Overrides

`Calendar.getMaximum()`

Description

This method returns the maximum value for the given time field. For example, for a `GregorianCalendar`, the maximum value of `DATE_OF_MONTH` is 31.

getMinimum

public int getMinimum(int field)

Parameters

`field`

A time field constant.

Returns

The minimum value for the given time field.

Overrides

`Calendar.getMinimum()`

Description

This method returns the minimum value for the given time field. For example, for a `GregorianCalendar`, the minimum value of `DATE_OF_MONTH` is 1.

hashCode

public synchronized int hashCode()

Returns

A hashcode for this `GregorianCalendar`.

Overrides

`Object.hashCode()`

Description

This method returns a hashcode for this object.

isLeapYear

public boolean isLeapYear(int year)

Parameters

`year`

The year to test.

Returns

`true` if the given year is a leap year; `false` otherwise.

Description

This method returns a `boolean` value that indicates whether or not the specified year is a leap year. Leap years are those years that are divisible by 4, except those that are divisible by 100, unless they are divisible by 400. For example, 1900 is not a leap year because it is divisible by 100 but not by 400. The year 2000 is a leap year.

roll

public void roll(int field, boolean up)

Parameters

`field`

The time field to be adjusted.

`up`

A `boolean` value that indicates if the given field should be incremented.

Throws

`IllegalArgumentException`

If `field` is not a valid time field.

Overrides

`Calendar.roll()`

Description

This method adds or subtracts one time unit from the given time field. For example, to increase the current date by one day, you can call `roll(GregorianCalendar.DATE, true)`.

The method maintains the field being rolled within its valid range. For example, in a calendar system that uses hours and minutes to measure time, rolling the minutes up from 59 sets that field to 0. By the same token, rolling that field down from 0 sets it to 59.

The `roll()` method does not adjust the value of any other field than the one specified by its `field` argument. In particular, for calendar systems that have months with different numbers of days, it may be necessary to adjust the month and also year when the day of the month is rolled up. For example, calling `roll(GregorianCalendar.DAY_OF_MONTH, true)` on a `GregorianCalendar` that represents December 31, 1996 changes the date to December 1, 1996. In addition, calling `roll()` may make the fields inconsistent. For example, calling `roll(GregorianCalendar.MONTH, true)` on a `GregorianCalendar` that represents January 31, 1997 changes the date to February 31, 1997. It is the responsibility of the caller of `roll()` to adjust the other fields.

setGregorianChange

public void setGregorianChange(Date date)

Parameters

`date`

A `Date` object that represents the new time value.

Description

This method sets the date that this `GregorianCalendar` uses as the change date between the Julian and Gregorian calendars. The default is midnight local time, October 15, 1582. This is the date that Pope Gregory instituted the calendar in many Catholic countries in Europe. Most Catholic countries followed within a few years. Protestant England and America did not adopt the new calendar until September 14, 1752.

Protected Instance Methods

computeFields

protected void computeFields()

`Overrides`

`Calendar.computeFields()`

Description

This method calculates the time fields of this `GregorianCalendar` from its raw time value.

computeTime

protected void computeTime()

`Overrides`

`Calendar.computeTime()`

Description

This method calculates the raw time value of this `GregorianCalendar` from its time field values.

Inherited Variables

Variable	Inherited From	Variable	Inherited From
`areFieldsSet`	`Calendar`	`fields`	`Calendar`
`isSet`	`Calendar`	`isTimeSet`	`Calendar`
`time`	`Calendar`		

Inherited Methods

Method	Inherited From	Method	Inherited From

clear()	Calendar	clear(int)	Calendar
complete()	Calendar	finalize()	Object
get(int)	Calendar	getClass()	Object
getFirstDayOfWeek()	Calendar	getMinimumDaysInFirstWeek()	Calendar
getTime()	Calendar	getTimeInMillis()	Calendar
getTimeZone()	Calendar	internalGet(int)	Calendar
isLenient()	Calendar	isSet(int)	Calendar
notify()	Object	notifyAll()	Object
set(int, int)	Calendar	set(int, int, int)	Calendar
set(int, int, int, int, int)	Calendar	set(int, int, int, int, int, int)	Calendar
setFirstDayOfWeek(int)	Calendar	setLenient(boolean)	Calendar
setMinimalDaysInFirstWeek(int)	Calendar	setTime(Date)	Calendar
setTimeInMillis(long)	Calendar	setTimeZone(TimeZone)	Calendar
toString()	Object	wait()	Object
wait(long)	Object	wait(long, int)	Object

See Also

Calendar, Cloneable, Date, IllegalArgumentException, Locale, Serializable, TimeZone

PREVIOUS HOME NEXT
EventObject BOOK INDEX Hashtable

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

PREVIOUS

NEXT

Hashtable

Name

Hashtable

Synopsis

Class Name:

`java.util.Hashtable`

Superclass:

`java.util.Dictionary`

Immediate Subclasses:

`java.util.Properties`

Interfaces Implemented:

`java.lang.Cloneable`, `java.io.Serializable`

Availability:

JDK 1.0 or later

Description

The Hashtable class is a concrete subclass of Dictionary that builds a table of key/value pairs. Any non-null object can be used as a key or as a value. The objects used as keys must implement the equals() and hashCode() methods in a way that computes comparisons and hashcodes from the contents of an object. Once the table is built, a value can be efficiently retrieved by supplying its associated key.

Hashtable is an excellent example of how a well-written class can hide an arcane algorithm. The casual user simply instantiates a Hashtable and uses put() and get() to add and retrieve key and value pairs. However, when performance is an issue, you need to be aware of the considerations discussed in the following paragraphs.

Internally, a Hashtable keeps an array of key/value pairs. When a new key/value pair is added to a Hashtable, it is added to the array at an index that is calculated from the hashcode of the key. If a key/value pair already exists at this index, the new pair is linked to the existing key and value. Thus, a Hashtable has an overall structure of an array of linked lists.

For a given key, the retrieval of the matching value from a Hashtable is quite fast. The Hashtable computes the hashcode of the key and uses it as an index into the array. Then it only needs to search the linked list of key/value pairs at that index to find a match for the given key. If the array is short, but the Hashtable contains many key/value pairs, however, the linked lists will be lengthy, which adversely affects performance.

A Hashtable has a capacity, which is the length of its array, and a load factor, which determines when rehashing is performed. The load factor is a number between 0 and 1. If the number of key/value pairs added to the Hashtable exceeds the capacity multiplied by the load factor, the capacity of the Hashtable is increased and the key/value pairs are rehashed into the new array. Obviously, this is an undesirable performance hit, so if you know approximately how many items you will add to a Hashtable, you should create one with an appropriate initial capacity.

Class Summary

```
public class java.util.Hashtable extends java.util.Dictionary
             implements java.lang.Cloneable, java.io.Serializable {
  // Constructors
  public Hashtable();
  public Hashtable(int initialCapacity);
  public Hashtable(int initialCapacity, float loadFactor);
  // Instance Methods
  public synchronized void clear();
  public synchronized Object clone();
  public synchronized boolean contains(Object value);
  public synchronized boolean containsKey(Object key);
  public synchronized Enumeration elements();
  public synchronized Object get(Object key);
  public boolean isEmpty();
  public synchronized Enumeration keys();
  public synchronized Object put(Object key, Object value);
  public synchronized Object remove(Object key);
  public int size();
  public synchronized String toString();
  // Protected Instance Methods
  protected void rehash();
}
```

Constructors

Hashtable

public Hashtable()

Description

> This constructor creates a `Hashtable` with a default capacity of 101 and a default load factor of .75.

public Hashtable(int initialCapacity)

Parameters

> `initialCapacity`
>
> > The initial capacity.

Throws

> `IllegalArgumentException`
>
> > If `initialCapacity` is less than or equal to zero.

Description

This constructor creates a `Hashtable` with the given capacity and a default load factor of .75.

public Hashtable(int initialCapacity, float loadFactor)

Parameters

`initialCapacity`

The initial capacity.

`loadFactor`

The load factor.

Throws

`IllegalArgumentException`

If `initialCapacity` or `loadFactor` is less than or equal to zero.

Description

This constructor creates a `Hashtable` with the given capacity and load factor.

Instance Methods

clear

public synchronized void clear()

Description

This method removes all of the key/value pairs from this `Hashtable`.

clone

public synchronized Object clone()

Returns

A copy of this `Hashtable`.

Overrides

`Object.clone()`

Description

This method returns a shallow copy of this `Hashtable`. This means that the internal array of the `Hashtable` is copied, but the keys and values themselves are not copied.

contains

public synchronized boolean contains(Object value)

Parameters

`value`

The value to find.

Returns

`true` if this `Hashtable` contains the given value; `false` otherwise.

Throws

`NullPointerException`

If the given value is `null`.

Description

This method returns `true` if the given value is contained in this `Hashtable` object. The entire table is searched, which can be a time-consuming operation.

containsKey

public synchronized boolean containsKey(Object key)

Parameters

`key`

The key to find.

Returns

`true` if this `Hashtable` contains the given value; `false` otherwise.

Description

This method returns `true` if the given key is contained in this `Hashtable` object. Because the key is hashed to perform the search, this method runs quite fast, especially in comparison to `contains()`.

elements

public synchronized Enumeration elements()

Returns

The values in this `Hashtable` as an `Enumeration`.

Overrides

`Dictionary.elements()`

Description

This method returns an `Enumeration` that iterates through the values in this `Hashtable`.

get

public synchronized Object get(Object key)

Parameters

`key`

The key of the value to retrieve.

Returns

The value that corresponds to this key or `null` if the key is not associated with any value.

Overrides

`Dictionary.get()`

Description

This method returns the value that is associated with the given key.

isEmpty

public boolean isEmpty()

Returns

`true` if there are no values in the `Hashtable`; `false` otherwise.

Overrides

`Dictionary.isEmpty()`

Description

This method returns a `boolean` value that indicates whether or not the `Hashtable` is empty.

keys

public synchronized Enumeration keys()

Returns

The keys in the `Hashtable` as an `Enumeration`.

Overrides

`Dictionary.keys()`

Description

This method returns an `Enumeration` that iterates through the keys in this `Hashtable`.

put

public synchronized Object put(Object key, Object value)

Parameters

`key`

A key object.

`value`

A value object.

Returns

The previous value associated with the given key or `null` if `key` has not previously been associated with a value.

Throws

`NullPointerException`

If either the key or the value is `null`.

Overrides

`Dictionary.put()`

Description

This method associates the given key with the given value in this `Hashtable`.

remove

public synchronized Object remove(Object key)

Parameters

`key`

A key of the value to remove.

Returns

The value associated with the given key, or `null` if `key` is not associated with a value.

Overrides

`Dictionary.remove()`

Description

This method removes a key/value pair from this `Hashtable`. If the given key is not in the `Hashtable`, the method does nothing.

size

public int size()

Returns

The number of key in the `Hashtable`.

Overrides

`Dictionary.size()`

Description

This method returns the number of key/value pairs in the `Hashtable`.

toString

public String toString()

Returns

A string that represents this `Hashtable`.

Overrides

`Object.toString()`

Description

This method returns a string representation of this `Hashtable`. The string includes every key/value pair that is contained in the `Hashtable`, so the string returned by `toString()` can be quite long.

Protected Instance Methods

rehash

protected void rehash()

Description

This method increases the capacity of this `Hashtable`. A larger internal array is created and all existing key/value pairs are rehashed into the new array.

Inherited Methods

Method	Inherited From	Method	Inherited From
`equals(Object)`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Cloneable`, `Dictionary`, `Enumeration`, `IllegalArgumentException`, `NullPointerException`, `Properties`, `Serializable`

PREVIOUS | HOME | NEXT
GregorianCalendar | BOOK INDEX | ListResourceBundle

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Locale

Name

Locale

Synopsis

Class Name:

`java.util.Locale`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

`java.lang.Cloneable`, `java.io.Serializable`

Availability:

New as of JDK 1.1

Description

The `Locale` class is used for internationalization. Instances of `Locale` specify language and formatting customs by identifying a language and a country. A `Locale` object may also specify a platform-specific variant. Other classes throughout the JDK use `Locale` objects to determine how to represent themselves to the user. The tasks performed by these classes are called locale-sensitive tasks; the tasks should be done in a way that conforms with the conventions of a particular country and language.

There are a number of classes provided with Java that have `static` methods that create instances of locale-specific subclasses. For example, the `NumberFormat` class contains `static` methods named `getInstance()` that create and return locale-specific instances of subclasses of `NumberFormat`. A particular `NumberFormat` instance knows how to format numbers, currency values, and percentages appropriately for a particular locale. Note that it is the responsibiity of a class like `NumberFormat` to implement the logic needed to translate locale-identifying information into actual subclass instances.

Classes like `NumberFormat` that can create locale-specific instances are expected to follow certain conventions:

- Methods like `getInstance()` in `NumberFormat` are expected to have two variants: one that takes a `Locale` argument and one that does not. The variant that does not take a locale argument is expected to use the default locale, which is normally determined by calling `Locale.getDefault()`.

- Classes that can create a variety of locale-specific instances are expected to implement a method that has the following signature:

  ```
  public static Locale[] getAvailableLocales()
  ```

  This requirement is not specified through an interface declaration because interfaces cannot declare `static` methods. The purpose of this method is to facilitate presenting the user with a list or menu of locale choices. The `getAvailableLocales()` method should return an array of `Locale` objects that identifies all of the locales for which the class can create locale-specific instances.

  Two additional methods are recommended for helping to display the locale choices:

  ```
  public static final String getDisplayName(Locale objectLocale)
  public static String getDisplayName(Locale objectLocale,
                                      Locale displayLocale)
  ```

  The first form of `getDisplayName()` should return a description of `objectLocale` that is suitable for display in the default locale. The second form should return a description of `objectLocale` that is suitable for display in the locale specified by `displayLocale`. Implementations of these methods generally call the `getDisplayName()` method of the `Locale` object.

The language, country and variant information that are encapsulated by a `Locale` object are specified to a constructor as strings. The language for a `Locale` should be specified as one of the two-letter lowercase language codes defined by ISO-639. Look for a complete list at *http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt*.

The country for a `Locale` object should be specified as either `""` to indicate that no country is specified, or as one of the two-letter uppercase country codes defined by ISO-3166. Check the site, *http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html*, for a complete list

Variant codes are platform-specific.

Although the `Locale` is constructed from these three types of codes, human-readable names can be obtained by calling `getDisplayLanguage()`, `getDisplayCountry()`, and `getDisplayVariant()`.

The `Locale` class defines a number of constant `Locale` objects that represent some of the major languages and countries of the world.

Class Summary

```
public abstract class java.util.Locale extends java.lang.Object
                      implements java.lang.Cloneable, java.io.Serializable {
  // Constants
  public final static Locale CANADA;
  public final static Locale CANADA_FRENCH;
  public final static Locale CHINA;
  public final static Locale CHINESE;
  public final static Locale ENGLISH;
  public final static Locale FRANCE;
  public final static Locale FRENCH;
  public final static Locale GERMAN;
  public final static Locale GERMANY;
  public final static Locale ITALIAN;
  public final static Locale ITALY;
  public final static Locale JAPAN;
  public final static Locale JAPANESE;
  public final static Locale KOREA;
  public final static Locale KOREAN;
  public final static Locale PRC;
  public final static Locale SIMPLIFIED_CHINESE;
  public final static Locale TAIWAN;
  public final static Locale TRADITIONAL_CHINESE;
  public final static Locale UK;
  public final static Locale US;
  // Constructors
  public Locale(String language, String country);
  public Locale(String language, String country, String variant);
  // Class Methods
  public static synchronized Locale getDefault();
  public static synchronized void setDefault(Locale newLocale);
  // Instance Methods
  public Object clone();
  public boolean equals(Object obj);
  public String getCountry();
  public final String getDisplayCountry();
  public String getDisplayCountry(Locale inLocale);
  public final String getDisplayLanguage();
  public String getDisplayLanguage(Locale inLocale);
  public final String getDisplayName();
  public String getDisplayName(Locale inLocale);
  public final String getDisplayVariant();
  public String getDisplayVariant(Locale inLocale);
  public String getISO3Country();
  public String getISO3Language();
  public String getLanguage();
  public String getVariant();
  public synchronized int hashCode();
  public final String toString();
}
```

Constants

CANADA

public final static Locale CANADA

Description

> A locale that represents English-speaking Canada.

CANADA_FRENCH

public final static Locale CANADA_FRENCH

Description

> A locale that represents French-speaking Canada.

CHINA

public final static Locale CHINA

Description

A locale that represents China.

CHINESE

public final static Locale CHINESE

Description

A locale that represents the Chinese language.

ENGLISH

public final static Locale ENGLISH

Description

A locale that represents the English language.

FRANCE

public final static Locale FRANCE

Description

A locale that represents France.

FRENCH

public final static Locale FRENCH

Description

A locale that represents the French language.

GERMAN

public final static Locale GERMAN

Description

A locale that represents the German language.

GERMANY

public final static Locale GERMANY

Description

A locale that represents Germany.

ITALIAN

public final static Locale ITALIAN

Description

A locale that represents the Italian language.

ITALY

public final static Locale ITALY

Description

A locale that represents Italy.

JAPAN

public final static Locale JAPAN

Description

A locale that represents Japan.

JAPANESE

public final static Locale JAPANESE

Description

A locale that represents the Japanese language.

KOREA

public final static Locale KOREA

Description

A locale that represents Korea.

KOREAN

public final static Locale KOREAN

Description

A locale that represents the Korean language.

PRC

public final static Locale PRC

Description

A locale that represents the People's Republic of China. It is equivalent to `CHINA`.

SIMPLIFIED_CHINESE

public final static Locale SIMPLIFIED_CHINESE

Description

A locale that represents the Chinese language as used in mainland China.

TAIWAN

public final static Locale TAIWAN

Description

A locale that represents Taiwan.

TRADITIONAL_CHINESE

public final static Locale TRADITIONAL_CHINESE

Description

A locale that represents the Chinese language as used in Taiwan.

UK

public final static Locale UK

Description

A locale that represents the United Kingdom.

US

public final static Locale US

Description

A locale that represents the United States.

Constructors

Locale

public Locale(String language, String country)

Parameters

`language`

A two-letter ISO-639 language code.

`country`

A two-letter ISO-3166 country code or `""` to omit the country specification.

Description

This constructor creates a `Locale` that represents the given language and country.

public Locale(String language, String country, String variant)

Parameters

`language`

A two-letter ISO-639 language code.

`country`

A two-letter ISO-3166 country code or `""` to omit the country specification.

`variant`

A vendor-specific variant code.

Description

This constructor creates a `Locale` that represents the given language, country, and variant.

Class Methods

getDefault

public static synchronized Locale getDefault()

Returns

The default `Locale`.

Description

This method returns the current default `Locale`. An application or applet uses this method to find out how to present locale-sensitive information, such as textual strings and numbers. The method is generally called during application initialization to get the default `Locale`. Once the locale is set, it almost never changes. If you do change the locale, you should probably reload the GUI for your application, so that any locale-sensitive information in the interface is changed.

The initial default `Locale` is set by the host system.

setDefault

public static synchronized void setDefault(Locale newLocale)

Parameters

`newLocale`

The new default locale.

Description

This method changes the current default locale to `newLocale`. Note that calling `setDefault()` does not change the default locale of the host system.

Instance Methods

clone

public Object clone()

Returns

A copy of this `Locale`.

Overrides

`Object.clone()`

Description

This method creates a copy of this `Locale` and returns it.

equals

public boolean equals(Object obj)

Parameters

`obj`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `obj` is an instance of `Locale`, and it contains the same value as the object this method is associated with.

getCountry

public String getCountry()

Returns

`The country of this Locale`.

Description

This method returns a `String` that represents the country of this `Locale`. This `String` is the same `String` that was passed to the constructor of this `Locale` object. The `String` is normally a two-letter ISO-3166 country code.

getDisplayCountry

public final String getDisplayCountry()

Returns

`The country of this Locale`.

Description

This method returns the country of this `Locale` as a country name in a form appropriate for this `Locale`. If the country name cannot be found, this method returns the same value as `getCountry()`.

public String getDisplayCountry(Locale inLocale)

Parameters

`inLocale`

The locale to use when finding the country name.

Returns

The country of this `Locale`, localized to the given locale.

Description

This method returns the country of this `Locale` as a country name in a form appropriate for `inLocale`. For example, `Locale.ITALY.getDisplayCountry(Locale.GERMAN)` returns the German name for Italy, Italien.

getDisplayLanguage

public final String getDisplayLanguage()

Returns

The language of this `Locale`.

Description

This method returns the language of this `Locale` as a language name in a form appropriate for this `Locale`. If the language name cannot be found, this method returns the same value as `getLanguage()`.

public String getDisplayLanguage(Locale inLocale)

Parameters

`inLocale`

The locale to use when finding the language name.

Returns

The language of this `Locale`, localized to the given locale.

Description

This method returns the language of this `Locale` as a language name in a form appropriate for `inLocale`. For example, `Locale.ITALY.getDisplayLanguage(Locale.GERMAN)` returns the German name for the Italian language, Italienisch.

getDisplayName

public final String getDisplayName()

Returns

A string that represents this `Locale`.

Description

This method constructs a string that represents this `Locale` by calling `getDisplayLanguage()`, `getDisplayCountry()`, and `getDisplayVariant()`. In other words, the method returns a string that contains the country name, language name, and variant in a form appropriate for this `Locale`. If any of the names cannot be found, the `String` that was passed to the constructor of this `Locale` object is used instead. These strings are normally two-letter ISO codes.

public String getDisplayName(Locale inLocale)

Parameters

`inLocale`

The locale to use when constructing the string representation.

Returns

A string that represents this `Locale`.

Description

This method constructs a string that represents this `Locale` by calling `getDisplayLanguage(inLocale)`, `getDisplayCountry(inLocale)`, and `getDisplayVariant(inLocale)`. In other words, the method returns a string that contains the country name, language name, and variant in a form appropriate for `inLocale`. If any of the names cannot be found, the `String` that was passed to the constructor of this `Locale` object is used instead. These strings are normally two-letter ISO codes.

getDisplayVariant

public final String getDisplayVariant()

Returns

`The variant of this Locale`.

Description

This method returns the variant of this `Locale` as a human-readable string in a form appropriate for this

Locale. If the variant name cannot be found, this method returns the same value as getVariant().

public String getDisplayVariant(Locale inLocale)

Parameters

inLocale

The locale to use when finding the variant name.

Returns

The variant of this Locale, localized to the given locale.

Description

This method returns the variant of this Locale as a human-readable string in a form appropriate for inLocale.

getISO3Country

public String getISO3Country() throws MissingResourceException

Returns

The ISO three-letter country code of this Locale.

Throws

MissingResourceException

If the requested code cannot be found.

Description

This method returns the country of this Locale as a three-letter ISO country code. The country code is obtained from a ResourceBundle for this Locale.

getISO3Language

public String getISO3Language() throws MissingResourceException

Returns

The ISO three-letter language code of this `Locale`.

Throws

`MissingResourceException`

If the requested code cannot be found.

Description

This method returns the language of this `Locale` as a three-letter ISO language code. The language code is obtained from a `ResourceBundle` for this `Locale`.

getLanguage

public String getLanguage()

Returns

The language of this `Locale`.

Description

This method returns a `String` that represents the language of this `Locale`. This `String` is the same `String` that was passed to the constructor of this `Locale` object. The `String` is normally a two-letter ISO-639 language code.

getVariant

public String getVariant()

Returns

The variant of this `Locale`.

Description

This method returns the variant code of this `Locale`. If no variant code is specified for this `Locale`, an empty string is returned.

hashCode

public synchronized int hashCode()

Returns

A hashcode for this `Locale`.

Overrides

`Object.hashCode()`

Description

This method returns a hashcode for this object.

toString

public final String toString()

Returns

A string representation of this `Locale`.

Overrides

`Object.toString()`

Description

This method returns a string representation of this `Locale`, constructed from the language code, country code, and variant code. The various codes are separated by underscore characters. If a code is missing, it is omitted.

Inherited Methods

Method	Inherited From	Method	Inherited From
`finalize()`	`Object`	`getClass()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Cloneable`, `DateFormat`, `NumberFormat`, `ResourceBundle`, `Serializable`

PREVIOUS | HOME | NEXT
ListResourceBundle | BOOK INDEX | MissingResourceException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Observable

Name

Observable

Synopsis

Class Name:

```
java.util.Observable
```

Superclass:

```
java.lang.Object
```

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

Subclasses of the `Observable` class are used to implement the model portion of the model-view paradigm. The idea is that an `Observable` object, the model, represents some data that is being manipulated through a user interface, while `Observer` objects provide the user with a view of the data. When the `Observable` object is modified, it tells the `Observer` objects that the model has been modified by calling `notifyObservers()`. An `Observer` object registers with an `Observable` object to receive notifications when the `Observable` is modified. The `Observer` object is then notified of changes via the `update()` method.

Class Summary

```
public class java.util.Observable extends java.lang.Object {
  // Constructors
  public Observable();
  // Instance Methods
  public synchronized void addObserver(Observer o);
  public synchronized int countObservers();
  public synchronized void deleteObserver(Observer o);
  public synchronized void deleteObservers();
  public synchronized boolean hasChanged();
  public void notifyObservers();
  public void notifyObservers(Object arg);
  // Protected Instance Methods
  protected synchronized void clearChanged();
  protected synchronized void setChanged();
}
```

Constructors

Observable

public Observable()

Description

> This constructor `creates an Observable` object with no registered `Observer` objects.

InstanceMethods

addObserver

public synchronized void addObserver(Observer o)

Parameters

o

The `Observer` to be added.

Description

This method registers the given `Observer` with this `Observable` object. The given `Observer` is then notified when `notifyObservers()` is called.

countObservers

public synchronized int countObservers()

Returns

The number of registered `Observer` objects for this `Observable` object.

Description

This method returns the number of `Observer` objects that are registered with this `Observable` object.

deleteObserver

public synchronized void deleteObserver(Observer o)

Parameters

o

The `Observer` to be removed.

Description

This method unregisters the given `Observer` with this `Observable` object. The given `Observer` is no longer notified when `notifyObservers()` is called.

deleteObservers

public synchronized void deleteObservers()

Description

This method unregisters all of the `Observer` objects of this `Observable` object. Thus, no objects are notified if `notifyObservers()` is called.

hasChanged

public synchronized boolean hasChanged()

Returns

`true` if this object has been flagged as changed; `false` otherwise.

Description

This method returns the value of an internal "dirty" flag. The flag can be modified using the `protected` methods `setChanged()` and `clearChanged()`.

notifyObservers

public void notifyObservers()

Description

This method calls the `update()` method of all registered `Observer` objects. The value passed as the second argument to each of the `update()` method calls is `null`.

public void notifyObservers(Object arg)

Parameters

`arg`

A "hint" object that describes a change.

Description

This method calls the `update()` method of all registered `Observer` objects. The value passed as the second argument to each of the `update()` method calls is the given object `arg`.

This "hint" object can be used to efficiently update the views of a model. For example, an `Observable` object could represent satellite image data. A set of `Observer` objects would provide different graphical views of the data. If the model data changes, the `arg` object describes the part of the data that changed, and the `Observer` views could use this "hint" to update only parts of their displays.

Protected Instance Methods

clearChanged

protected synchronized void clearChanged()

Description

This method sets an internal "dirty" flag to `false`. After this method is called, this object's `hasChanged()` method returns `false`.

setChanged

protected synchronized void setChanged()

Description

This method sets an internal "dirty" flag to `true`. After this method is called, this object's `hasChanged()` method returns `true`.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`

`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Observer`

PREVIOUS
NoSuchElementException

HOME
BOOK INDEX

NEXT
Observer

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

SimpleTimeZone

Name

SimpleTimeZone

Synopsis

Class Name:

`java.util.SimpleTimeZone`

Superclass:

`java.util.TimeZone`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `SimpleTimeZone` class is a concrete subclass of the `abstract` `TimeZone` class. `SimpleTimeZone` represents a time zone offset for use with `GregorianCalendar`. `SimpleTimeZone` does not take into account the historical vicissitudes of daylight savings time, however. Instead, it assumes that the shifts to and from daylight savings time always occur at the same time of the year.

Normally, `SimpleTimeZone` objects are not created directly, but instead obtained by calling

TimeZone.getDefault(). This method creates a subclass of TimeZone that is appropriate for the current locale. You can also call TimeZone.getTimeZone() to obtain a TimeZone object for a specific time zone.

Class Summary

```
public class java.util.SimpleTimeZone extends java.util.TimeZone {
  // Constructors
  public SimpleTimeZone(int rawOffset, String ID);
  public SimpleTimeZone(int rawOffset, String ID, int startMonth,
                        int startDayOfWeekInMonth, int startDayOfWeek,
                        int startTime, int endMonth, int endDayOfWeekInMonth,
                        int endDayOfWeek, int endTime);
  // Instance Methods
  public Object clone();
  public boolean equals(Object obj);
  public int getOffset(int era, int year, int month, int day,
                       int dayOfWeek, int millis);
  public int getRawOffset();
  public synchronized int hashCode();
  public boolean inDaylightTime(Date date);
  public void setEndRule(int month, int dayOfWeekInMonth,
                         int dayOfWeek, int time);
  public void setRawOffset(int offsetMillis);
  public void setStartRule(int month, int dayOfWeekInMonth,
                           int dayOfWeek, int time);
  public void setStartYear(int year);
  public boolean useDaylightTime();
}
```

Constructors

SimpleTimeZone

public SimpleTimeZone(int rawOffset, String ID)

Parameters

rawOffset

The raw offset of this time zone from GMT, in milliseconds.

ID

The ID of this time zone.

Description

This constructor creates a SimpleTimeZone that uses the given offset from GMT and has the specified ID. This constructor creates a `SimpleTimeZone` that does not use daylight savings time.

`public SimpleTimeZone(int rawOffset, String ID, int startMonth, int startDayOfWeekInMonth, int startDayOfWeek, int startTime, int endMonth, int endDayOfWeekInMonth, int endDayOfWeek, int endTime)`

Parameters

`rawOffset`

The raw offset of this time zone from GMT, in milliseconds.

`ID`

The ID of this time zone.

`startMonth`

The month when daylight savings time begins.

`startDayOfWeekInMonth`

The week in the month when daylight savings time begins.

`startDayOfWeek`

The day of the week when daylight savings time begins.

`startTime`

The time of day when daylight savings time begins, in milliseconds.

`endMonth`

The month when daylight savings time ends.

`endDayOfWeekInMonth`

The week in the month when daylight savings time ends.

`endDayOfWeek`

The day of the week when daylight savings time ends.

`endTime`

The time of day when daylight savings time ends, in milliseconds.

Description

This constructor creates a `SimpleTimeZone` that uses the given offset from GMT, has the specified ID, and uses daylight savings time. Daylight savings time begins and ends at the specified dates and times.

For example, Brazil Eastern Time (BET) is three hours behind GMT. Daylight savings time for BET starts on the first Sunday in April at 2:00 AM, and ends on the last Sunday in October, also at 2:00 A.M. To construct a `TimeZone` that represents BET, you would use the following:

```
new SimpleTimeZone(-3 * 60 * 60 * 1000, "BET",
                   Calendar.APRIL, 1,
                   Calendar.SUNDAY, 2 * 60 * 60 * 1000,
                   Calendar.OCTOBER, -1,
                   Calendar.SUNDAY, 2 * 60 * 60 * 1000)
```

Instance Methods

clone

public Object clone()

Returns

A copy of this `SimpleTimeZone`.

Overrides

`TimeZone.clone()`

Description

This method creates a copy of this `SimpleTimeZone` and returns it.

equals

public boolean equals(Object obj)

Parameters

`obj`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `obj` is an instance of `SimpleTimeZone`, and it contains the same value as the object this method is associated with.

getOffset

`public int getOffset(int era, int year, int month, int day, int dayOfWeek, int millis)`

Parameters

`era`

The era.

`year`

The year.

`month`

The month.

`day`

The day.

`dayOfWeek`

The day of the week.

`millis`

The time of day in milliseconds.

Returns

An offset from GMT, in milliseconds.

Overrides

`TimeZone.getOffset()`

Description

This method calculates an offset from GMT for the given date for this `SimpleTimeZone`. In other words, the offset takes daylight savings time into account. The return value should be added to GMT to get local time.

getRawOffset

public int getRawOffset()

Returns

An offset from GMT, in milliseconds.

Overrides

`TimeZone.getRawOffset()`

Description

This method returns the raw offset from GMT for this `SimpleTimeZone`. In other words, the offset does not take daylight savings time into account.

hashCode

public synchronized int hashCode()

Returns

A hashcode for this SimpleTimeZone.

Overrides

`Object.hashCode()`

Description

This method returns a hashcode for this object.

inDaylightTime

public boolean inDaylightTime(Date date)

Parameters

`date`

The date to be tested.

Returns

`true` if the given date is between the start and end of daylight savings time for this `SimpleTimeZone`; `false` otherwise.

Overrides

`TimeZone.inDaylightTime()`

Description

This method returns a `boolean` value that indicates if the given date is in daylight savings time for this `SimpleTimeZone`.

setEndRule

`public void setEndRule(int month, int dayOfWeekInMonth, int dayOfWeek, int time)`

Parameters

`month`

The month when daylight savings time ends.

`DayOfWeekInMonth`

The week of the month when daylight savings time ends.

`dayOfWeek`

The day of the week when daylight savings time ends.

`time`

The time of day when daylight savings time ends, in milliseconds.

Description

This method sets the time when daylight savings time ends for this `SimpleTimeZone`. For example, to set the end of daylight savings time to 2 A.M. on the last Sunday in October, you would use the following:

```
setEndRule(Calendar.OCTOBER, -1, Calendar.SUNDAY,
           2 * 60 * 60 * 1000)
```

setRawOffset

public void setRawOffset(int offsetMillis)

Parameters

`offsetMillis`

The new raw offset from GMT, in milliseconds.

Overrides

`TimeZone.setRawOffset()`

Description

This method is used to set the raw offset value for this `SimpleTimeZone`.

setStartRule

```
public void setStartRule(int month, int dayOfWeekInMonth, int dayOfWeek, int time)
```

Parameters

`month`

The month when daylight savings time begins.

`DayOfWeekInMonth`

The week of the month when daylight savings time begins.

`dayOfWeek`

The day of the week when daylight savings time begins.

`time`

The time of day when daylight savings time begins, in milliseconds.

Description

This method sets the time when daylight savings time begins for this `SimpleTimeZone`. For example, to set the beginning of daylight savings time to 2 A.M. on the first Sunday in April, you would use the following:

```
setEndRule(Calendar.APRIL, 1, Calendar.SUNDAY,
           2 * 60 * 60 * 1000)
```

setStartYear

public void setStartYear(int year)

Parameters

`year`

The year when daylight savings time begins.

Description

This method sets the year after which the start and end rules for daylight savings time are observed.

useDaylightTime

public boolean useDaylightTime()

Returns

`true` if this `SimpleTimeZone` uses daylight savings time; `false` otherwise.

Overrides

`TimeZone.useDaylightTime()`

Description

This method returns a `boolean` value that indicates whether or not this `SimpleTimeZone` uses daylight savings time.

Inherited Methods

Method	Inherited From	Method	Inherited From
`finalize()`	`Object`	`getClass()`	`Object`
`getID()`	`TimeZone`	`notify()`	`Object`
`notifyAll()`	`Object`	`setID()`	`TimeZone`
`toString()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`Calendar`, `Cloneable`, `Date`, `GregorianCalendar`, `Locale`, `TimeZone`

PREVIOUS
ResourceBundle

HOME
BOOK INDEX

NEXT
Stack

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

StringTokenizer

Name

StringTokenizer

Synopsis

Class Name:

```
java.util.StringTokenizer
```

Superclass:

```
java.lang.Object
```

Immediate Subclasses:

None

Interfaces Implemented:

```
java.util.Enumeration
```

Availability:

JDK 1.0 or later

Description

The StringTokenizer class implements a simple, delimiter-based string tokenizer. In other words, a StringTokenizer is used to break a String into tokens separated by delimiter characters. By default, the class uses the standard whitespace characters as delimiters, but you can specify any delimiter characters you want, either when the StringTokenizer is created or on a token-by-token basis. You can also specify whether the delimiters are returned as tokens or not. A StringTokenizer returns the tokens in its String in order, either as String objects or as Objects in an Enumeration.

StringTokenizer shouldn't be confused with the more complex java.io.StreamTokenizer, which parses a stream into tokens that are similar to those used in the Java language.

Class Summary

```
public class java.util.StringTokenizer extends java.lang.Object
             implements java.util.Enumeration {
  // Constructors
  public StringTokenizer(String str);
  public StringTokenizer(String str, String delim);
  public StringTokenizer(String str, String delim, boolean returnTokens);
  // Instance Methods
  public int countTokens();
  public boolean hasMoreElements();
  public boolean hasMoreTokens();
  public Object nextElement();
  public String nextToken();
  public String nextToken(String delim);
}
```

Constructors

StringTokenizer

public StringTokenizer(String str)

Parameters

    str

        The String to be tokenized.

Description

This constructor creates a `StringTokenizer` that breaks apart the given string using the default delimiter characters. The default delimiters are the standard whitespace characters: space, tab (`\t`), carriage return (`\r`), and newline (`\n`).

public StringTokenizer(String str, String delim)

Parameters

`str`

The `String` to be tokenized.

`delim`

The delimiter characters.

Description

This constructor creates a `StringTokenizer` that breaks apart the given string using the characters in `delim` as delimiter characters.

`public StringTokenizer(String str, String delim, boolean returnTokens)`

Parameters

`str`

The `String` to be tokenized.

`delim`

The delimiter characters.

`returnTokens`

A `boolean` value that indicates whether or not delimiters should be returned as tokens.

Description

This constructor creates a `StringTokenizer` that breaks apart the given string using the characters in `delim` as delimiter characters. If `returnTokens` is `true`, the delimiters are returned as tokens; otherwise they are skipped.

Instance Methods

countTokens

public int countTokens()

Returns

> The number of tokens remaining in the string.

Description

> This method returns the number of tokens that remain in the string, which is the same as the number of times `nextToken()` can be called before an exception is thrown.

hasMoreElements

public boolean hasMoreElements()

Returns

> `true` if there are more tokens to be returned; `false` otherwise.

Implements

> `Enumeration.hasMoreElements()`

Description

> This method returns the result of calling `hasMoreTokens()`.

hasMoreTokens

public boolean hasMoreTokens()

Returns

> `true` if there are more tokens to be returned; `false` otherwise.

Description

This method returns `true` if this object has more tokens to return.

nextElement

public Object nextElement()

Returns

The next token as an `Object`.

Throws

`NoSuchElementException`

If there are no more tokens in the string.

Implements

`Enumeration.nextElement()`

Description

This method returns result of calling `nextToken()`.

nextToken

public String nextToken()

Returns

The next token as a `String`.

Throws

`NoSuchElementException`

If there are no more tokens in the string.

Description

This method returns the next token.

public String nextToken(String delim)

Parameters

delim

The delimiter characters.

Returns

The next token as a String.

Throws

NoSuchElementException

If there are no more tokens in the string.

Description

This method sets the delimiter characters to the characters in the given string and then returns the next token. The change in delimiters persists until another call to this method changes them again.

Inherited Methods

Method	Inherited From	Method	Inherited From
clone()	Object	equals(Object)	Object
finalize()	Objxect	getClass()	Object
hashCode()	Object	notify()	Object
notifyAll()	Object	toString()	Object
wait()	Object	wait(long)	Object
wait(long, int)	Object		

See Also

Enumeration, NoSuchElementException, StreamTokenizer, String

PREVIOUS HOME NEXT

TimeZone

Name

TimeZone

Synopsis

Class Name:

`java.util.TimeZone`

Superclass:

`java.lang.Object`

Immediate Subclasses:

`java.util.SimpleTimeZone`

Interfaces Implemented:

`java.io.Serializable`, `java.lang.Cloneable`

Availability:

New as of JDK 1.1

Description

The `TimeZone` class is an `abstract` class that represents a time zone offset. In addition, the class incorporates knowledge about daylight savings time. Usually, you get a `TimeZone` object by calling `getDefault()`. This method creates a `TimeZone` that is appropriate for the current locale. You can also call `getTimeZone()` to obtain a `TimeZone` object for a specific time zone.

Class Summary

```
public abstract class java.util.TimeZone extends java.lang.Object
                      implements java.io.Serializable, java.lang.Cloneable {
  // Class Methods
  public static synchronized String[] getAvailableIDs();
  public static synchronized String[] getAvailableIDs(int rawOffset);
  public static synchronized TimeZone getDefault();
  public static synchronized TimeZone getTimeZone(String ID);
  public static synchronized void setDefault(TimeZone zone);
  // Instance Methods
  public Object clone();
  public String getID();
  public abstract int getOffset(int era, int year, int month,
                               int day, int dayOfWeek, int milliseconds);
  public abstract int getRawOffset();
  public abstract boolean inDaylightTime(Date date);
  public void setID(String ID);
  public abstract void setRawOffset(int offsetMillis);
  public abstract boolean useDaylightTime();
}
```

Class Methods

getAvailiableIDs

public static synchronized String[] getAvailableIDs()

Returns

An array of strings that contains the predefined time zone IDs.

Description

This method returns a list of the predefined time zone IDs. Time zones are defined for the following ID values, starting from Greenwich, England, and progressing eastward around the world:

GMT	Greenwich Mean Time
ECT	European Central Time
EET	Eastern European Time
ART	(Arabic) Egypt Standard Time
EAT	Eastern African Time
MET	Middle East Time
NET	Near East Time
PLT	Pakistan Lahore Time
IST	India Standard Time
BST	Bangladesh Standard Time
VST	Vietnam Standard Time

CTT	China Taiwan Time
JST	Japan Standard Time
ACT	Australia Central Time
AET	Australia Eastern Time
SST	Solomon Standard Time
NST	New Zealand Standard Time
MIT	Midway Islands Time
HST	Hawaii Standard Time
AST	Alaska Standard Time
PST	Pacific Standard Time
PNT	Phoenix Standard Time
MST	Mountain Standard Time
CST	Central Standard Time
EST	Eastern Standard Time
IET	Indiana Eastern Standard Time
PRT	Puerto Rico and U.S. Virgin Islands Time
CNT	Canada Newfoundland Time
AGT	Argentina Standard Time
BET	Brazil Eastern Time
CAT	Central African Time

public static synchronized String[] getAvailableIDs(int rawOffset)

Returns

An array of strings that contains the predefined time zone IDs with the given raw offset value.

Description

This method returns a list of the predefined time zone IDs that have the given raw offset value from `GMT`. For example, both `PNT` and `MST` have an offset of GMT-07:00.

getDefault

public static synchronized TimeZone getDefault()

Returns

A `TimeZone` that represents the local time zone.

Description

This method returns the default `TimeZone` object for the local system.

getTimeZone

public static synchronized TimeZone getTimeZone(String ID)

Parameters

`ID`

The ID of a time zone.

Returns

A `TimeZone` that represents the time zone with the given ID.

Description

This method returns the `TimeZone` object that corresponds to the time zone with the given ID.

setDefault

public static synchronized void setDefault(TimeZone zone)

Parameters

`zone`

The new default time zone.

Description

This method sets the `TimeZone` that is returned by `getDefault()`.

Instance Methods

clone

public Object clone()

Returns

A copy of this `TimeZone`.

Overrides

`Object.clone()`

Description

This method creates a copy of this `TimeZone` and returns it.

getID

public String getID()

Returns

`The ID of this TimeZone`.

Description

This method returns the ID string of this `TimeZone`.

getOffset

`public abstract int getOffset(int era, int year, int month, int day, int dayOfWeek, int millis)`

Parameters

`era`

The era.

`year`

The year.

`month`

The month.

`day`

The day.

`dayOfWeek`

The day of the week.

`millis`

The time of day in milliseconds.

Returns

An offset from GMT, in milliseconds.

Description

This method calculates an offset from GMT for the given date for this `TimeZone`. In other words, the offset takes daylight savings time into account. The return value should be added to GMT to get local time.

getRawOffset

public abstract int getRawOffset()

Returns

An offset from GMT, in milliseconds.

Description

This method returns the raw offset from GMT for this `TimeZone`. In other words, the offset does not take daylight savings time into account.

inDaylightTime

public abstract boolean isDaylightTime(Date date)

Parameters

`date`

The date to be tested.

Returns

`true` if the given date is between the start and end of daylight savings time for this `TimeZone`; `false` otherwise.

Description

This method returns a `boolean` value that indicates if the given date is in daylight savings time for this `TimeZone`.

setID

public void setID(String ID)

Parameters

`ID`

The new time zone ID.

Description

This method sets the ID of this `TimeZone`.

setRawOffset

public abstract void setRawOffset(int offsetMillis)

Parameters

`offsetMillis`

The new raw offset from GMT, in milliseconds.

Description

This method is used to set the raw offset value for this `TimeZone`.

useDaylightTime

public abstract boolean useDaylightTime()

Returns

`true` if this `TimeZone` uses daylight savings time; `false` otherwise.

Description

This method returns a `boolean` value that indicates whether or not this `TimeZone` uses daylight savings time.

Inherited Methods

Method	Inherited From	Method	Inherited From
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Calendar`, `Cloneable`, `GregorianCalendar`, `Locale`, `Serializable`, `SimpleTimeZone`

Vector

Name

Vector

Synopsis

Class Name:

`java.util.Vector`

Superclass:

`java.lang.Object`

Immediate Subclasses:

`java.util.Stack`

Interfaces Implemented:

`java.io.Serializable`, `java.lang.Cloneable`

Availability:

JDK 1.0 or later

Description

The Vector class implements a variable-length array that can hold any kind of object. Like an array, the elements in a Vector are accessed with an integer index. However, unlike an array, the size of a Vector can grow and shrink as needed to accommodate a changing number of objects. Vector provides methods to add and remove elements, as well as ways to search for objects in a Vector and iterate through all of the objects.

The initial capacity of a Vector specifies how many objects it can contain before more space must be allocated. The capacity increment specifies how much more space is allocated each time the Vector needs to increase its capacity. You can fine-tune the performance of a Vector by adjusting the initial capacity and capacity increment.

Class Summary

```
public class java.util.Vector extends java.lang.Object
             implements java.io.Serializable, java.lang.Cloneable {
  // Variables
  protected int capacityIncrement;
  protected int elementCount;
  protected Object[] elementData;
  // Constructors
  public Vector();
  public Vector(int initialCapacity);
  public Vector(int initialCapacity, int capacityIncrement);
  // Instance Methods
  public final synchronized void addElement(Object obj);
  public final int capacity();
  public synchronized Object clone();
  public final boolean contains(Object elem);
  public final synchronized void copyInto(Object[] anArray);
  public final synchronized Object elementAt(int index);
  public final synchronized Enumeration elements();
  public final synchronized void ensureCapacity(int minCapacity);
  public final synchronized Object firstElement();
  public final int indexOf(Object elem);
  public final synchronized int indexOf(Object elem, int index);
  public final synchronized void insertElementAt(Object obj, int index);
  public final boolean isEmpty();
  public final synchronized Object lastElement();
  public final int lastIndexOf(Object elem);
  public final synchronized int lastIndexOf(Object elem, int index);
  public final synchronized void removeAllElements();
  public final synchronized boolean removeElement(Object obj);
  public final synchronized void removeElementAt(int index);
  public final synchronized void setElementAt(Object obj, int index);
  public final synchronized void setSize(int newSize);
  public final int size();
  public final synchronized String toString();
  public final synchronized void trimToSize();
}
```

Variables

capacityIncrement

protected int capacityIncrement

Description

> The amount by which the internal array grows when more space is needed. If the value is 0, the internal array doubles in size when more space is needed.

elementCount

protected int elementCount

Description

> The count of how many objects are contained in this `Vector`.

elementData

protected Object[] elementData

Description

> The array that holds the contents of this `Vector`.

Constructors

Vector

public Vector()

Description

This constructor creates an empty `Vector` with the default capacity of 10 and the default capacity increment of 0.

public Vector(int initialCapacity)

Parameters

`initialCapacity`

The initial capacity of the `Vector`.

Description

This constructor creates an empty `Vector` with the given capacity and the default capacity increment of 0.

public Vector(int initialCapacity, int capacityIncrement)

Parameters

`initialCapacity`

The initial capacity of the `Vector`.

`CapacityIncrement`

The amount to increase the capacity when more space is needed.

Description

This constructor creates an empty `Vector` with the given capacity and capacity increment.

Instance Methods

addElement

public final synchronized void addElement(Object obj)

Parameters

`obj`

The object to be added.

Description

This method adds the given object to this `Vector` as its last element and increases its size by 1. The capacity of the `Vector` is increased if its size becomes greater than its capacity. Any kind of object can be added to a `Vector`.

capacity

public final int capacity()

Returns

`The capacity of this Vector`.

Description

This method returns the size of the internal array of this `Vector`.

clone

public synchronized Object clone()

Returns

A copy of this `Vector`.

Overrides

`Object.clone()`

Description

This method creates a copy of this `Vector` and returns it.

contains

public final boolean contains(Object elem)

Parameters

`elem`

The object to be found.

Returns

`true` if the given object is contained in this `Vector`; `false` otherwise.

Description

This method determines whether or not the given object is contained in this `Vector`.

copyInto

public final synchronized void copyInto(Object[] anArray)

Parameters

`anArray`

The array to be filled.

Throws

`ArrayIndexOutOfBoundsException`

If the given array is too small to hold all of the objects in this `Vector`.

Description

This method copies the object references in this `Vector` to the given array.

elementAt

public final synchronized Object elementAt(int index)

Parameters

`index`

The index of the object to be returned.

Returns

The object at the position given by `index`.

Throws

`ArrayIndexOutOfBoundsException`

If `index` is less than zero or greater than the size of this `Vector`.

Description

This method returns the object at the given index in this `Vector`.

elements

public final synchronized Enumeration elements()

Returns

The objects in this `Vector` as an `Enumeration`.

Description

This method returns an `Enumeration` that iterates through the objects in this `Vector`.

ensureCapacity

public final synchronized void ensureCapacity(int minCapacity)

Parameters

`minCapacity`

The minimum new capacity.

Description

This method expands the internal array, if necessary, to make the capacity of the `Vector` at least `minCapacity`.

firstElement

public final synchronized Object firstElement()

Returns

The first object in this Vector.

Throws

`NoSuchElementException`

If the `Vector` is empty.

Description

This method returns the object at index 0 in this `Vector`.

indexOf

public final int indexOf(Object elem)

Parameters

`elem`

The object to be found.

Returns

The index of the given object or `-1` if it cannot be found.

Description

This method returns the index of the first occurrence of the given object in this `Vector`.

public final int indexOf(Object elem, int index)

Parameters

`elem`

The object to be found.

`index`

The starting index.

Returns

The index of the next occurrence of the given object after the specified index or `-1` if it cannot be found.

Description

This method returns the index of the next occurrence of the given object in this `Vector` after the given index.

insertElementAt

public final synchronized void insertElementAt(Object obj, int index)

Parameters

`obj`

The object to be inserted.

`index`

The index at which to insert the object.

Throws

`ArrayIndexOutOfBoundsException`

If `index` is less than zero or greater than the size of this `Vector`.

Description

This method inserts the given object at the given index in this `Vector`. Each object in the `Vector` with an index greater than or equal to `index` is shifted upward in the `Vector`, so that it has an index of one greater than it did previously.

isEmpty

public final boolean isEmpty()

Returns

`true` if there are no objects in the `Vector`; `false` otherwise.

Description

This method returns a `boolean` value that indicates whether or not the `Vector` is empty.

lastElement

public final synchronized Object lastElement()

Returns

The last object in this `Vector`.

Throws

`NoSuchElementException`

If the `Vector` is empty.

Description

This method returns the last object in this `Vector`.

lastIndexOf

public final int lastIndexOf(Object elem)

Parameters

`elem`

The object to be found.

Returns

The index of the given object or `-1` if it cannot be found.

Description

This method returns the index of the last occurrence of the given object in this `Vector`.

public final synchronized int lastIndexOf(Object elem, int index)

Parameters

`elem`

The object to be found.

`index`

The starting index.

Returns

The index of the last occurrence of the given object before the specified index or `-1` if it cannot be found.

Description

This method returns the index of the last occurrence of the given object in this `Vector` before the given index. In other words, the method searches backwards from `index` for the next occurrence.

removeAllElements

public final synchronized void removeAllElements()

Description

This method removes all of the objects from this `Vector`, but does not change its capacity or capacity increment.

removeElement

public final synchronized boolean removeElement(Object obj)

Parameters

`obj`

The object to be removed.

Returns

`true` if the given object is found and removed; `false` otherwise.

Description

This method searches for the first occurrence of the given object in this `Vector` and removes it. If the object is found, each object in the `Vector` with an index greater than or equal to the index of the removed object is shifted downward in the `Vector`, so that it has an index of one less than it did previously.

removeElementAt

public final synchronized void removeElementAt(int index)

Parameters

`index`

The index of the object to be removed.

Throws

`ArrayIndexOutOfBoundsException`

If `index` is less than zero or greater than the size of this `Vector`.

Description

This method removes the object at the given index from this `Vector`. Each object in the `Vector` with an index greater than or equal to `index` is shifted downward in the `Vector`, so that it has an index of one less than it did previously.

setElementAt

public final synchronized void setElementAt(Object obj, int index)

Parameters

`obj`

The object to be put in the `Vector`.

`index`

The index at which to put the object.

Throws

`ArrayIndexOutOfBoundsException`

If `index` is less than zero or greater than the size of this `Vector`.

Description

This method puts the given object at the given index in this `Vector`, replacing the object that was previously there.

setSize

public final synchronized void setSize(int newSize)

Parameters

`newSize`

The new size.

Description

This method sets the size (not the capacity) of this `Vector`. If the new size is bigger than the current size, `null` elements are added to the end of the `Vector`. If the new size is smaller than the current size, elements are discarded from index `newSize` to the end of the `Vector`.

size

public final int size()

Returns

The size of this `Vector`.

Description

This method returns the number of objects contained in this `Vector`.

toString

public final synchronized String toString()

Returns

A string that represents this `Vector`.

Overrides

`Object.toString()`

Description

This method returns a string representation of this `Vector`. The string includes every object that is contained in the `Vector`, so the string returned by `toString()` can be quite long.

trimToSize

public final synchronized void trimToSize()

Description

This method sets the capacity of this `Vector` equal to its size. You can use this method to minimize the storage space used by the `Vector`, but any subsequent calls to `addElement()` or `insertElement()` forces the `Vector` to allocate more space.

Inherited Methods

Method	Inherited From	Method	Inherited From
`equals(Object)`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`ArrayIndexOutOfBoundsException`, `Cloneable`, `Enumeration`, `NoSuchElementException`, `Serializable`, `Stack`

CheckedInputStream

Name

CheckedInputStream

Synopsis

Class Name:

```
java.util.zip.CheckedInputStream
```

Superclass:

```
java.io.FilterInputStream
```

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `CheckedInputStream` class represents an `InputStream` with an associated checksum. In other words, a `CheckedInputStream` wraps a regular input stream and computes a checksum value as data is read from the stream. The checksum can verify the integrity of the data. When you create a `CheckedInputStream`, you must specify an object that implements the `Checksum` interface that computes the checksum.

Class Summary

```
public class java.util.zip.CheckedInputStream
             extends java.io.FilterInputStream {
  // Constructors
  public CheckedInputStream(InputStream in, Checksum cksum);

  // Instance Methods
  public Checksum getChecksum();
  public int read();
  public int read(byte[] buf, int off, int len);
  public long skip(long n);
}
```

Constructors

CheckedInputStream

public CheckedInputStream(InputStream in, Checksum cksum)

Parameters

`in`

The underlying input stream to use as a data source.

`cksum`

The checksum object.

Description

This constructor creates a `CheckedInputStream` that reads data from the given `InputStream` and updates the given `Checksum`.

Instance Methods

getChecksum

public Checksum getChecksum()

Returns

The `Checksum` associated with this input stream.

Description

This method returns the `Checksum` object associated with this input stream.

read

public int read() throws IOException

Returns

The next byte from the stream or `-1` if the end of the stream has been reached.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterInputStream.read()`

Description

This method reads the next byte from the underlying `InputStream` and then updates the checksum. The method blocks until some data is available, the end of the stream is detected, or an exception is thrown.

public int read(byte[] buf, int off, int len) throws IOException

Parameters

`buf`

An array of bytes to be filled from the stream.

`off`

An offset into the byte array.

`len`

The number of bytes to read.

Returns

The number of bytes read or `-1` if the end of the stream is encountered immediately.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterInputStream.read(byte[], int, int)`

Description

This method reads up to `len` bytes from the underlying `InputStream` and places them into the given array starting at `off`. The checksum is then updated with the data that has been read. The method blocks until some data is available.

skip

public long skip(long n) throws IOException

Parameters

`n`

The number of bytes to skip.

Returns

The actual number of bytes skipped.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterInputStream.skip()`

Description

This method skips over the specified number of bytes in the underlying `InputStream`. The skipped bytes are not calculated into the checksum.

Inherited Methods

Method	Inherited From	Method	Inherited From
`available()`	`FilterInputStream`	`clone()`	`Object`
`close()`	`FilterInputStream`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`mark(int)`	`FilterInputStream`
`markSupported()`	`FilterInputStream`	`notify()`	`Object`
`notifyAll()`	`Object`	`read(byte[])`	`FilterInputStream`
`reset()`	`FilterInputStream`	`toString()`	`Object`

`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Checksum`, `FilterInputStream`, `InputStream`, `IOException`

PREVIOUS Adler32 | HOME BOOK INDEX | NEXT CheckedOutputStream

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

CheckedOutputStream

Name

CheckedOutputStream

Synopsis

Class Name:

`java.util.zip.CheckedOutputStream`

Superclass:

`java.io.FilterOutputStream`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `CheckOutputStream` class represents an `OutputStream` with an associated checksum. In other words, a `CheckedOutputStream` wraps a regular output stream and computes a checksum value as data is written to the stream. The checksum can verify the integrity of the data. When you create a `CheckedOutputStream`, you must specify an object that implements the `Checksum` interface that computes the checksum.

Class Summary

```
public class java.util.zip.CheckedOutputStream
             extends java.io.FilterOutputStream {
  // Constructors
  public CheckedOutputStream(OutputStream out, Checksum cksum);

  // Instance Methods
  public Checksum getChecksum();
  public void write(int b);
  public void write(byte[] b, int off, int len);
}
```

Constructors

CheckedOutputStream

public CheckedOutputStream(OutputStream out, Checksum cksum)

Parameters

`out`

The underlying output stream.

`cksum`

The checksum object.

Description

This constructor creates a `CheckedOutputStream` that writes data to the given `OutputStream` and updates the given `Checksum`.

Instance Methods

getChecksum

public Checksum getChecksum()

Returns

The `Checksum` associated with this output stream.

Description

This method returns the `Checksum` object associated with this output stream.

write

public void write(int b) throws IOException

Parameters

`b`

The value to write.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterOutputStream.write(int)`

Description

This method writes a byte that contains the lowest eight bits of the given integer value to the underlying `OutputStream` and then updates the checksum.

public void write(byte[] b, int off, int len) throws IOException

Parameters

`b`

An array of bytes to write to the stream.

`off`

An offset into the byte array.

`len`

The number of bytes to write.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterOutputStream.write(byte[], int, int)`

Description

This method writes `len` bytes to the underlying `OutputStream` from the given array, starting at `off`. The checksum is then updated with the data that has been written.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`close()`	`FilterOutputStream`

`equals(Object)`	`Object`	`finalize()`	`Object`
`flush()`	`FilterOutputStream`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`	`write(byte[])`	`FilterOutputStream`

See Also

`Checksum`, `FilterOutputStream`, `IOException`, `OutputStream`

Checksum

Name

Checksum

Synopsis

Interface Name:

`java.util.zip.Checksum`

Super-interface:

None

Immediate Sub-interfaces:

None

Implemented By:

`java.util.zip.Adler32`, `java.util.zip.CRC32`

Availability:

New as of JDK 1.1

Description

The `Checksum` interface defines the methods that are needed to compute a checksum value for a stream of data. The checksum value can be used for error checking purposes. Note, however, that the checksum value must fit into a `long` value, so this interface is not suitable for cryptographic checksum algorithms.

The `Adler32` and `CRC32` classes implement the `Checksum` interface, using the Adler-32 and CRC-32 algorithms, respectively. The `CheckedInputStream` and `CheckedOutputStream` classes provide a higher-level mechanism for computing checksums on data streams.

Class Summary

```
public abstract interface java.util.zip.Checksum {
  // Methods
  public abstract long getValue();
  public abstract void reset();
  public abstract void update(int b);
  public abstract void update(byte[] b, int off, int len);
}
```

Methods

getValue

public abstract long getValue()

Returns

> The current checksum value.

Description

> This method returns the current value of this checksum.

reset

public abstract void reset()

Description

This method resets the checksum to its initial value, making it appear as though it has not been updated by any data.

update

public abstract void update(int b)

Parameters

`b`

The value to be added to the data stream for the checksum calculation.

Description

This method adds the specified value to the data stream and updates the checksum value. The method uses only the lowest eight bits of the given `int`.

public abstract void update(byte[] b, int off, int len)

Parameters

`b`

An array of bytes to be added to the data stream for the checksum calculation.

`off`

An offset into the byte array.

`len`

The number of bytes to use.

Description

This method adds `len` bytes from the specified array, starting at `off`, to the data stream and updates the checksum value.

See Also

`Adler32`, `CheckedInputStream`, `CheckedOutputStream`, `CRC32`

PREVIOUS CheckedOutputStream | HOME BOOK INDEX | NEXT CRC32

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

CRC32

Name

CRC32

Synopsis

Class Name:

    java.util.zip.CRC32

Superclass:

    java.lang.Object

Immediate Subclasses:

None

Interfaces Implemented:

    java.util.zip.Checksum

Availability:

New as of JDK 1.1

Description

The `CRC32` class implements the `Checksum` interface using the CRC-32 algorithm. This algorithm is significantly slower than Adler-32 and only slightly more reliable.

Class Summary

```
public class java.util.zip.CRC32 extends java.lang.Object
             implements java.util.zip.Checksum {
  // Constructors
  public CRC32();

  // Instance Methods
  public long getValue();
  public void reset();
  public void update(int b);
  public void update(byte[] b);
  public native void update(byte[] b, int off, int len);
}
```

Constructors

CRC32

public CRC32()

Description

> This constructor creates an `CRC32` object.

Instance Methods

getValue

public long getValue()

Returns

The current checksum value.

Implements

`Checksum.getValue()`

Description

This method returns the current value of this checksum.

reset

public void reset()

Implements

`Checksum.reset()`

Description

This method resets the checksum to its initial value, making it appear as though it has not been updated by any data.

update

public void update(int b)

Parameters

`b`

The value to be added to the data stream for the checksum calculation.

Implements

`Checksum.update(int)`

Description

This method adds the specified value to the data stream and updates the checksum value. The method uses only the lowest eight bits of the given `int`.

public void update(byte[] b)

Parameters

`b`

An array of bytes to be added to the data stream for the checksum calculation.

Description

This method adds the bytes from the specified array to the data stream and updates the checksum value.

public native void update(byte[] b, int off, int len)

Parameters

`b`

An array of bytes to be added to the data stream for the checksum calculation.

`off`

An offset into the byte array.

`len`

The number of bytes to use.

Implements

`Checksum.update(byte[], int, int)`

Description

This method adds `len` bytes from the specified array, starting at `off`, to the data stream and updates the checksum value.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Adler32`, `Checksum`

PREVIOUS HOME NEXT

Checksum BOOK INDEX DataFormatException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

DataFormatException

Name

DataFormatException

Synopsis

Class Name:

    java.util.zip.DataFormatException

Superclass:

    java.lang.Exception

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

A `DataFormatException` is thrown when data is not in the expected format, which can mean that the data is invalid or corrupt. In particular, the `inflate()` methods of `Inflater` throw this exception if they encounter data in an unexpected format.

Class Summary

```
public class java.util.zip.DataFormatException
             extends java.lang.Exception {
  // Constructors
  public DataFormatException();
  public DataFormatException(String s);
}
```

Constructors

DataFormatException

protected DataFormatException()

Description

> This constructor creates a `DataFormatException` with no associated detail message.

public DataFormatException(String s)

Parameters

> `s`
>
> > The detail message.

Description

> This constructor creates a `DataFormatException` with the specified detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`

`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Throwable`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Exception`, `Inflater`, `Throwable`

PREVIOUS HOME NEXT
CRC32 BOOK INDEX Deflater

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Deflater

Name

Deflater

Synopsis

Class Name:

`java.util.zip.Deflater`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `Deflater` class provides support for general-purpose data compression. The class uses the ZLIB compression algorithms described in RFC 1950, RFC 1951, and RFC 1952. These documents can be found at:

- *ftp://ds.internic.net/rfc/rfc1950.txt*
- *ftp://ds.internic.net/rfc/rfc1951.txt*
- *ftp://ds.internic.net/rfc/rfc1952.txt*

The `Inflater` class uncompresses data that has been compressed using `Deflater`.

The `DeflaterOutputStream` uses an internal `Deflater` to compress data. Typically, you do not need to create a `Deflater`; instead, you can just use an instance of one of the subclasses of `DeflaterOutputStream` : `GZIPOutputStream` or `ZipOutputStream`.

Class Summary

```
public class java.util.zip.Deflater extends java.lang.Object {
  // Constants
  public static final int BEST_COMPRESSION;
  public static final int BEST_SPEED;
  public static final int DEFAULT_COMPRESSION;
  public static final int DEFAULT_STRATEGY;
  public static final int DEFLATED;
  public static final int FILTERED;
  public static final int HUFFMAN_ONLY;
  public static final int NO_COMPRESSION;
  // Constructors
  public Deflater();
  public Deflater(int level);
  public Deflater(int level, boolean nowrap);
  // Public Instance Methods
  public int deflate(byte[] b);
  public synchronized native int deflate(byte[] b, int off, int len);
  public synchronized native void end();
  public synchronized void finish();
  public synchronized boolean finished();
  public synchronized native int getAdler();
  public synchronized native int getTotalIn();
  public synchronized native int getTotalOut();
  public boolean needsInput();
  public synchronized native void reset();
  public void setDictionary(byte[] b);
  public synchronized native void setDictionary(byte[] b, int off, int len);
  public void setInput(byte[] b);
  public synchronized void setInput(byte[] b, int off, int len);
  public synchronized void setLevel(int level);
  public synchronized void setStrategy(int strategy);
  // Protected Instance Methods
  protected void finalize();
}
```

Constants

BEST_COMPRESSION

public static final int BEST_COMPRESSION = 9

Description

> A constant that represents a compression level that sacrifices speed for the smallest compressed data size. The compression level for a `Deflater` object can be set with `setLevel()`, where the level ranges from 0 to 9.

BEST_SPEED

public static final int BEST_SPEED = 1

Description

> A constant that represents a compression level that sacrifices compressed data size for speed. The compression level for a `Deflater` object can be set with `setLevel()`, where the level ranges from 0 to 9.

DEFAULT_COMPRESSION

public static final int DEFAULT_COMPRESSION = -1

Description

> A constant that represents the default compression level.

DEFAULT_STRATEGY

public static final int DEFAULT_STRATEGY

Description

> A constant that represents the default compression strategy.

DEFLATED

public static final int DEFLATED

Description

> A constant that represents a compression method that uses the deflate algorithm.

FILTERED

public static final int FILTERED

Description

> A constant that represents a compression strategy that works well for small values with a random distribution.

HUFFMAN_ONLY

public static final int HUFFMAN_ONLY

Description

> A constant that represents a compression strategy that uses only Huffman coding.

NO_COMPRESSION

public static final int NO_COMPRESSION = 0

Description

> A constant that represents a compression level that does not compress data at all. The compression level for a `Deflater` object can be set with `setLevel()`, where the level ranges from 0 to 9.

Constructors

Deflater

public Deflater()

Description

This constructor creates a `Deflater` that generates compressed data in the ZLIB format using the `DEFAULT_COMPRESSION` level.

public Deflater(int level)

Parameters

`level`

The compression level, from 0 (`NO_COMPRESSION`) to 9 (`BEST_COMPRESSION`).

Description

This constructor creates a `Deflater` that generates compressed data in the ZLIB format using the given compression level.

public Deflater(int level, boolean nowrap)

Parameters

`level`

The compression level, from 0 (`NO_COMPRESSION`) to 9 (`BEST_COMPRESSION`).

`nowrap`

A `boolean` value that specifies whether or not the ZLIB header and checksum data are omitted from the compressed data.

Description

This constructor creates a `Deflater` that generates compressed data using the given compression level. If `nowrap` is `true`, the ZLIB header and checksum fields are not used, which means that the compressed data is in the format used by GZIP and PKZIP. If the parameter is `false`, the data is compressed into ZLIB format.

Public Instance Methods

deflate

public int deflate(byte[] b)

Parameters

`b`

A byte array to be filled.

Returns

The number of compressed bytes actually written to the array or 0 if more data may be required.

Description

This method compresses the data passed to `setInput()` and fills the given array with the compressed data. If this method returns zero, `needsInput()` should be called to determine whether the `Deflater` needs more data in its input buffer.

public synchronized native int deflate(byte[] b, int off, int len)

Parameters

`b`

A byte array to be filled.

`off`

An offset into the byte array.

`len`

The number of bytes to fill.

Returns

The number of compressed bytes actually written to the array or 0 if more data may be required.

Description

This method compresses the data passed to `setInput()` and writes `len` bytes of the compressed data into the given array, starting at `off`. If this method returns 0, `needsInput()` should be called to determine whether the `Deflater` needs more data in its input buffer.

end

public synchronized native void end()

Description

> This method discards any uncompressed input data and frees up internal buffers.

finish

public synchronized void finish()

Description

> This method tells the `Deflater` that the compression should end with the data that currently occupies the input buffer.

finished

public synchronized boolean finished()

Returns

> A `boolean` value that indicates whether or not the end of the compressed data has been reached.

Description

> This method returns `true` if the last of the compressed data has been read using `deflate()`. Otherwise it returns `false`.

getAdler

public synchronized native int getAdler()

Returns

> The Adler-32 checksum value of the uncompressed data.

Description

> This method returns an Adler32 checksum value that is calculated on the uncompressed data passed to `setInput()`.

getTotalIn

public synchronized native int getTotalIn()

Returns

The total number of bytes that have been input so far.

Description

This method returns the number of bytes that have been passed to `setInput()` since this `Deflater` was created or since `reset()`was last called.

getTotalOut

public synchronized native int getTotalOut()

Returns

The total number of bytes that have been output so far.

Description

This method returns the number of bytes that have been read from `deflate()` since this `Deflater` was created, or since `reset()` was last called.

needsInput

public boolean needsInput()

Returns

A `boolean` value that indicates whether or not the input buffer is empty.

Description

This method returns `true` if the input buffer is empty. Otherwise it returns `false`.

reset

public synchronized native void reset()

Description

This method resets the `Deflater` to the state it was in when it was created, which means that a new set of data can be compressed.

setDictionary

public void setDictionary(byte[] b)

Parameters

`b`

An array of byte values.

Description

This method sets the preset dictionary for compression using the data in the given array.

`public synchronized native void setDictionary(byte[] b, int off, int len)`

Parameters

`b`

An array of byte values.

`off`

An offset into the byte array.

`len`

The number of bytes to use.

Description

This method sets the preset dictionary for compression using `len` bytes from the given array, starting from `off`.

setInput

public void setInput(byte[] b)

Parameters

b

An array of byte values.

Description

This method places the contents of the given array into the input buffer of this `Deflater`. Use the `deflate()` method to compress the data and retrieve it in compressed form.

public synchronized void setInput(byte[] b, int off, int len)

Parameters

b

An array of byte values.

off

An offset into the byte array.

len

The number of bytes to use.

Description

This method places `len` bytes from the given array, starting at `off`, into the input buffer of this `Deflater`. Use the `deflate()` method to compress the data and retrieve it in compressed form.

setLevel

public synchronized void setLevel(int level)

Parameters

level

The compression level, from 0 (`NO_COMPRESSION`) to 9 (`BEST_COMPRESSION`).

Throws

IllegalArgumentException

If `level` is not valid.

Description

This method sets the compression level of this `Deflater`. A value of 0 corresponds to `NO_COMPRESSION`. A value of 1 indicates the fastest, least space-efficient compression level (`BEST_SPEED`). A value of 9 indicates the slowest, most space-efficient compression level (`BEST_COMPRESSION`).

setStrategy

public synchronized void setStrategy(int strategy)

Parameters

`strategy`

The compression strategy.

Throws

`IllegalArgumentException`

If `strategy` is not valid.

Description

This method sets the compression strategy of this `Deflater`, which should be one of `FILTERED`, `HUFFMAN_ONLY`, or `DEFAULT_STRATEGY`.

Protected Instance Methods

finalize

protected void finalize()

Overrides

`Object.finalize()`

Description

This method calls `end()` when this `Deflater` is garbage collected.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`getClass()`	`Object`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`toString()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`DeflaterOutputStream`, `Inflater`, `GZIPOutputStream`, `ZipOutputStream`

PREVIOUS DataFormatException | HOME BOOK INDEX | NEXT DeflaterOutputStream

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

DeflaterOutputStream

Name

DeflaterOutputStream

Synopsis

Class Name:

`java.util.zip.DeflaterOutputStream`

Superclass:

`java.io.FilterOutputStream`

Immediate Subclasses:

`java.util.zip.GZIPOutputStream`, `java.util.zip.ZipOutputStream`

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `DeflaterOutputStream` class represents an `OutputStream` with an associated `Deflater`. In other words, a `DeflaterOutputStream` wraps a regular output stream, so that data written to the stream is compressed and written to the underlying stream. Two subclasses, `GZIPOutputStream` and `ZipOutputStream`, write compressed data in widely-recognized formats.

Class Summary

```
public class java.util.zip.DeflaterOutputStream
             extends java.io.FilterOutputStream {
  // Variables
  protected byte[] buf;
  protected Deflater def;
  // Constructors
  public DeflaterOutputStream(OutputStream out);
  public DeflaterOutputStream(OutputStream out, Deflater def);
  public DeflaterOutputStream(OutputStream out, Deflater def, int size);
  // Public Instance Methods
  public void close();
  public void finish();
  public void write(int b);
  public void write(byte[] b, int off, int len);
  // Protected Instance Methods
  protected void deflate();
}
```

Variables

buf

protected byte[] buf

Description

> A buffer that holds the compressed data that is written to the underlying stream.

def

protected Deflater def

Description

The `Deflater` object that is used internally.

Constructors

DeflaterOutputStream

public DeflaterOutputStream(OutputStream out)

Parameters

`out`

The underlying output stream.

Description

This constructor creates a `DeflaterOutputStream` that writes data to the given `OutputStream`. Before being written, the data is compressed by a default `Deflater`. The `DeflaterOutputStream` uses a compression buffer with the default size of 512 bytes.

public DeflaterOutputStream(OutputStream out, Deflater def)

Parameters

`out`

The underlying output stream.

`def`

The `Deflater` object.

Description

This constructor creates a `DeflaterOutputStream` that writes data to the given `OutputStream`. Before being written, the data is compressed by the given `Deflater`. The `DeflaterOutputStream` uses a compression buffer with the default size of 512 bytes.

`public DeflaterOutputStream(OutputStream out, Deflater def, int size)`

Parameters

`out`

The underlying output stream.

`def`

The `Deflater` object.

`size`

The size of the output buffer.

Description

This constructor creates a `DeflaterOutputStream` that writes data to the given `OutputStream`. Before being written, the data is compressed by the given `Deflater`. The `DeflaterOutputStream` uses a compression buffer of the given size.

Public Instance Methods

close

public void close() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterOutputStream.close()`

Description

This method closes the stream and releases any system resources that are associated with it.

finish

public void finish() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method finishes writing compressed data to the underlying stream without closing it.

write

public void write(int b) throws IOException

Parameters

`b`

The value to write.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterOutputStream.write(int)`

Description

This method compresses a byte that contains the lowest eight bits of the given integer value and writes it to the underlying `OutputStream`. The method blocks until the byte is written.

public void write(byte[] b, int off, int len) throws IOException

Parameters

`b`

An array of bytes to write to the stream.

`off`

An offset into the byte array.

`len`

The number of bytes to write.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterOutputStream.write(byte[], int, int)`

Description

This method takes `len` bytes from the given buffer, starting at `off`, and compresses them. The method then writes the compressed data to the underlying `OutputStream`. The method blocks until all of the bytes have been written.

Protected Instance Methods

deflate

protected void deflate() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Description

> This method asks the internal `Deflater` to compress the data in the buffer for this `DeflaterOutputStream`. The method then writes the compressed contents of the buffer to the underlying stream. The method is called by both `write()` methods.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`flush()`	`FilterOutputStream`
`getClass()`	`Object`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`toString()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`
`write(byte[])`	`FilterOutputStream`		

See Also

`Deflater`, `FilterOutputStream`, `GZIPOutputStream`, `IOException`, `OutputStream`, `ZipOutputStream`

GZIPInputStream

Name

GZIPInputStream

Synopsis

Class Name:

`java.util.zip.GZIPInputStream`

Superclass:

`java.util.zip.InflaterInputStream`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `GZIPInputStream` class decompresses data that has been compressed using the GZIP format. To use it, simply construct a `GZIPInputStream` that wraps regular input stream and use the `read()` methods to read the compressed data.

Class Summary

```
public class java.util.zip.GZIPInputStream
             extends java.util.zip.InflaterInputStream {
  // Constants
  public static final int GZIP_MAGIC;
  // Variables
  protected CRC32 crc;
  protected boolean eos;
  // Constructors
  public GZIPInputStream(InputStream in);
  public GZIPInputStream(InputStream in, int size);
  // Instance Methods
  public void close();
  public int read(byte[] buf, int off, int len);
}
```

Constants

GZIP_MAGIC

public static final int GZIP_MAGIC

Description

> A constant that contains the "magic number" that appears in the header of GZIP files.

Variables

crc

protected CRC32 crc

Description

A checksum value of the uncompressed data. When an entire file has been read, this checksum is compared to a value stored in the GZIP trailer. If the values do not match, an exception is thrown from `read()`.

eos

protected boolean eos

Description

A flag that indicates whether or not the end of the compressed stream has been reached. It is set to `true` when the compressed data and the GZIP trailer have been read.

Constructors

GZIPInputStream

public GZIPInputStream(InputStream in) throws IOException

Parameters

`in`

The underlying input stream.

Throws

`IOException`

If an error occurs while reading the GZIP header.

Description

This constructor creates a `GZIPInputStream` that inflates data from the given `InputStream`. The `GZIPInputStream` uses a decompression buffer with the default size of 512 bytes. The GZIP header is read immediately.

`public GZIPInputStream(InputStream in, int size) throws IOException`

Parameters

`in`

The underlying input stream.

`size`

The size of the input buffer.

Throws

`IOException`

If an error occurs while reading the GZIP header.

Description

This constructor creates a `GZIPInputStream` that inflates data from the given `InputStream`. The `GZIPInputStream` uses a decompression buffer of the given size. The GZIP header is read immediately.

Instance Methods

close

public void close() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterInputStream.close()`

Description

This method closes this stream and releases any system resources that are associated with it.

read

public int read(byte[] buf, int off, int len) throws IOException

Parameters

`buf`

An array of bytes to be filled from the stream.

`off`

An offset into the byte array.

`len`

The number of bytes to read.

Returns

The number of bytes read or `-1` if the end of the stream is encountered immediately.

Throws

`IOException`

If any kind of I/O error occurs or the checksum of the uncompressed data does not match that in the GZIP trailer.

Overrides

`InflaterInputStream.read(byte[], int, int)`

Description

This method reads enough data from the underlying `InputStream` to return `len` bytes of uncompressed data. The uncompressed data is placed into the given array starting at `off`. The method blocks until some data is available for decompression.

Inherited Methods

Method	Inherited From	Method	Inherited From
available()	FilterInputStream	clone()	Object
equals(Object)	Object	finalize()	Object
getClass()	Object	hashCode()	Object
mark(int)	FilterInputStream	markSupported()	FilterInputStream
notify()	Object	notifyAll()	Object
read()	InflaterInputStream	read(byte[])	FilterInputStream
reset()	FilterInputStream	skip(long)	InflaterInputStream
toString()	Object	wait()	Object
wait(long)	Object	wait(long, int)	Object

See Also

`FilterInputStream`, `Inflater`, `InflaterInputStream`, `InputStream`, `IOException`

PREVIOUS HOME NEXT
DeflaterOutputStream BOOK INDEX GZIPOutputStream

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

GZIPOutputStream

Name

GZIPOutputStream

Synopsis

Class Name:

```
java.util.zip.GZIPOutputStream
```

Superclass:

```
java.util.zip.DeflaterOutputStream
```

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `GZIPOutputStream` class compresses data using the GZIP format. To use it, simply construct a `GZIPOutputStream` that wraps a regular output stream and use the `write()` methods to write compressed data.

Class Summary

```
public class java.util.zip.GZIPOutputStream
             extends java.util.zip.DeflaterOutputStream {
  // Variables
  protected CRC32 crc;
  // Constructors
  public GZIPOutputStream(OutputStream out);
  public GZIPOutputStream(OutputStream out, int size);
  // Instance Methods
  public void close();
  public void finish();
  public synchronized void write(byte[] buf, int off, int len);
}
```

Variables

crc

protected CRC32 crc

Description

> A checksum that is updated with the uncompressed stream data. The checksum value is written into the GZIP trailer.

Constructors

GZIPOutputStream

public GZIPOutputStream(OutputStream out) throws IOException

Parameters

`out`

The underlying output stream.

Throws

`IOException`

If an error occurs while writing the GZIP header.

Description

This constructor creates a `GZIPOutputStream` that writes compressed data to the given `OutputStream`. The `GZIPOutputStream` uses a compression buffer with the default size of 512 bytes.

public GZIPOutputStream(OutputStream out, int size) throws IOException

Parameters

`out`

The underlying output stream.

`size`

The size of the output buffer.

Throws

`IOException`

If an error occurs while writing the GZIP header.

Description

This constructor creates a `GZIPOutputStream` that writes compressed data to the given `OutputStream`. The `GZIPOutputStream` uses a compression buffer of the given size.

Instance Methods

close

public void close() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`DeflaterOutputStream.close()`

Description

This method closes the stream and releases any system resources that are associated with it.

finish

public void finish() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`DeflaterOutputStream.finish()`

Description

This method finishes writing compressed data to the underlying stream without closing it.

write

public void write(byte[] buf, int off, int len) throws IOException

Parameters

`buf`

An array of bytes to write to the stream.

`off`

An offset into the byte array.

`len`

The number of bytes to write.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`DeflaterOutputStream.write(byte[], int, int)`

Description

This method takes `len` bytes from the given buffer, starting at `off`, and compresses them. The method then writes the compressed data to the underlying `OutputStream`. The method blocks until all of the bytes have been written.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`flush()`	`FilterOutputStream`
`getClass()`	`Object`	`hashCode()`	`Object`

`notify()`	`Object`	`notifyAll()`	`Object`
`toString()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`
`write(int)`	`DeflaterOutputStream`	`write(byte[])`	`FilterOutputStream`

See Also

`Deflater`, `DeflaterOutputStream`, `FilterOutputStream`, `IOException`, `OutputStream`

Inflater

Name

Inflater

Synopsis

Class Name:

`java.util.zip.Inflater`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `Inflater` class provides support for general-purpose data decompression. The class uses the ZLIB compression algorithms described in RFC 1950, RFC 1951, and RFC 1952. These documents can be found at:

- *ftp://ds.internic.net/rfc/rfc1950.txt*
- *ftp://ds.internic.net/rfc/rfc1951.txt*
- *ftp://ds.internic.net/rfc/rfc1952.txt*

The `Deflater` class compresses data that can be uncompressed using `Inflater`.

The `InflaterInputStream` uses an internal `Inflater` to decompress data. Typically, you do not need to create a `Inflater`; instead, you can just use an instance of one of the subclasses of `InflaterInputStream`: `GZIPInputStream` or `ZipInputStream`.

Class Summary

```
public class java.util.zip.Inflater extends java.lang.Object {
  // Constructors
  public Inflater();
  public Inflater(boolean nowrap);
  // Public Instance Methods
  public synchronized native void end();
  public synchronized boolean finished();
  public synchronized native int getAdler();
  public synchronized int getRemaining();
  public synchronized native int getTotalIn();
  public synchronized native int getTotalOut();
  public int inflate(byte[] b);
  public synchronized native int inflate(byte[] b, int off, int len);
  public synchronized boolean needsDictionary();
  public synchronized boolean needsInput();
  public synchronized native void reset();
  public void setDictionary(byte[] b);
  public synchronized native void setDictionary(byte[] b, int off, int len);
  public void setInput(byte[] b);
  public synchronized void setInput(byte[] b, int off, int len);
  // Protected Instance Methods
  protected void finalize();
}
```

Constructors

Inflater

public Inflater()

Description

This constructor creates an `Inflater` that decompresses data in the ZLIB format.

public Inflater(boolean nowrap)

Parameters

`nowrap`

A `boolean` value that specifies whether or not the ZLIB header and checksum data are expected in the compressed data.

Description

This constructor creates an `Inflater` that decompresses data. If `nowrap` is `true`, the ZLIB header and checksum fields are not expected, which means that the compressed data is in the format used by GZIP and PKZIP. If the parameter is `false`, the data is decompressed in the ZLIB format.

Public Instance Methods

end

public synchronized native void end()

Description

This method discards any unprocessed input data and frees up internal buffers.

finished

public synchronized boolean finished()

Returns

A `boolean` value that indicates whether or not the end of the compressed data has been reached.

Description

This method returns `true` if the last of the compressed data has been read using `inflate()`. Otherwise it returns `false`.

getAdler

public synchronized native int getAdler()

Returns

> The Adler32 checksum value of the uncompressed data.

Description

> This method returns an Adler32 checksum value that is calculated from the uncompressed data returned by `inflate()`.

getRemaining

public synchronized int getRemaining()

Returns

> The number of bytes remaining in the input buffer.

Description

> This method returns the number of bytes that are in the input buffer. It can be called to find out how much data remains after a call to `inflate()`.

getTotalIn

public synchronized native int getTotalIn()

Returns

> The total number of bytes that have been input so far.

Description

> This method returns the number of bytes that have been passed to `setInput()` since this `Inflater` was created or since `reset()` was last called.

getTotalOut

public synchronized native int getTotalOut()

Returns

The total number of bytes that have been output so far.

Description

This method returns the number of bytes that have been read from `inflate()` since this `Inflater` was created or since `reset()` was last called.

inflate

public int inflate(byte[] b) throws DataFormatException

Parameters

`b`

A byte array to be filled.

Returns

The number of decompressed bytes actually written to the array or 0 if more data may be required.

Throws

`DataFormatException`

If the data in the input buffer is not in the correct format.

Description

This method decompresses the data passed to `setInput()` and fills the given array with decompressed data. If this method returns 0, `needsInput()` and `needsDictionary()` should be called in order to determine whether the `Inflater` needs more data in its input buffer or whether it needs a preset dictionary.

public synchronized native int inflate(byte[] b, int off, int len) throws DataFormatException

Parameters

`b`

A byte array to be filled.

`off`

An offset into the byte array.

`len`

The number of bytes to fill.

Returns

The number of decompressed bytes written to the array or 0 if more data may be required.

Throws

`DataFormatException`

If the data in the input buffer is not in the correct format.

Description

This method decompresses the data passed to `setInput()` and writes `len` bytes of the decompressed data into the given array, starting at `off`. If this method returns 0, `needsInput()` and `needsDictionary()` should be called in order to determine whether the `Inflater` needs more data in its input buffer or whether it needs a preset dictionary.

needsDictionary

public synchronized boolean needsDictionary()

Returns

A `boolean` value that indicates whether or not a preset dictionary is needed.

Description

This method returns `true` if a preset dictionary is needed for decompression. Otherwise it returns `false`.

needsInput

public synchronized boolean needsInput()

Returns

A `boolean` value that indicates whether or not the input buffer is empty.

Description

This method returns `true` if the input buffer is empty. Otherwise it returns `false`.

reset

public synchronized native void reset()

Description

This method resets the `Inflater` to the state it was in when it was created, which means that a new set of data can be decompressed.

setDictionary

public void setDictionary(byte[] b)

Parameters

`b`

An array of byte values.

Description

This method sets the preset dictionary for decompression using the data in the given array.

`public synchronized native void setDictionary(byte[] b, int off, int len)`

Parameters

`b`

An array of byte values.

`off`

An offset into the byte array.

`len`

The number of bytes to use.

Description

This method sets the preset dictionary for decompression using `len` bytes from the given array, starting from `off`.

setInput

public void setInput(byte[] b)

Parameters

`b`

An array of byte values.

Description

This method places the contents of the given array into the input buffer of this `Inflater`.

public synchronized void setInput(byte[] b, int off, int len)

Parameters

`b`

An array of byte values.

`off`

An offset into the byte array.

`len`

The number of bytes to use.

Description

This method places `len` bytes from the given array, starting at `off`, into the input buffer of this `Inflater`. Use the `inflate()` method to decompress the data and retrieve it in decompressed form.

Protected Instance Methods

finalize

protected void finalize()

Overrides

`Object.finalize()`

Description

This method calls `end()` when this `Inflater` is garbage collected.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`getClass()`	`Object`	`hashCode()`	`Object`
`notify()`	`Object`	`notifyAll()`	`Object`
`toString()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`Deflater`, `GZIPInputStream`, `InflaterInputStream`, `ZipInputStream`

PREVIOUS GZIPOutputStream
HOME
BOOK INDEX
NEXT InflaterInputStream

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

InflaterInputStream

Name

InflaterInputStream

Synopsis

Class Name:

`java.util.zip.InflaterInputStream`

Superclass:

`java.io.FilterInputStream`

Immediate Subclasses:

`java.util.zip.GZIPInputStream`, `java.util.zip.ZipInputStream`

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `InflaterInputStream` class represents an `InputStream` with an associated `Inflater`. In other words, an `InflaterInputStream` wraps a regular input stream, so that data read from the stream is read from an underlying stream and decompressed. Two subclasses, `GZIPInputStream` and `ZipInputStream`, read compressed data in widely recognized formats.

Class Summary

```
public class java.util.zip.InflaterInputStream
             extends java.io.FilterInputStream {
  // Variables
  protected byte[] buf;
  protected Inflater inf;
  protected int len;
  // Constructors
  public InflaterInputStream(InputStream in);
  public InflaterInputStream(InputStream in, Inflater inf);
  public InflaterInputStream(InputStream in, Inflater inf, int size);
  // Public Instance Methods
  public int read();
  public int read(byte[] b, int off, int len);
  public long skip(long n);
  // Protected Instance Methods
  protected void fill();
}
```

Variables

buf

protected byte[] buf

Description

> A buffer that holds the compressed data that is written to the underlying stream.

inf

protected Inflater inf

Description

The `Inflater` that is used internally.

len

protected int len

Description

The amount of data that is in the input buffer.

Constructors

InflaterInputStream

public InflaterInputStream(InputStream in)

Parameters

`in`

The underlying input stream.

Description

This constructor creates an `InflaterInputStream` that reads data from the given `InputStream`. Before being read, the data is decompressed by a default `Inflater`. The `InflaterInputStream` uses a decompression buffer with the default size of 512 bytes.

public InflaterInputStream(InputStream in, Inflater inf)

Parameters

`in`

The underlying input stream.

`inf`

The `Inflater` object.

Description

This constructor creates an `InflaterInputStream` that reads data from the given `InputStream`. Before being read, the data is decompressed by the given `Inflater`. The `InflaterInputStream` uses a decompression buffer with the default size of 512 bytes.

public InflaterInputStream(InputStream in, Inflater inf, int size)

Parameters

`in`

The underlying input stream.

`inf`

The `Inflater` object.

`size`

The size of the input buffer.

Description

This constructor creates an `InflaterInputStream` that reads data from the given `InputStream`. Before being read, the data is decompressed by the given `Inflater`. The `InflaterInputStream` uses a decompression buffer of the given size.

Instance Methods

read

public int read() throws IOException

Returns

The next uncompressed byte or `-1` if the end of the stream is encountered.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterInputStream.read()`

Description

This method reads enough data from the underlying `InputStream` to return a byte of uncompressed data. The method blocks until enough data is available for decompression, the end of the stream is detected, or an exception is thrown.

public int read(byte[] b, int off, int len) throws IOException

Parameters

`b`

An array of bytes to be filled from the stream.

`off`

An offset into the byte array.

`len`

The number of bytes to read.

Returns

The number of bytes read or `-1` if the end of the stream is encountered immediately.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterInputStream.read(byte[], int, int)`

Description

This method reads enough data from the underlying `InputStream` to return `len` bytes of uncompressed data. The uncompressed data is placed into the given array starting at `off`. The method blocks until some data is available for decompression.

skip

public long skip(long n) throws IOException

Returns

The actual number of bytes skipped.

Throws

`IOException`

If any kind of I/O error occurs.

Overrides

`FilterInputStream.skip()`

Description

This method skips over the specified number of uncompressed data bytes by reading data from the underlying `InputStream` and decompressing it.

Protected Instance Methods

fill

protected void fill() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method fills the input buffer with compressed data from the underlying `InputStream`.

Inherited Methods

Method	Inherited From	Method	Inherited From
`available()`	`FilterInputStream`	`clone()`	`Object`
`close()`	`FilterInputStream`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`mark(int)`	`FilterInputStream`
`markSupported()`	`FilterInputStream`	`notify()`	`Object`
`notifyAll()`	`Object`	`read()`	`InflaterInputStream`
`read(byte[])`	`FilterInputStream`	`reset()`	`FilterInputStream`
`toString()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`FilterInputStream`, `GZIPInputStream`, `Inflater`, `InputStream`, `IOException`, `ZipInputStream`

ZipEntry

Name

ZipEntry

Synopsis

Class Name:

```
java.util.zip.ZipEntry
```

Superclass:

```
java.lang.Object
```

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `ZipEntry` class represents a single entry in a ZIP file, which is either a compressed file or an uncompressed file. `ZipEntry` provides methods that set and retrieve various pieces of information about an entry.

When you are reading a ZIP file, you use `ZipInputStream.getNextEntry()` to return the next entry in the file. Then you can retrieve information about that particular entry. You can also read the entries in a ZIP file in a nonsequential order using the `ZipFile` class.

When you are writing a ZIP file, you use `ZipOutputStream.putNextEntry()` to write an entry, and you must create your own `ZipEntry` objects. If you are storing compressed (deflated) files, you need only specify a name for the `ZipEntry` ; the other fields are filled in automatically. If, however, you are placing uncompressed entries, you need to specify the size of each entry and provide a CRC-32 checksum value.

Class Summary

```
public class java.util.zip.ZipEntry extends java.lang.Object {
  // Constants
  public static final int DEFLATED;
  public static final int STORED;
  // Constructors
  public ZipEntry(String name);
  // Instance Methods
  public String getComment();
  public long getCompressedSize();
  public long getCrc();
  public byte[] getExtra();
  public int getMethod();
  public String getName();
  public long getSize();
  public long getTime();
  public boolean isDirectory();
  public void setComment(String comment);
  public void setCrc(long crc);
  public void setExtra(byte[] extra);
  public void setMethod(int method);
  public void setSize(long size);
  public void setTime(long time);
  public String toString();
}
```

Constants

DEFLATED

public static final int DEFLATED

Description

A constant that represents an entry that is stored using the deflate algorithm.

STORED

public static final int STORED

Description

A constant that represents an entry that is stored verbatim; in other words, with no compression applied.

Constructors

ZipEntry

public ZipEntry(String name)

Parameters

`name`

The name of the entry.

Throws

`NullPointerException`

If `name` is `null`.

`IllegalArgumentException`

If `name` is longer than `0xFFFF` bytes.

Description

This constructor creates a `ZipEntry` with the given name.

Instance Methods

getComment

public String getComment()

Returns

The comment of this entry or `null` if one has not been specified.

Description

This method returns the comment string for this `ZipEntry`.

getCompressedSize

public long getCompressedSize()

Returns

The compressed size of this entry or `-1` is the compressed size is not known.

Description

This method returns the compressed size of this `ZipEntry`.

getCrc

public long getCrc()

Returns

The checksum value for this entry.

Description

This method returns the CRC-32 checksum value for this `ZipEntry`.

getExtra

public byte[] getExtra()

Returns

The extra field data for this entry or `null` if there is none.

Description

This method returns the bytes in the extra field data for this `ZipEntry`.

getMethod

public int getMethod()

Returns

The compression method of this entry or `-1` if it has not been specified.

Description

This method returns the compression method of this `ZipEntry`. Valid values are `DEFLATED` and `STORED`.

getName

public String getName()

Returns

The name of this entry.

Description

This method returns the string name of this `ZipEntry`.

getSize

public long getSize()

Returns

The uncompressed size of this entry or `-1` if the uncompressed size is not known.

Description

This method returns the uncompressed size of this `ZipEntry`.

getTime

public long getTime()

Returns

The modification date of this entry.

Description

This method returns the modification date of this `ZipEntry` as the number of milliseconds since midnight, January 1, 1970 GMT.

isDirectory

public boolean isDirectory()

Returns

A `boolean` value that indicates whether or not this entry is a directory.

Description

This method returns `true` if this `ZipEntry` represents a directory.

setComment

public void setComment(String comment)

Parameters

`comment`

The new comment string.

Throws

`IllegalArgumentException`

If `comment` is longer than `0xFFFF` bytes.

Description

This method sets the comment string of this `ZipEntry`.

setCrc

public void setCrc(long crc)

Parameters

`crc`

The new checksum value.

Description

This method sets the CRC-32 checksum value for this `ZipEntry`.

setExtra

public void setExtra(byte[] extra)

Parameters

`extra`

The extra field data.

Throws

`IllegalArgumentException`

If `extra` is longer than `0xFFFF` bytes.

Description

This method sets the extra field data for this `ZipEntry`.

setMethod

public void setMethod(int method)

Parameters

`method`

The new compression method.

Throws

`IllegalArgumentException`

If `method` is not `DEFLATED` or `STORED`.

Description

This method sets the compression method of this `ZipEntry`. This corresponds to the compression level of `Deflater`.

setSize

public void setSize(long size)

Parameters

`size`

The new uncompressed entry size.

Throws

`IllegalArgumentException`

If `size` is less than 0 or greater than `0xFFFFFFFF` bytes.

Description

This method sets the uncompressed size of this `ZipEntry`.

setTime

public void setTime(long time)

Parameters

`time`

The new modification date, expressed as the number of seconds since midnight, January 1, 1970 GMT.

Description

This method sets the modification date of this `ZipEntry`.

toString

public String toString()

Returns

A string representation of this object.

Overrides

`Object.toString()`

Description

This method returns the name of this `ZipEntry`.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

`Deflater`, `IllegalArgumentException`, `Inflater`, `NullPointerException`, `ZipInputStream`, `ZipOutputStream`

PREVIOUS
InflaterInputStream

HOME
BOOK INDEX

NEXT
ZipException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

ZipException

Name

ZipException

Synopsis

Class Name:

`java.util.zip.ZipException`

Superclass:

`java.io.IOException`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

A `ZipException` is thrown when an error occurs when reading or writing a ZIP file. Normally this occurs when data is read that is not in the correct format.

Class Summary

```
public class java.util.ZipException extends java.io.IOException {
  // Constructors
  public ZipException();
  public ZipException(String s);
}
```

Constructors

ZipException

protected ZipException()

Description

This constructor creates a `ZipException` with no associated detail message.

public ZipException(String s)

Parameters

`s`

The detail message.

Description

This constructor creates a `ZipException` with the given detail message.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`fillInStackTrace()`	`Throwable`	`finalize()`	`Object`
`getClass()`	`Object`	`getLocalizedMessage()`	`Throwable`
`getMessage()`	`Throwable`	`hashCode()`	`Object`

`notify()`	`Object`	`notifyAll()`	`Object`
`printStackTrace()`	`Throwable`	`printStackTrace(PrintStream)`	`Throwable`
`printStackTrace(PrintWriter)`	`Throwable`	`toString()`	`Throwable`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Exception`, `IOException`, `RuntimeException`, `Throwable`

PREVIOUS HOME NEXT
ZipEntry BOOK INDEX ZipFile

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

ZipFile

Name

ZipFile

Synopsis

Class Name:

`java.util.zip.ZipFile`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `ZipFile` class represents a ZIP file. Unlike with a `ZipInputStream`, you can read the entries in a `ZipFile` nonsequentially. Internally, the class uses a `RandomAccessFile` so that you can read the entries from the file in any order. You can obtain a list of the entries in this ZIP file by calling `entries()`. Given an entry, you can get an `InputStream` for that entry using `getInputStream()`.

Class Summary

```
public class java.util.zip.ZipFile extends java.lang.Object {
  // Constructors
  public ZipFile(File file);
  public ZipFile(String name);
  // Instance Methods
  public void close();
  public Enumeration entries();
  public ZipEntry getEntry(String name);
  public InputStream getInputStream(ZipEntry ze);
  public String getName();
}
```

Constructors

ZipFile

public ZipFile(File file) throws ZipException, IOException

Parameters

`file`

The `File` to read.

Throws

`ZipException`

If the ZIP file cannot be read.

`IOException`

If any other kind of I/O error occurs.

Description

This constructor creates a `ZipFile` for reading from the given `File` object.

public ZipFile(String name) throws IOException

Parameters

`name`

A string that contains the path name of the file.

Throws

`ZipException`

If the ZIP file cannot be read.

`IOException`

If any other kind of I/O error occurs.

Description

This constructor creates a `ZipFile` for reading from the file specified by the given path.

Instance Methods

close

public void close() throws IOException

Throws

`IOException`

If any kind of I/O error occurs.

Description

This method closes the `ZipFile` and releases its system resources.

entries

public Enumeration entries()

Returns

An `Enumeration` of `ZipEntry` objects.

Description

This method returns an enumeration of `ZipEntry` objects that represents the contents of this `ZipFile`.

getEntry

public ZipEntry getEntry(String name)

Parameters

`name`

The entry name.

Returns

The entry corresponding to the given name or `null` if there is no such entry.

Description

This method returns the `ZipEntry` object that corresponds to the given entry name.

getInputStream

public InputStream getInputStream(ZipEntry ze) throws IOException

Parameters

`ze`

A `ZipEntry` in this file.

Returns

An `InputStream` for the given entry.

Throws

`ZipException`

If a ZIP file format error occurs.

`IOException`

If any other kind of I/O error occurs.

Description

This method returns an input stream that can read the entry described by the supplied `ZipEntry`.

getName

public String getName()

Returns

The path of this file.

Description

This method returns the path name of this `ZipFile`.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Enumeration`, `File`, `InputStream`, `IOException`, `RandomAccessFile`, `String`, `ZipEntry`, `ZipException`, `ZipInputStream`

PREVIOUS HOME NEXT
ZipException BOOK INDEX ZipInputStream

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

PREVIOUS

NEXT

ZipInputStream

Name

ZipInputStream

Synopsis

Class Name:

`java.util.zip.ZipInputStream`

Superclass:

`java.util.zip.InflaterInputStream`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The ZipInputStream class reads files that have been compressed using the ZIP format. To read uncompressed data from a ZIP file, simply construct a ZipInputStream that wraps a regular input stream. The getNextEntry() method returns each entry in the ZIP file in order. Once you have a ZipEntry object, you use the read() method to retrieve uncompressed data from it. If you want to read the entries in a nonsequential order, use a ZipFile instead.

Class Summary

```
public class java.util.zip.ZipInputStream
             extends java.util.zip.InflaterInputStream {
  // Constructors
  public ZipInputStream(InputStream in);
  // Instance Methods
  public void close();
  public void closeEntry();
  public ZipEntry getNextEntry();
  public int read(byte[] b, int off, int len);
  public long skip(long n);
}
```

Constructors

ZipInputStream

public ZipInputStream(InputStream in)

Parameters

in

The underlying input stream.

Description

This constructor creates a ZipInputStream that inflates data from the given InputStream.

Instance Methods

close

public void close() throws IOException

Throws

`IOException`

If any I/O error occurs.

Overrides

`FilterInputStream.close()`

Description

This method closes this stream and releases any system resources that are associated with it.

closeEntry

public void closeEntry() throws IOException

Throws

`ZipException`

If a ZIP file format error occurs.

`IOException`

If any other kind of I/O error occurs.

Description

This method closes the currently open entry in the ZIP file. The stream is then positioned to read the next entry using `getNextEntry()`.

getNextEntry

public ZipEntry getNextEntry() throws IOException

Returns

> The `ZipEntry` for the next entry or `null` if there are no more entries.

Throws

> `ZipException`
>
> > If a ZIP file format error occurs.
>
> `IOException`
>
> > If any other kind of I/O error occurs.

Description

> This method returns a `ZipEntry` that represents the next entry in the ZIP file and positions the stream to read that entry.

read

public int read(byte[] b, int off, int len) throws IOException

Parameters

> `b`
>
> > An array of bytes to be filled from the stream.
>
> `off`
>
> > An offset into the byte array.
>
> `len`
>
> > The number of bytes to read.

Returns

The number of bytes read or `-1` if the end of the entry is encountered immediately.

Throws

`ZipException`

If a ZIP file format error occurs.

`IOException`

If any other kind of I/O error occurs.

Overrides

`InflaterInputStream.read(byte[], int, int)`

Description

This method reads enough data from the underlying `InputStream` to return `len` bytes of uncompressed data. The uncompressed data is placed into the given array starting at `off`. The method blocks until some data is available for decompression.

skip

public long skip(long n) throws IOException

Returns

The actual number of bytes skipped.

Throws

`ZipException`

If a ZIP file format error occurs.

`IOException`

If any kind of I/O error occurs.

Overrides

`InflaterInputStream.skip()`

Description

This method skips over the specified number of uncompressed data bytes by reading data from the underlying `InputStream` and decompressing it.

Inherited Methods

Method	Inherited From	Method	Inherited From
`available()`	`FilterInputStream`	`clone()`	`Object`
`equals(Object)`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`hashCode()`	`Object`
`mark(int)`	`FilterInputStream`	`markSupported()`	`FilterInputStream`
`notify()`	`Object`	`notifyAll()`	`Object`
`read()`	`InflaterInputStream`	`read(byte[])`	`FilterInputStream`
`reset()`	`FilterInputStream`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

`Inflater`, `InflaterInputStream`, `InputStream`, `IOException`, `ZipEntry`, `ZipException`, `ZipFile`

ZipOutputStream

Name

ZipOutputStream

Synopsis

Class Name:

`java.util.zip.ZipOutputStream`

Superclass:

`java.util.zip.DeflaterOutputStream`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `ZipOutputStream` class writes compressed files in the ZIP format. To write a ZIP file, construct a `ZipOutputStream` that wraps a regular output stream. You have to create a `ZipEntry` for each entry in the file. The `putNextEntry()` method puts the entry in the file, so that you can then use the `write()` method to write data for that entry. When you finish writing the data for an entry, call `closeEntry()` to close that entry and `putNextEntry()` to start another entry.

The `setMethod()` method specifies whether the data is compressed or uncompressed by default; `setLevel()` specifies the level of compression that is used by default. These values can be overridden by the method and level set for a particular entry. If you are storing compressed (deflated) files, you need only specify a name for each `ZipEntry` ; the other fields are filled in automatically. If, however, you are placing uncompressed entries, you need to specify the size of each entry and provide a CRC-32 checksum value.

Class Summary

```
public class java.util.zip.ZipOutputStream
             extends java.util.zip.DeflaterOutputStream {
  // Constants
  public static final int DEFLATED;
  public static final int STORED;
  // Constructors
  public ZipOutputStream(OutputStream out);
  // Instance Methods
  public void close();
  public void closeEntry();
  public void finish();
  public void putNextEntry(ZipEntry e);
  public void setComment(String comment);
  public void setLevel(int level);
  public void setMethod(int method);
  public synchronized void write(byte[] b, int off, int len);
}
```

Constants

DEFLATED

public static final int DEFLATED

Description

> A constant that represents an entry is stored using the deflate algorithm.

STORED

public static final int STORED

Description

A constant that represents a ZIP file entry is stored verbatim; in other words, with no compression applied.

Constructors

ZipOutputStream

public ZipOutputStream(OutputStream out)

Parameters

`out`

The underlying output stream.

Description

This constructor creates a `ZipOutputStream` that writes compressed data to the given `OutputStream`.

Instance Methods

close

public void close() throws IOException

Throws

`ZipException`

If a ZIP file format error occurs.

`IOException`

If any other kind of I/O error occurs.

Overrides

`DeflaterOutputStream.close()`

Description

This method closes the stream and releases any system resources that are associated with it.

closeEntry

public void closeEntry() throws IOException

Throws

`ZipException`

If a ZIP file format error occurs.

`IOException`

If any other kind of I/O error occurs.

Description

This method closes the currently open entry in the ZIP file. A subsequent entry can be started with `putNextEntry()`.

finish

public void finish() throws IOException

Throws

`ZipException`

If a ZIP file format error occurs.

`IOException`

If any other kind of I/O error occurs.

Overrides

`DeflaterOutputStream.finish()`

Description

This method finishes writing compressed data to the underlying stream without closing it.

putNextEntry

public void putNextEntry(ZipEntry e) throws IOException

Parameters

`e`

The new entry.

Throws

`ZipException`

If a ZIP file format error occurs.

`IOException`

If any other kind of I/O error occurs.

Description

This method writes the information in the given `ZipEntry` to the stream and positions the stream for the entry data. The actual entry data can then be written to the stream using `write()`. The default compression method and level are used if one is not specified for the entry. When all of the entry data has been written, use `closeEntry()` to finish the entry.

If this method is called when there is already an open entry, that entry is closed and this entry is opened.

setComment

public void setComment(String comment)

Parameters

`comment`

The new comment string.

Throws

`IllegalArgumentException`

If `comment` is longer than `0xFFFF` bytes.

Description

This method sets the comment string for this ZIP file.

setLevel

public void setLevel(int level)

Parameters

`level`

A compression level, from 0 (`NO_COMPRESSION`) to 9 (`BEST_COMPRESSION`).

Throws

`IllegalArgumentException`

If `level` is not valid.

Description

This method sets the default compression level of subsequent `DEFLATED` entries. The default value is `Deflater.DEFAULT_COMPRESSION`.

setMethod

public void setMethod(int method)

Parameters

`method`

A compression method, either `DEFLATED` or `STORED`.

Throws

`IllegalArgumentException`

If `method` is not valid.

Description

This method sets the default compression method of this `ZipOutputStream` for entries that do not specify a method.

write

`public synchronized void write(byte[] b, int off, int len) throws IOException`

Parameters

`b`

An array of bytes to write to the stream.

`off`

An offset into the byte array.

`len`

The number of bytes to write.

Throws

`ZipException`

If a ZIP file format error occurs.

`IOException`

If any other kind of I/O error occurs.

Overrides

`DeflaterOutputStream.write(byte[], int, int)`

Description

This method takes `len` bytes from the given buffer, starting at `off`, and compresses them. The method then writes the compressed data to the underlying `OutputStream` for the current entry. The method blocks until all the bytes have been written.

Inherited Methods

Method	**Inherited From**	**Method**	**Inherited From**
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`flush()`	`FilterOutputStream`
`getClass()`	`Object`	`hashCode()`	`Object`

`notify()`	`Object`	`notifyAll()`	`Object`
`toString()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`
`write(int)`	`DeflaterOutputStream`	`write(byte[])`	`FilterOutputStream`

See Also

`Deflater`, `DeflaterOutputStream`, `IllegalArgumentException`, `IOException`, `OutputStream`, `ZipEntry`, `ZipException`

PREVIOUS HOME NEXT
ZipInputStream BOOK INDEX The Unicode 2.0 Character Set

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

What This Book Covers

The *Java AWT Reference* is the definitive resource for programmers working with AWT. It covers all aspects of the AWT package, in versions 1.0.2 and 1.1. If there are any changes to AWT after 1.1 (at least two patch releases are expected), we will integrate them as soon as possible. Watch the book's Web site http://www.ora.com/catalog/javawt/ for details on changes.

Specifically, this book completely covers the following packages:

`java.awt` (1.0 and 1.1)
`java.awt.image` (1.0 and 1.1)
`java.awt.event` (new to 1.1)
`java.awt.datatransfer` (new to 1.1)
`java.awt.peer` (1.0 and 1.1)
`java.applet` (1.0 and 1.1)

The book also covers some aspects of the `sun.awt` package (some interesting and useful layout managers) and the `sun.audio` package (some more flexible ways of working with audio files). It also gives a brief overview of the behind-the-scenes machinery for rendering images, much of which is in the `sun.awt.image` package.

Organization

The *Java AWT Reference* is divided into two large parts. The first part is a thorough guide to using AWT. Although this guide is organized by class, it was designed to flow logically, rather than alphabetically. I know that few people read a book like this from beginning to end, but if you want to, it's possible. With a few exceptions, you should be able to read the early chapters without knowing the material that's covered in the later chapters. You'll want to read this section to find out how any chunk of the AWT package works in detail.

The second part is a set of documentation pages typical of what you find in most reference sets. It is organized alphabetically by package, and within each package, alphabetically by class. It is designed to answer questions like "What are the arguments to the `FilteredImageSource` constructor?" The reference section provides brief summaries, rather than detailed discussions and examples. When you use a typical reference book, you're usually trying to look up some detail, rather than learn how something works from scratch.

In other words, this book provides two views of AWT: terse summaries designed to help you when you need to look something up quickly, and much more detailed explanations designed to help you understand how to use AWT to the fullest. In doing so, it goes well beyond the standard reference manual. A reference manual alone gives you a great view of hundreds of individual trees; this book gives you the trees, but also gives you the forest that allows you to put the individual pieces in context. There are dozens of complete examples, together with background information, overview material, and other information that doesn't fit into the standard reference manual format.

About the Source Code

The source code for the programs presented in this book is available online. See http://www.ora.com/catalog/javawt/ for downloading instructions.

Obtaining the Example Programs

The example programs in this book are available electronically in a number of ways: by FTP, Ftpmail, BITFTP, and UUCP. The cheapest, fastest, and easiest ways are listed first. If you read from the top down, the first one that works for you is probably the best. Use FTP if you are directly on the Internet. Use Ftpmail if you are not on the Internet but can send and receive electronic mail to Internet sites (this includes CompuServe users). Use BITFTP if you send electronic mail via BITNET. Use UUCP if none of the above works.

FTP

To use FTP, you need a machine with direct access to the Internet. A sample session is shown, with what you should type in `boldface`.

```
% ftp ftp.ora.com
Connected to ftp.ora.com.
220 FTP server (Version 6.21 Tue Mar 10 22:09:55 EST 1992) ready.
Name (ftp.ora.com:yourname): anonymous
331 Guest login ok, send domain style e-mail address as password.
Password: yourname@yourhost.com (use your user name and host here)
230 Guest login ok, access restrictions apply.
ftp> cd /published/oreilly/java/awt
250 CWD command successful.
ftp> binary (Very important! You must specify binary transfer for compressed files.)
200 Type set to I.
ftp> get examples.tar.gz
200 PORT command successful.
150 Opening BINARY mode data connection for examples.tar.gz.
226 Transfer complete.
ftp> quit
221 Goodbye.
%
```

The file is a compressed *tar* archive; extract the files from the archive by typing:

```
% zcat examples.tar.gz | tar xvf -
```

System V systems require the following *tar* command instead:

```
% zcat examples.tar.gz | tar xof -
```

If *zcat* is not available on your system, use separate *gunzip* and *tar* commands.

```
% gunzip examples.tar.gz
% tar xvf examples.tar
```

Ftpmail

Ftpmail is a mail server available to anyone who can send electronic mail to, and receive it from, Internet sites. This includes any company or service provider that allows email connections to the Internet. Here's how you do it.

You send mail to *ftpmail@online.ora.com*. (Be sure to address the message to *ftpmail* and not to *ftp*.) In the message body, give the FTP commands you want to run. The server will run anonymous FTP for you and mail the files back to you. To get a complete help file, send a message with no subject and the single word "help" in the body. The following is a sample mail session that should get you the examples. This command sends you a listing of the files in the selected directory and the requested example files. The listing is useful if there's a later version of the examples you're interested in.

```
% mail ftpmail@online.ora.com
Subject:
reply-to yourname@yourhost.com    Where you want files mailed
open
cd /published/oreilly/java/awt
dir
mode binary
uuencode
get examples.tar.gz
quit
.
```

A signature at the end of the message is acceptable as long as it appears after "quit."

BITFTP

BITFTP is a mail server for BITNET users. You send it electronic mail messages requesting files, and it sends you back the files by electronic mail. BITFTP currently serves only users who send it mail from nodes that are directly on BITNET, EARN, or NetNorth. BITFTP is a public service of Princeton University. Here's how it works.

To use BITFTP, send mail containing your FTP commands to *BITFTP@PUCC*. For a complete help file, send HELP as the message body.

The following is the message body you send to BITFTP:

```
FTP  ftp.uu.net  NETDATA
USER  anonymous
PASS  yourname@yourhost.edu Put your Internet email address here (not your BITNET
address)
```

```
CD  /published/oreilly/java/awt
DIR
BINARY
GET  examples.tar.gz
QUIT
```

Once you've got the desired file, follow the directions under FTP to extract the files from the archive. Since you are probably not on a UNIX system, you may need to get versions of *uudecode*, *uncompress*, *atob*, and *tar* for your system. VMS, DOS, and Mac versions are available. The VMS versions are on *gatekeeper.dec.com* in */pub/VMS*.

UUCP

UUCP is standard on virtually all UNIX systems and is available for IBM-compatible PCs and Apple Macintoshes. The examples are available by UUCP via modem from UUNET; UUNET's connect-time charges apply.

If you or your company has an account with UUNET, you have a system somewhere with a direct UUCP connection to UUNET. Find that system, and type:

```
uucp uunet\!~/published/oreilly/java/awt/examples.tar.gz yourhost\!~/yourname/
```

The backslashes can be omitted if you use the Bourne shell (*sh*) instead of *csh*. The file should appear some time later (up to a day or more) in the directory */usr/spool/uucppublic/*`yourname`. If you don't have an account, but would like one so that you can get electronic mail, contact UUNET at 703-204-8000.

Once you've got the desired file, follow the directions under FTP to extract the files from the archive.

Other Java Books and Resources

This book is part of a series of Java books from O'Reilly & Associates that covers everything you wanted to know, and then some. The *Java AWT Reference* is paired with the *Java Fundamental Class Reference* to document the entire Core Java API. Other books in the series provide an introduction (*Exploring Java*) and document the virtual machine (*Java Virtual Machine*), the language (*Java Language Reference*), multithreaded programming (*Java Threads*), and network programming (*Java Network Programming*), with more to come. *Java in a Nutshell* is another popular Java book in the Nutshell series from O'Reilly. For a complete up-to-date list of the available Java resources, refer to http://www.ora.com/info/java/.

In addition to the resources from O'Reilly, Sun's online documentation on Java is maintained at http://www.javasoft.com/nav/download/index.html. Information on specific Java-capable browsers can be found at their respective Web sites, which are listed in Table 0.1. More are sure to be on the way. (Some browsers are platform specific, while others are multi-platform.)

Table 0.1: Popular Web Browsers that Support Java

Browser	**Location**
Netscape Navigator	http://home.netscape.com/comprod/products/navigator/
Microsoft's Internet Explorer	http://www.microsoft.com/ie
Sun's HotJava	http://www.javasoft.com/HotJava/
Oracle's PowerBrowser	http://www.oracle.com/products/websystem/powerbrowser
Apple's Cyberdog	http://cyberdog.apple.com/

Newsgroups also serve as a discussion area for Java-related topics. The *comp.lang.java* group has formally split into several others. The new groups are:

comp.lang.java.advocacy
comp.lang.java.announce
comp.lang.java.beans
comp.lang.java.databases
comp.lang.java.gui
comp.lang.java.help
comp.lang.java.machine
comp.lang.java.programmer
comp.lang.java.security
comp.lang.java.setup
comp.lang.java.softwaretools
comp.lang.java.tech

For folks without time to dig through all the noise, *Digital Espresso* provides a periodic digest of the newsfeed at http://www.io.org./~mentor/DigitalEspresso.html. A list of Java FAQs is at http://www-net.com/java/faq/; one of the most interesting is *Cafe Au Lait*, at http://sunsite.unc.edu/javafaq/. (*Cafe Au Lait* is written by Elliotte Rusty Harold, author of *Java Network Programming*.)

Local Java user groups are another good resource. (Having founded one myself, I'm biased.) What they offer varies greatly, but unless you look at one, you are potentially leaving out a vast resource for knowledge and experience. Lists of area user groups are available from JavaSoft at http://www.javasoft.com/Mail/usrgrp.html; also check out the Sun User Group's Special Interest Group for Users of Java at http://www.sug.org/Java/groups.html. In addition to the usual monthly meetings and forums, some maintain a mailing list for technical exchanges.

Security is a major issue with Java. If you are interested in reading more about Java security issues, Princeton University's Safe Internet Programming Web site at http://www.cs.princeton.edu/sip/News.html is an excellent resource.

1.3 Layouts

Layouts allow you to format components on the screen in a platform-independent way. Without layouts, you would be forced to place components at explicit locations on the screen, creating obvious problems for programs that need to run on multiple platforms. There's no guarantee that a `TextArea` or a `Scrollbar` or any other component will be the same size on each platform; in fact, you can bet they won't be. In an effort to make your Java creations portable across multiple platforms, Sun created a `LayoutManager` interface that defines methods to reformat the screen based on the current layout and component sizes. Layout managers try to give programs a consistent and reasonable appearance, regardless of the platform, the screen size, or actions the user might take.

The standard JDK provides five classes that implement the `LayoutManager` interface. They are `FlowLayout`, `GridLayout`, `BorderLayout`, `CardLayout`, and `GridBagLayout`. All of these layouts are covered in much greater detail in Chapter 7, *Layouts*. This chapter also discusses how to create complex layouts by combining layout managers and how to write your own `LayoutManager`. The Java 1.1 JDK includes the `LayoutManager2` interface. This interface extends the `LayoutManager` interface for managers that provide constraint-based layouts.

FlowLayout

The `FlowLayout` is the default layout for the `Panel` class, which includes its most famous subclass, `Applet`. When you add components to the screen, they flow left to right (centered within the applet) based upon the order added and the width of the applet. When there are too many components to fit, they "wrap" to a new row, similar to a word processor with word wrap enabled. If you resize an applet, the components' flow will change based upon the new width and height. Figure 1.11 shows an example both before and after resizing. FlowLayout contains all the `FlowLayout` details.

Figure 1.11: A FlowLayout before and after resizing

[Graphic: Figure 1-11]

GridLayout

The GridLayout is widely used for arranging components in rows and columns. As with FlowLayout, the order in which you add components is relevant. You start at row one, column one, move across the row until it's full, then continue on to the next row. However, unlike FlowLayout, the underlying components are resized to fill the row-column area, if possible. GridLayout can reposition or resize objects after adding or removing components. Whenever the area is resized, the components within it are resized. Figure 1.12 shows an example before and after resizing. GridLayout contains all the details about GridLayout.

Figure 1.12: A GridLayout before and after resizing

[Graphic: Figure 1-12]

BorderLayout

BorderLayout is one of the more unusual layouts provided. It is the default layout for Window, along with its children, Frame and Dialog. BorderLayout provides five areas to hold components. These areas are named after the four different borders of the screen, North, South, East, and West, with any remaining space going into the Center area. When you add a component to the layout, you must specify which area to place it in. The order in which components are added to the screen is not important, although you can have only one component in each area. Figure 1.13 shows a `BorderLayout` that has one button in each area, before and after resizing. BorderLayout covers the details of the `BorderLayout`.

Figure 1.13: A BorderLayout

[Graphic: Figure 1-13]

CardLayout

The `CardLayout` is a bit on the strange side. A `CardLayout` usually manages several components, displaying one of them at a time and hiding the rest. All the components are given the same size. Usually, the `CardLayout` manages a group of `Panels` (or some other container), and each `Panel` contains several components of its own. With a little work, you can use the `CardLayout` to create tabbed dialog boxes or property sheets, which are not currently part of AWT. `CardLayout` lets you assign names to the components it is managing and lets you jump to a component by name. You can also cycle through components in order. Figure 1.11, Figure 1.12, and Figure 1.13 show multiple cards controlled by a single `CardLayout`. Selecting the `Choice` button displays a different card. CardLayout discusses the details of `CardLayout`.

GridBagLayout

`GridBagLayout` is the most sophisticated and complex of the layouts provided in the development kit. With the `GridBagLayout`, you can organize components in multiple rows and columns, stretch specific rows or columns when space is available, and anchor objects in different corners. You provide all the details of each component through instances of the `GridBagConstraints` class. Figure 1.14 shows an example of a `GridBagLayout`. `GridBagLayout` and `GridBagConstraints` are discussed in GridBagLayout and GridBagConstraints.

Figure 1.14: A GridBagLayout

[Graphic: Figure 1-14]

PREVIOUS | HOME | NEXT
Peers | BOOK INDEX | Containers

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

1.4 Containers

A Container is a type of component that provides a rectangular area within which other components can be organized by a LayoutManager. Because Container is a subclass of Component, a Container can go inside another Container, which can go inside another Container, and so on, like Russian nesting dolls. Subclassing Container allows you to encapsulate code for the components within it. This allows you to create reusable higher-level objects easily. Figure 1.15 shows the components in a layout built from several nested containers.

Figure 1.15: Components within containers

[Graphic: Figure 1-15]

Panels

A Panel is the basic building block of an applet. It provides a container with no special features. The default layout for a Panel is FlowLayout. The details of Panel are discussed in Panel. Figure 1.16 shows an applet that contains panels within panels within panels.

Figure 1.16: A multilevel panel

[Graphic: Figure 1-16]

Windows

A Window provides a top-level window on the screen, with no borders or menu bar. It provides a way to implement pop-up messages, among other things. The default layout for a Window is BorderLayout. Window explores the Window class in greater detail. Figure 1.17 shows a pop-up message using a Window in Microsoft Windows and Motif.

Figure 1.17: Pop-up windows

[Graphic: Figure 1-17]

Frames

A Frame is a Window with all the window manager's adornments (window title, borders, window minimize/maximize/close functionality) added. It may also include a menu bar. Since Frame subclasses Window, its default layout is BorderLayout. Frame provides the basic building block for screen-oriented applications. Frame allows you to change the mouse cursor, set an icon image, and have menus. All the details of Frame are discussed in Frames. Figure 1.18 shows an example Frame.

Figure 1.18: A frame

[Graphic: Figure 1-18]

Dialog and FileDialog

A `Dialog` is a `Window` that accepts input from the user. `BorderLayout` is the default layout of `Dialog` because it subclasses `Window`. A `Dialog` is a pop-up used for user interaction; it can be modal to prevent the user from doing anything with the application before responding. A `FileDialog` provides a prebuilt `Dialog` box that interacts with the filesystem. It implements the Open/Save dialog provided by the native windowing system. You will primarily use `FileDialog` with applications since there is no guarantee that an applet can interact with the local filesystem. (Netscape Navigator will throw an exception if you try to use it.) The details of `Dialog` are revealed in Dialogs, while `FileDialog` is discussed in FileDialog. Figure 1.19 shows sample `Dialog` and `FileDialog` boxes.

Figure 1.19: Examples of Dialog and FileDialog boxes

[Graphic: Figure 1-19]

ScrollPane

Java 1.1 introduces the `ScrollPane` container. In version 1.0, if you want to have a scrolling area (for example, to display an image that won't fit onto the screen), you create a panel using `BorderLayout` that contains scrollbars on the right and bottom, and display part of the image in the rest of the screen. When the user scrolls, you capture the event, figure out what part of the image to display, and update the screen accordingly. Although this works, its performance is poor, and it's inconvenient. With version 1.1 of Java, you can tell the `ScrollPane` what needs to scroll; it creates the scrollbars and handles all the events automatically. ScrollPane covers the `ScrollPane`; Figure 1.20 shows a `ScrollPane`. Chapter 11, *Scrolling*, covers the `Adjustable` interface that `Scrollbar` implements and `ScrollPane` utilizes.

Figure 1.20: A ScrollPane

[Graphic: Figure 1-20]

PREVIOUS
Layouts

HOME
BOOK INDEX

NEXT
And the Rest

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

1.5 And the Rest

Several of the remaining classes within `java.awt` are important to mention here but did not fit well into a general category. The following sections are a grab bag that summarize the remaining classes.

Drawing and Graphics

Java provides numerous primitives for drawing lines, squares, circles, polygons, and images. Figure 1.21 shows a simple drawing. The drawing components of AWT are discussed in Chapter 2, *Simple Graphics*.

The `Font`, `FontMetrics`, `Color`, and `SystemColor` classes provide the ability to alter the displayed output. With the `Font` class, you adjust how displayed text will appear. With `FontMetrics`, you can find out how large the output will be, for the specific system the user is using. You can use the `Color` class to set the color of text and graphics. `SystemColor` is new to Java 1.1; it lets you take advantage of desktop color schemes. These classes are discussed in Chapter 3, *Fonts and Colors*.

Figure 1.21: A simple drawing

[Graphic: Figure 1-21]

AWT also includes a number of classes that support more complex graphics manipulations: displaying images, generating images in memory, and transforming images. These classes make up the package `java.awt.image`, which is covered in Chapter 12, *Image Processing*.

Events

Like most windows programming environments, AWT is event driven. When an event occurs (for example, the user presses a key or moves the mouse), the environment generates an event and passes it along to a handler to process the event. If nobody wants to handle the event, the system ignores it. Unlike some windowing environments, you do not have to provide a main loop to catch and process all the events, or an infinite busy-wait loop. AWT does all the event management and passing for you.

Probably the most significant difference between versions 1.0.2 and 1.1 of AWT is the way events work. In older versions of Java, an event is distributed to every component that might conceivably be interested in it, until some component declares that it has handled the event. This event model can still be used in 1.1, but there is also a new event model in which objects listen for particular events. This new model is arguably a little more work for the programmer but promises to be much more efficient, because events are distributed only to objects that want to hear about them. It is also how JavaBeans works.

In this book, examples that are using the older (1.0.2) components use the old event model, unless otherwise indicated. Examples using new components use the new event model. Don't let this mislead you; all components in Java 1.1 support the new event model. The details of `Event` for both version 1.0.2 and 1.1 can be found in Chapter 4, *Events*.

Applets

Although it is not a part of the `java.awt` package, the Core Java API provides a framework for applet development. This includes support for getting parameters from HTML files, changing the web page a browser is displaying, and playing audio files. Chapter 14, *And Then There Were Applets*, describes all the details of the `java.applet` package. Because audio support is part of `java.applet`, portable audio playing is limited to applets. Chapter 14, *And Then There Were Applets* also shows a nonportable way to play audio in applications. Additional audio capabilities are coming to the Java Core API in the announced extensions.

Clipboards

In Java 1.1, programs can access the system clipboard. This process makes it easier to transfer (cut, copy, and paste) data between various other sources and your Java programs and introduces developers to the concepts involved with JavaBeans. Chapter 16, *Data Transfer*, describes the `java.awt.datatransfer` package.

Printing

Java 1.1 adds the ability to print. Adding printing to an existing program is fairly simple: you don't have to do much beside adding a Print menu button. Chapter 17, *Printing*, describes these capabilities.

PREVIOUS HOME NEXT
Containers BOOK INDEX Summary

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

5.2 Labels

Having covered the features of the `Component` class, we can now look at some of the simplest components. The first component introduced here is a `Label`. A label is a `Component` that displays a single line of static text.[3] It is useful for putting a title or message next to another component. The text can be centered or justified to the left or right. Labels react to all events they receive. However, they do not get any events from their peers.

> [3] *Java in A Nutshell* (from O'Reilly & Associates) includes a multiline `Label` component.

Label Methods

Constants

There are three alignment specifiers for labels. The alignment tells the `Label` where to position its text within the space allotted. Setting an alignment for a `Label` might not do anything noticeable if the `LayoutManager` being used does not resize the `Label` to give it more space. With `FlowLayout`, the alignment is barely noticeable. See Chapter 7, *Layouts*, for more information.

public final static int LEFT

> `LEFT` is the constant for left alignment. If no alignment is specified in the constructor, left alignment is the default.

public final static int CENTER

> `CENTER` is the constant for center alignment.

public final static int RIGHT

> `RIGHT` is the constant for right alignment.

Constructors

public Label ()

> This constructor creates an empty `Label`. By default, the label's text is left justified.

public Label (String label)

> This constructor creates a `Label` whose initial text is `label`. By default, the label's text is left justified.

public Label (String label, int alignment)

> This constructor creates a `Label` whose initial text is `label`. The alignment of the label is `alignment`. If `alignment` is invalid (not `LEFT`, `RIGHT`, or `CENTER`), the constructor throws the run-time exception `IllegalArgumentException`.

Text

public String getText ()

> The `getText()` method returns the current value of `Label`.

public void setText (String label)

> The `setText()` method changes the text of the `Label` to `label`. If the new label is a different size from the old one, you should revalidate the display to ensure the label's entire contents will be seen.

Alignment

public int getAlignment ()

> The `getAlignment()` method returns the current alignment of the `Label`.

public void setAlignment (int alignment)

> The `setAlignment()` method changes the alignment of the `Label` to `alignment`. If

alignment is invalid (not LEFT, RIGHT, or CENTER), setAlignment() throws the run-time exception IllegalArgumentException. Figure 5.2 shows all three alignments.

Figure 5.2: Labels with different alignments

[Graphic: Figure 5-2]

Miscellaneous methods

public synchronized void addNotify ()

The addNotify() method creates the Label peer. If you override this method, first call super.addNotify(), then put in your customizations. Then you will be able to do everything you need with the information about the newly created peer.

protected String paramString ()

The paramString() method overrides Component's paramString() method. It is a protected method that calls the overridden paramString() to build a String from the different parameters of the Component. When the method paramString() is called for a Label, the alignment and label's text are added. Thus, for the Label created by the constructor new Label (`ZapfDingbats`, Label.RIGHT), the results displayed from a call to toString() would be:

```
java.awt.Label[0,0,0x0,invalid,align=right,label=ZapfDingbats]
```

Label Events

The Label component can react to any event it receives, though the Label peer normally does not send any. However, there is nothing to stop you from posting an event yourself.

PREVIOUS Component

HOME
BOOK INDEX

NEXT Buttons

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

5.3 Buttons

The `Button` component provides one of the most frequently used objects in graphical applications. When the user selects a button, it signals the program that something needs to be done by sending an action event. The program responds in its `handleEvent()` method (for Java 1.0) or its `actionPerformed()` method (defined by Java 1.1's `ActionListener` interface). Next to `Label`, which does nothing, `Button` is the simplest component to understand. Because it is so simple, we will use a lot of buttons in our examples for the next few chapters.

Button Methods

Constructors

public Button ()

> This constructor creates an empty `Button`. You can set the label later with `setLabel()`.

public Button (String label)

> This constructor creates a `Button` whose initial text is `label`.

Button Labels

public String getLabel ()

> The `getLabel()` method retrieves the current text of the label on the `Button` and returns it as a `String`.

public synchronized void setLabel (String label)

> The `setLabel()` method changes the text of the label on the `Button` to `label`. If the new text is a different size from the old, it is necessary to revalidate the screen to ensure that the button size is correct.

Action Commands

With Java 1.1, every button can have two names. One is what the user sees (the button's label); the other is what the programmer sees and is called the button's *action command*. Distinguishing between the label and the action command is a major help to internationalization. The label can be localized for the user's environment. However, this means that labels can vary at run-time and are therefore useless for comparisons within the program. For example, you can't test whether the user pushed the Yes button if that button might read Oui or Ja, depending on some run-time environment setting. To give the programmer something reliable for comparisons, Java 1.1 introduces the action command. The action command for our button might be Yes, regardless of the button's actual label.

By default, the action command is equivalent to the button's label. Java 1.0 code, which only relies on the label, will continue to work. Furthermore, you can continue to write in the Java 1.0 style as long as you're sure that your program will never have to account for other languages. These days, that's a bad bet. Even if you aren't implementing multiple locales now, get in the habit of testing a button's action command rather than its label; you will have less work to do when internationalization does become an issue.

public String getActionCommand () ★

> The `getActionCommand()` method returns the button's current action command. If no action command was explicitly set, this method returns the label.

public void setActionCommand (String command) ★

> The `setActionCommand()` method changes the button's action command to `command`.

Miscellaneous methods

public synchronized void addNotify ()

> The `addNotify()` method creates the `Button` peer. If you override this method, first call `super.addNotify()`, then add your customizations. Then you can do everything you need with the information about the newly created peer.

protected String paramString ()

> The `paramString()` method overrides the component's `paramString()` method. It is a

protected method that calls the overridden `paramString()` to build a `String` from the different parameters of the `Component`. When the method `paramString()` is called for a `Button`, the button's label is added. Thus, for the `Button` created by the constructor `new Button ("ZapfDingbats")`, the results displayed from a call to `toString()` could be:

```
java.awt.Button[77,5,91x21,label=ZapfDingbats]
```

Button Events

With the 1.0 event model, `Button` components generate an `ACTION_EVENT` when the user selects the button.

With the version 1.1 event model, you register an `ActionListener` with the method `addActionListener()`. When the user selects the `Button`, the method `ActionListener.actionPerformed()` is called through the protected `Button.processActionEvent()` method. Key, mouse, and focus listeners are registered through the `Component` methods of `addKeyListener()`, `addMouseListener()`, or `addMouseMotionListener()`, and `addFocusListener()`, respectively. Action

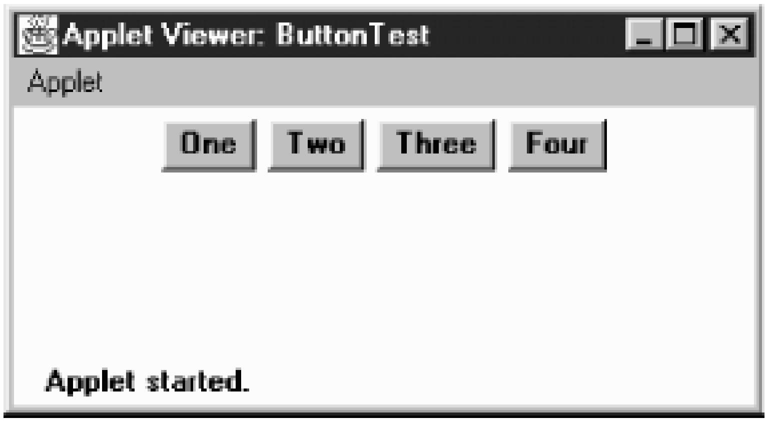
public boolean action (Event e, Object o)

The `action()` method for a `Button` is called when the user presses and releases the button. `e` is the `Event` instance for the specific event, while `o` is the button's label. The default implementation of `action()` does nothing and returns `false`, passing the event to the button's container for processing. For a button to do something useful, you should override either this method or the container's `action()` method. Example 5.1 is a simple applet called `ButtonTest` that demonstrates the first approach; it creates a `Button` subclass called `TheButton`, which overrides `action()`. This simple subclass doesn't do much; it just labels the button and prints a message when the button is pressed. Figure 5.3 shows what `ButtonTest` looks like.

Example 5.1: Button Event Handling

```
import java.awt.*;
import java.applet.*;
class TheButton extends Button {
    TheButton (String s) {
        super (s);
    }
    public boolean action (Event e, Object o) {
        if ("One".equals(o)) {
            System.out.println ("Do something for One");
        } else if ("Two".equals(o)) {
            System.out.println ("Ignore Two");
        } else if ("Three".equals(o)) {
            System.out.println ("Reverse Three");
        } else if ("Four".equals(o)) {
            System.out.println ("Four is the one");
        } else {
            return false;
        }
        return true;
    }
}
public class ButtonTest extends Applet {
   public void init () {
        add (new TheButton ("One"));
        add (new TheButton ("Two"));
        add (new TheButton ("Three"));
        add (new TheButton ("Four"));
   }
}
```

Figure 5.3: The ButtonTest applet



Applet started.

Keyboard

Buttons are able to capture keyboard-related events once the button has the input focus. In order to give a `Button` the input focus without triggering the action event, call `requestFocus()`. The button also gets the focus if the user selects it and drags the mouse off of it without releasing the mouse.

public boolean keyDown (Event e, int key) ☆

The `keyDown()` method is called whenever the user presses a key while the `Button` has the input focus. `e` is the `Event` instance for the specific event, while `key` is the integer representation of the character pressed. The identifier for the event (`e.id`) could be either `Event.KEY_PRESS` for a regular key or `Event.KEY_ACTION` for an action-oriented key (i.e., an arrow or a function key). There is no visible indication that the user has pressed a key over the button.

public boolean keyUp (Event e, int key) ☆

The `keyUp()` method is called whenever the user releases a key while the `Button` has the input focus. `e` is the `Event` instance for the specific event, while `key` is the integer representation of the character pressed. The identifier for the event (`e.id`) could be either `Event.KEY_RELEASE` for a regular key or `Event.KEY_ACTION_RELEASE` for an action-oriented key (i.e., an arrow or a function key). `keyUp()` may be used to determine how long `key` has been pressed.

Listeners and 1.1 event handling

With the 1.1 event model, you register listeners, which are told when the event happens.

public void addActionListener(ActionListener listener) ★

The `addActionListener()` method registers `listener` as an object interested in receiving notifications when an `ActionEvent` passes through the `EventQueue` with this `Button` as its target. The `listener.actionPerformed()` method is called when these events occur. Multiple listeners can be registered. The following code demonstrates how to use an `ActionListener` to handle the events that occur when the user selects a button. This applet has the same display as the previous one, shown in Figure 5.3.

```
// Java 1.1 only
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
public class ButtonTest11 extends Applet implements ActionListener {
    Button b;
    public void init () {
        add (b = new Button ("One"));
        b.addActionListener (this);
        add (b = new Button ("Two"));
        b.addActionListener (this);
        add (b = new Button ("Three"));
        b.addActionListener (this);
        add (b = new Button ("Four"));
        b.addActionListener (this);
    }
    public void actionPerformed (ActionEvent e) {
        String s = e.getActionCommand();
        if ("One".equals(s)) {
            System.out.println ("Do something for One");
        } else if ("Two".equals(s)) {
            System.out.println ("Ignore Two");
        } else if ("Three".equals(s)) {
            System.out.println ("Reverse Three");
        } else if ("Four".equals(s)) {
            System.out.println ("Four is the one");
        }
    }
}
```

public void removeActionListener(ActionListener listener) ★

The `removeActionListener()` method removes `listener` as an interested listener. If `listener` is not registered, nothing happens.

protected void processEvent(AWTEvent e) ★

The `processEvent()` method receives `AWTEvent` with this `Button` as its target. `processEvent()` then passes them along to any listeners for processing. When you subclass `Button`, overriding `processEvent()` allows you to process all events yourself, before sending them to any listeners. In a way, overriding `processEvent()` is like overriding `handleEvent()` using the 1.0 event model.

If you override `processEvent()`, remember to call `super.processEvent(e)` last to ensure that regular event processing can occur. If you want to process your own events, it's a good idea to call `enableEvents()` (inherited from `Component`) to ensure that events are delivered even in the absence of registered listeners.

protected void processActionEvent(ActionEvent e) ★

The `processActionEvent()` method receives `ActionEvent` with this `Button` as its target. `processActionEvent()` then passes them along to any listeners for processing. When you subclass `Button`, overriding `processActionEvent()` allows you to process all action events yourself, before sending them to any listeners. In a way, overriding processActionEvent() is like overriding action() using the 1.0 event model.

If you override the processActionEvent() method, you must remember to call super.processActionEvent(e) last to ensure that regular event processing can occur. If you want to process your own events, it's a good idea to call enableEvents() (inherited from Component) to ensure that events are delivered even in the absence of registered listeners.

5.5 Canvas

A `Canvas` is a class just waiting to be subclassed. Through `Canvas`, you can create additional AWT objects that are not provided by the base classes. `Canvas` is also useful as a drawing area, particularly when additional components are on the screen. It is tempting to draw directly onto a `Container`, but this often isn't a good idea. Anything you draw might disappear underneath the components you add to the container. When you are drawing on a container, you are essentially drawing on the background. The container's layout manager doesn't know anything about what you have drawn and won't arrange components with your artwork in mind. To be safe, do your drawing onto a `Canvas` and place that `Canvas` in a `Container`.

Canvas Methods

Constructors

public Canvas () ★

> The constructor creates a new `Canvas` with no default size. If you place the canvas in a container, the container's layout manager sizes the canvas for you. If you aren't placing the canvas in a container, call `setBounds()` to specify the canvas's size.
>
> Java 1.0 used the default constructor for `Canvas`; there was no explicit constructor.

Miscellaneous methods

public void paint (Graphics g) ★

> The default implementation of the `paint()` method colors the entire `Canvas` with the current background color. When you subclass this method, your `paint()` method needs to draw whatever should be shown on the canvas.

public synchronized void addNotify ()

> The `addNotify()` method creates the `Canvas` peer. If you override this method, first call `super.addNotify()`, then add your customizations. Then you can do everything you need with the information about the newly created peer.

Canvas Events

The `Canvas` peer passes all events to you, which is why it's well suited to creating your own components.

2.2 Point

The `Point` class encapsulates x and y coordinates within a single object. It is probably one of the most underused classes within Java. Although there are numerous places within AWT where you would expect to see a `Point`, its appearances are surprisingly rare. Java 1.1 is starting to use `Point` more heavily. The `Point` class is most often used when a method needs to return a pair of coordinates; it lets the method return both x and y as a single object. Unfortunately, `Point` usually is not used when a method requires x and y coordinates as arguments; for example, you would expect the `Graphics` class to have a version of `translate()` that takes a point as an argument, but there isn't one.

The `Point` class does *not* represent a point on the screen. It is not a visual object; there is no `drawPoint()` method.

Point Methods

Variables

The two public variables of `Point` represent a pair of coordinates. They are accessible directly or use the `getLocation()` method. There is no predefined origin for the coordinate space.

public int x

> The coordinate that represents the horizontal position.

public int y

> The coordinate that represents the vertical position.

Constructors

public Point ()

The first constructor creates an instance of `Point` with an initial x value of 0 and an initial y value of 0.

public Point (int x, int y)

The next constructor creates an instance of `Point` with an initial x value of `x` and an initial y value of `y`.

public Point (Point p)

The last constructor creates an instance of `Point` from another point, the x value of `p.x` and an initial y value of `p.y`.

Locations

public Point getLocation () ★

The `getLocation()` method retrieves the current location of this point as a new `Point`.

public void setLocation (int x, int y) ★
public void move (int x, int y) ☆

The `setLocation()` method changes the point's location to (`x`, `y`).

`move()` is the Java 1.0 name for this method.

public void setLocation (Point p) ★

This `setLocation()` method changes the point's location to (`p.x`, `p.y`).

public void translate (int x, int y)

The `translate()` method moves the point's location by adding the parameters (`x`, `y`) to the corresponding fields of the `Point`. If the original `Point p` is (3, 4) and you call `p.translate(4, -5)`, the new value of `p` is (7, -1).

Miscellaneous methods

public int hashCode ()

The `hashCode()` method returns a hash code for the point. The system calls this method when a `Point` is used as the key for a hash table.

public boolean equals (Object object)

The `equals()` method overrides the `Object.equals()` method to define equality for points. Two `Point` objects are equal if their x and y values are equal.

public String toString ()

The `toString()` method of `Point` displays the current values of the x and y variables. For example:

```
java.awt.Point[x=100,y=200]
```

2.3 Dimension

The `Dimension` class is similar to the `Point` class, except it encapsulates a width and height in a single object. Like `Point`, `Dimension` is somewhat underused; it is used primarily by methods that need to return a width and a height as a single object; for example, `getSize()` returns a `Dimension` object.

Dimension Methods

Variables

A `Dimension` instance has two variables, one for width and one for height. They are accessible directly or through use of the `getSize()` method.

public int width

> The width variable represents the size of an object along the x axis (left to right). Width should not be negative; however, there is nothing within the class to prevent this from happening.

public int height

> The height variable represents the size of an object along the y axis (top to bottom). Height should not be negative; however, there is nothing within the class to prevent this from happening.

Constructors

public Dimension ()

> This constructor creates a `Dimension` instance with a width and height of 0.

public Dimension (Dimension dim)

This constructor creates a copy of `dim`. The initial width is `dim.width`. The initial height is `dim.height`.

public Dimension (int width, int height)

This constructor creates a `Dimension` with an initial width of `width` and an initial height of `height`.

Sizing

public Dimension getSize () ★

The `getSize()` method retrieves the current size as a new `Dimension`, even though the instance variables are public.

public void setSize (int width, int height) ★

The `setSize()` method changes the dimension's size to `width` x `height`.

public void setSize (Dimension d) ★

The `setSize()` method changes the dimension's size to `d.width` x `d.height`.

Miscellaneous methods

public boolean equals (Object object)

The `equals()` method overrides the `Object.equals()` method to define equality for dimensions. Two `Dimension` objects are equal if their width and height values are equal.

public String toString ()

The `toString()` method of `Dimension` returns a string showing the current width and height settings. For example:

```
java.awt.Dimension[width=0,height=0]
```

2.5 Rectangle

The `Rectangle` class encapsulates x and y coordinates and width and height (`Point` and `Dimension` information) within a single object. It is often used by methods that return a rectangular boundary as a single object: for example, `Polygon.getBounds()`, `Component.getBounds()`, and `Graphics.getClipBounds()`. Like `Point`, the `Rectangle` class is not a visual object and does not represent a rectangle on the screen; ironically, `drawRect()` and `fillRect()` don't take `Rectangle` as an argument.

Rectangle Methods

Variables

The four public variables available for `Rectangle` have the same names as the public instance variables of `Point` and `Dimension`. They are all accessible directly or through use of the `getBounds()` method.

public int x

> The x coordinate of the upper left corner.

public int y

> The y coordinate of the upper left corner.

public int width

> The width variable represents the size of the `Rectangle` along the horizontal axis (left to right). Width should not be negative; however, there is nothing within the class to prevent this from happening.

public int height

> The height variable represents the size of the `Rectangle` along the vertical axis (top to bottom). Height should not be negative; however, there is nothing within the class to prevent this from happening.

Constructors

The following seven constructors create `Rectangle` objects. When you create a `Rectangle`, you provide the location of the top left corner, along with the `Rectangle`'s width and height. A `Rectangle` located at (0,0) with a width and height of 100 has its bottom right corner at (99, 99). The `Point` (100, 100) lies outside the `Rectangle`, since that would require a width and height of 101.

public Rectangle ()

> This `Rectangle` constructor creates a `Rectangle` object in which x, y, width, and height are all 0.

public Rectangle (int width, int height)

> This `Rectangle` constructor creates a `Rectangle` with (`x`, `y`) coordinates of (0,0) and the specified `width` and `height`. Notice that there is no `Rectangle(int x, int y)` constructor because that would have the same method signature as this one, and the compiler would have no means to differentiate them.

public Rectangle (int x, int y, int width, int height)

> The `Rectangle` constructor creates a `Rectangle` object with an initial x coordinate of `x`, y coordinate of `y`, width of `width`, and height of `height`. Height and width should be positive, but the constructor does not check for this.

public Rectangle (Rectangle r)

> This `Rectangle` constructor creates a `Rectangle` matching the original. The (`x`, `y`) coordinates are (`r.x`, `r.y`), with a width of `r.width` and a height of `r.height`.

public Rectangle (Point p, Dimension d)

> This `Rectangle` constructor creates a `Rectangle` with (`x`, `y`) coordinates of (`p.x`, `p.y`), a width of `d.width`, and a height of `d.height`.

public Rectangle (Point p)

This `Rectangle` constructor creates a `Rectangle` with (`x`, `y`) coordinates of (`p.x`, `p.y`). The width and height are both zero.

public Rectangle (Dimension d)

The last `Rectangle` constructor creates a `Rectangle` with (`x`, `y`) coordinates of (0, 0). The initial `Rectangle` width is `d.width` and height is `d.height`.

Shaping and sizing

public Rectangle getBounds() ★

The `getBounds()` method returns a copy of the original `Rectangle`.

public void setBounds (int x, int y, int width, int height) ★
public void reshape (int x, int y, int width, int height) ☆

The `setBounds()` method changes the origin of the `Rectangle` to (`x`, `y`) and changes the dimensions to `width` by `height`.

`reshape()` is the Java 1.0 name for this method.

public void setBounds (Rectangle r) ★

The `setBounds()` method changes the origin of the `Rectangle` to (`r.x`, `r.y`) and changes the dimensions to `r.width` by `r.height`.

public Point getLocation() ★

The `getLocation()`retrieves the current origin of this rectangle as a `Point`.

public void setLocation (int x, int y) ★
public void move (int x, int y) ☆

The `setLocation()` method changes the origin of the `Rectangle` to (`x`, `y`).

`move()` is the Java 1.0 name for this method.

public void setLocation (Point p) ★

The `setLocation()` method changes the `Rectangle`'s origin to (`p.x`, `p.y`).

public void translate (int x, int y)

The `translate()` method moves the `Rectangle`'s origin by the amount (`x`, `y`). If the original `Rectangle`'s location (r) is (3, 4) and you call `r.translate (4, 5)`, then r's location becomes (7, 9). x and y may be negative. `translate()` has no effect on the `Rectangle`'s width and height.

public Dimension getSize () ★

The `getSize()` method retrieves the current size of the rectangle as a `Dimension`.

public void setSize() (int width, int height) ★
public void resize (int width, int height) ☆

The `setSize()` method changes the `Rectangle`'s dimensions to `width` x `height`.

`resize()` is the Java 1.0 name for this method.

public void setSize() (Dimension d) ★

The `setSize()` method changes the `Rectangle`'s dimensions to `d.width` x `d.height`.

public void grow (int horizontal, int vertical)

The `grow()` method increases the `Rectangle`'s dimensions by adding the amount `horizontal` on the left and the right and adding the amount `vertical` on the top and bottom. Therefore, all four of the rectangle's variables change. If the original location is (`x`, `y`), the new location will be (x-`horizontal`, y-`vertical`) (moving left and up if both values are positive); if the original size is (`width, height`), the new size will be (`width+2*horizontal, height+2*vertical`). Either horizontal or vertical can be negative to decrease the size of the `Rectangle`. The following code demonstrates the changes:

```
import java.awt.Rectangle;
public class rect {
    public static void main (String[] args) {
        Rectangle r = new Rectangle (100, 100, 200, 200);
        System.out.println (r);
        r.grow (50, 75);
        System.out.println (r);
        r.grow (-25, -50);
        System.out.println (r);
    }
}
```

This program produces the following output:

```
java.awt.Rectangle[x=100,y=100,width=200,height=200]
java.awt.Rectangle[x=50,y=25,width=300,height=350]
java.awt.Rectangle[x=75,y=75,width=250,height=250]
```

public void add (int newX, int newY)

> The `add()` method incorporates the point (`newX`, `newY`) into the `Rectangle`. If this point is already in the `Rectangle`, there is no change. Otherwise, the size of the `Rectangle` increases to include (`newX`, `newY`) within itself.

public void add (Point p)

> This `add()` method incorporates the point (`p.x`, `p.y`) into the `Rectangle`. If this point is already in the `Rectangle`, there is no change. Otherwise, the size of the `Rectangle` increases to include (`p.x`, `p.y`) within itself.

public void add (Rectangle r)

> This `add()` method incorporates another `Rectangle r` into this `Rectangle`. This transforms the current rectangle into the union of the two `Rectangles`. This method might be useful in a drawing program that lets you select multiple objects on the screen and create a rectangular area from them.
>
> We will soon encounter a method called `union()` that is almost identical. `add()` and `union()` differ in that `add()` modifies the current `Rectangle`, while `union()` returns a new `Rectangle`. The resulting rectangles are identical.

Intersections

public boolean contains (int x, int y) ★
public boolean inside (int x, int y) ☆

The `contains()` method determines if the point (`x`, `y`) is within this `Rectangle`. If so, `true` is returned. If not, `false` is returned.

`inside()` is the Java 1.0 name for this method.

public boolean contains (Point p) ★

The `contains()` method determines if the point (`p.x`, `p.y`) is within this `Rectangle`. If so, `true` is returned. If not, `false` is returned.

public boolean intersects (Rectangle r)

The `intersects()` method checks whether `Rectangle r` crosses this `Rectangle` at any point. If it does, `true` is returned. If not, `false` is returned.

public Rectangle intersection (Rectangle r)

The `intersection()` method returns a new `Rectangle` consisting of all points that are in both the current `Rectangle` and `Rectangle r`. For example, if `r = new Rectangle (50, 50, 100, 100)` and `r1 = new Rectangle (100, 100, 75, 75)`, then `r.intersection (r1)` is the `Rectangle (100, 100, 50, 50)`, as shown in Figure 2.13.

public Rectangle union (Rectangle r)

The `union()` method combines the current `Rectangle` and `Rectangle r` to form a new `Rectangle`. For example, if `r = new Rectangle (50, 50, 100, 100)` and `r1 = new Rectangle (100, 100, 75, 75)`, then `r.union (r1)` is the `Rectangle (50, 50, 125, 125)`. The original rectangle is unchanged. Figure 2.14 demonstrates the effect of `union()`. Because `fillRect()` fills to `width-1` and `height-1`, the rectangle drawn appears slightly smaller than you would expect. However, that's an artifact of how rectangles are drawn; the returned rectangle contains all the points within both.

Figure 2.13: Rectangle intersection

[Graphic: Figure 2-13]

Figure 2.14: Rectangle union

[Graphic: Figure 2-14]

Miscellaneous methods

public boolean isEmpty ()

> The `isEmpty()` method checks whether there are any points within the `Rectangle`. If the width and height of the `Rectangle` are both 0 (or less), the `Rectangle` is empty, and this method returns `true`. If either width or height is greater than zero, `isEmpty()` returns `false`. This method could be used to check the results of a call to any method that returns a `Rectangle` object.

public int hashCode ()

> The `hashCode()` method returns a hash code for the rectangle. The system calls this method when a `Rectangle` is used as the key for a hash table.

public boolean equals (Object object)

The `equals()` method overrides the `Object`'s `equals()` method to define what equality means for `Rectangle` objects. Two `Rectangle` objects are equal if their `x`, `y`, width, and height values are equal.

public String toString ()

The `toString()` method of `Rectangle` displays the current values of the `x`, `y`, width, and height variables. For example:

```
java.awt.Rectangle[x=100,y=200,width=300,height=400]
```

PREVIOUS | HOME | NEXT
Shape | BOOK INDEX | Polygon

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

2.6 Polygon

A `Polygon` is a collection of points used to create a series of line segments. Its primary purpose is to draw arbitrary shapes like triangles or pentagons. If the points are sufficiently close, you can create a curve. To display the `Polygon`, call `drawPolygon()` or `fillPolygon()`.

Polygon Methods

Variables

The collection of points maintained by `Polygon` are stored in three variables:

public int npoints

> The `npoints` variable stores the number of points.

public int xpoints[]

> The `xpoints` array holds the x component of each point.

public int ypoints[]

> The `ypoints` array holds the y component of each point.

You might expect the `Polygon` class to use an array of points, rather than separate arrays of integers. More important, you might expect the instance variables to be private or protected, which would prevent them from being modified directly. Since the three instance variables are public, there is no guarantee that the array sizes are in sync with each other or with `npoints`. To avoid trouble, always use `addPoints()` to modify your polygons, and avoid modifying the instance variables directly.
Constructors

public Polygon ()

This constructor creates an empty `Polygon`.

public Polygon (int xPoints[], int yPoints[], int numPoints)

This constructor creates a `Polygon` that consists of `numPoints` points. Those points are formed from the first `numPoints` elements of the `xPoints` and `yPoints` arrays. If the `xPoints` or `yPoints` arrays are larger than `numPoints`, the additional entries are ignored. If the `xPoints` or `yPoints` arrays do not contain at least `numPoints` elements, the constructor throws the run-time exception `ArrayIndexOutOfBoundsException`.

Miscellaneous methods

public void addPoint (int x, int y)

The `addPoint()` method adds the point (`x`, `y`) to the `Polygon` as its last point. If you alter the `xpoints`, `ypoints`, and `npoints` instance variables directly, `addPoint()` could add the new point at a place other than the end, or it could throw the run-time exception `ArrayIndexOutOfBoundsException` with a message showing the position at which it tried to add the point. Again, for safety, don't modify a `Polygon`'s instance variables yourself; always use `addPoint()`.

public Rectangle getBounds () ★
public Rectangle getBoundingBox () ☆

The `getBounds()` method returns the `Polygon`'s bounding `Rectangle` (i.e., the smallest rectangle that contains all the points within the polygon). Once you have the bounding box, it's easy to use methods like `copyArea()` to copy the `Polygon`.

`getBoundingBox()` is the Java 1.0 name for this method.

public boolean contains (int x, int y) ★
public boolean inside (int x, int y) ☆

The `contains()` method checks to see if the (`x`, `y`) point is within an area that would be filled if the `Polygon` was drawn with `Graphics.fillPolygon()`. A point may be within the bounding rectangle of the polygon, but `contains()` can still return `false` if not within a closed part of the polygon.

`inside()` is the Java 1.0 name for this method.

public boolean contains (Point p) ★

The `contains()` method checks to see if the point `p` is within an area that would be filled if the `Polygon` were drawn with `Graphics.fillPolygon()`.

public void translate (int x, int y) ★

The `translate()` method moves all the `Polygon`'s points by the amount (`x`, `y`). This allows you to alter the location of the `Polygon` by shifting the points.

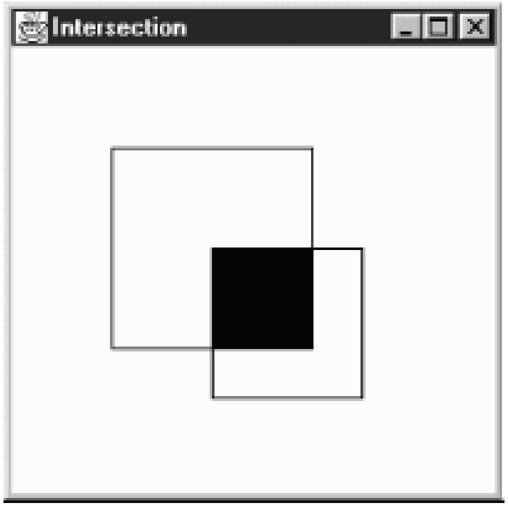
PREVIOUS | HOME | NEXT
Rectangle | BOOK INDEX | Image

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

2.7 Image

An `Image` is a displayable object maintained in memory. AWT has built-in support for reading files in GIF and JPEG format, including GIF89a animation. Netscape Navigator, Internet Explorer, HotJava, and Sun's JDK also understand the XBM image format. Images are loaded from the filesystem or network by the `getImage()` method of either `Component` or `Toolkit`, drawn onto the screen with `drawImage()` from `Graphics`, and manipulated by several objects within the `java.awt.image` package. Figure 2.15 shows an `Image`.

Figure 2.15: An Image



`Image` is an abstract class implemented by many different platform-specific classes. The system that runs your program will provide an appropriate implementation; you do not need to know anything about the platform-specific classes, because the `Image` class completely defines the API for working with images. If you're curious, the platform-specific packages used by the JDK are:

- `sun.awt.win32.Win32Image` on Java 1.0 Windows NT/95 platforms
- `sun.awt.windows.WImage` on Java 1.1 Windows NT/95 platforms
- `sun.awt.motif.X11Image` on UNIX/Motif platforms
- `sun.awt.macos.MacImage` on the Macintosh

This section covers only the `Image` object itself. AWT also includes a package named `java.awt.image` that includes more advanced image processing utilities. The classes in `java.awt.image` are covered in Chapter 12, *Image Processing*.

Image Methods

Constants

public static final Object UndefinedProperty

> In Java 1.0, the sole constant of `Image` is `UndefinedProperty`. It is used as a return value from the `getProperty()` method to indicate that the requested property is unavailable.

Java 1.1 introduces the `getScaledInstance()` method. The final parameter to the method is a set of hints to tell the method how best to scale the image. The following constants provide possible values for this parameter.

public static final int SCALE_DEFAULT ★

> The `SCALE_DEFAULT` hint should be used alone to tell `getScaledInstance()` to use the default scaling algorithm.

public static final int SCALE_FAST ★

> The `SCALE_FAST` hint tells `getScaledInstance()` that speed takes priority over smoothness.

public static final int SCALE_SMOOTH ★

> The `SCALE_SMOOTH` hint tells `getScaledInstance()` that smoothness takes priority over speed.

public static final int SCALE_REPLICATE ★

> The `SCALE_REPLICATE` hint tells `getScaledInstance()` to use `ReplicateScaleFilter` or a reasonable alternative provided by the toolkit. `ReplicateScaleFilter` is discussed in Chapter 12, *Image Processing*.

public static final int SCALE_AREA_AVERAGING ★

> The `SCALE_AREA_AVERAGING` hint tells `getScaledInstance()` to use `AreaAveragingScaleFilter` or a reasonable alternative provided by the toolkit. `AreaAveragingScaleFilter` is discussed in Chapter 12, *Image Processing*.

Constructors

There are no constructors for `Image`. You get an `Image` object to work with by using the `getImage()` method of `Applet` (in an applet), `Toolkit` (in an application), or the `createImage()` method of `Component` or `Toolkit`. `getImage()` uses a separate thread to fetch the image. The thread starts when you call `drawImage()`, `prepareImage()`, or any other method that requires image information. `getImage()` returns immediately. You can also use the `MediaTracker` class to force an image to load before it is needed. `MediaTracker` is discussed in the next section. Characteristics

public abstract int getWidth (ImageObserver observer)

> The `getWidth()` method returns the width of the image object. The width may not be available if the image has not started loading; in this case, `getWidth()` returns -1. An image's size is available long before loading is complete, so it is often useful to call `getWidth()` while the image is loading.

public abstract int getHeight (ImageObserver observer)

> The `getHeight()` method returns the height of the image object. The height may not be available if the image has not started loading; in this case, the `getHeight()` method returns -1. An image's size is available long before loading is complete, so it is often useful to call `getHeight()` while the image is loading.

Miscellaneous methods

public Image getScaledInstance (int width, int height, int hints) ★

> The `getScaledInstance()` method enables you to generate scaled versions of images before they are needed. Prior to Java 1.1, it was necessary to tell the `drawImage()` method to do the scaling. However, this meant that scaling didn't take place until you actually tried to draw the image. Since scaling takes time, drawing the image required more time; the net result was degraded appearance. With Java 1.1, you can generate scaled copies of images before drawing them; then you can use a version of `drawImage()` that does not do scaling, and therefore is much quicker.
>
> The `width` parameter of `getScaledInstance()` is the new width of the image. The `height` parameter is the new height of the image. If either is -1, the scaling retains the aspect ratio of the original image. For instance, if the original image size was 241 by 72 pixels, and `width` and `height` were 100 and -1, the new image size would be 100 by 29 pixels. If both width and height are -1, the `getScaledInstance()` method retains the image's original size. The `hints` parameter is one of the `Image` class constants.

```
Image i = getImage (getDocumentBase(), "rosey.jpg");
Image j = i.getScaledInstance (100, -1, Image.SCALE_FAST);
```

public abstract ImageProducer getSource ()

The getSource() method returns the image's producer, which is an object of type ImageProducer. This object represents the image's source. Once you have the ImageProducer, you can use it to do additional image processing; for example, you could create a modified version of the original image by using a FilteredImageSource. Image producers and image filters are covered in Chapter 12, *Image Processing*.

public abstract Graphics getGraphics ()

The getGraphics() method returns the image's graphics context. The method getGraphics() works only for Image objects created in memory with Component.createImage (int, int). If the image came from a URL or a file (i.e., from getImage()), getGraphics() throws the run-time exception ClassCastException.

public abstract Object getProperty (String name, ImageObserver observer)

The getProperty() method interacts with the image's property list. An object representing the requested property name will be returned for observer. observer represents the Component on which the image is rendered. If the property name exists but is not available yet, getProperty() returns null. If the property name does not exist, the getProperty() method returns the Image.UndefinedProperty object.

Each image type has its own property list. A property named comment stores a comment String from the image's creator. The CropImageFilter adds a property named croprect. If you ask getProperty() for an image's croprect property, you get a Rectangle that shows how the original image was cropped.

public abstract void flush()

The flush() method resets an image to its initial state. Assume you acquire an image over the network with getImage(). The first time you display the image, it will be loaded over the network. If you redisplay the image, AWT normally reuses the original image. However, if you call flush() before redisplaying the image, AWT fetches the image again from its source. (Images created with createImage() aren't affected.) The flush() method is useful if you expect images to change while your program is running. The following program demonstrates flush(). It reloads and displays the file *flush.gif* every time you click the mouse. If you change the file *flush.gif* and click on the mouse, you will see the new file.

```
import java.awt.*;
public class flushMe extends Frame {
    Image im;
    flushMe () {
        super ("Flushing");
        im = Toolkit.getDefaultToolkit().getImage ("flush.gif");
        resize (175, 225);
    }
    public void paint (Graphics g) {
        g.drawImage (im, 0, 0, 175, 225, this);
    }
    public boolean mouseDown (Event e, int x, int y) {
        im.flush();
        repaint();
        return true;
    }
    public static void main (String [] args) {
        Frame f = new flushMe ();
        f.show();
    }
}
```

Simple Animation

Creating simple animation sequences in Java is easy. Load a series of images, then display the images one at a time. Example 2.1 is an application that displays a simple animation sequence. Example 2.2 is an applet that uses a thread to run the application. These programs are far from ideal. If you try them, you'll probably notice some flickering or missing images. We discuss how to fix these problems shortly.

Example 2.1: Animation Application

```
import java.awt.*;
public class Animate extends Frame {
    static Image im[];
    static int numImages = 12;
    static int counter=0;
    Animate () {
        super ("Animate");
    }
    public static void main (String[] args) {
        Frame f = new Animate();
        f.resize (225, 225);
        f.show();
        im = new Image[numImages];
        for (int i=0;i<numImages;i++) {
            im[i] = Toolkit.getDefaultToolkit().getImage ("clock"+i+".jpg");
        }
    }
    public synchronized void paint (Graphics g) {
        g.translate (insets().left, insets().top);
        g.drawImage (im[counter], 0, 0, this);
        counter++;
        if (counter == numImages)
            counter = 0;
        repaint (200);
    }
}
```

This application displays images with the name *clock*n*.jpg*, where *n* is a number between 0 and 11. It fetches the images using the `getImage()` method of the `Toolkit` class--hence, the call to `Toolkit.getDefaultToolkit()`, which gets a `Toolkit` object to work with. The `paint()` method displays the images in sequence, using `drawImage()`. `paint()` ends with a call to `repaint(200)`, which schedules another call to `paint()` in 200 milliseconds.

The `AnimateApplet`, whose code is shown in Example 2.2, does more or less the same thing. It is able to use the `Applet.getImage()` method. A more significant difference is that the applet creates a new thread to control the animation. This thread calls `sleep(200)`, followed by `repaint()`, to display a new image every 200 milliseconds.

Example 2.2: Multithreaded Animation Applet

```
import java.awt.*;
import java.applet.*;
public class AnimateApplet extends Applet implements Runnable {
    static Image im[];
    static int numImages = 12;
    static int counter=0;
    Thread animator;
    public void init () {
        im = new Image[numImages];
        for (int i=0;i<numImages;i++)
            im[i] = getImage (getDocumentBase(), "clock"+i+".jpg");
    }
    public void start() {
        if (animator == null) {
            animator = new Thread (this);
            animator.start ();
        }
    }
    public void stop() {
        if ((animator != null) && (animator.isAlive())) {
            animator.stop();
            animator = null;
        }
    }
    public void run () {
        while (animator != null) {
            try {
                animator.sleep(200);
                repaint ();
                counter++;
                if (counter==numImages)
                    counter=0;
            } catch (Exception e) {
                e.printStackTrace ();
            }
        }
    }
    public void paint (Graphics g) {
        g.drawImage (im[counter], 0, 0, this);
    }
}
```

One quick fix will help the flicker problem in both of these examples. The `update()` method (which is inherited from the `Component` class) normally clears the drawing area and calls `paint()`. In our examples, clearing the drawing area is unnecessary and, worse, results in endless flickering; on slow machines, you'll see `update()` restore the background color between each image. It's a simple matter to override `update()` so that it doesn't clear the drawing area first. Add the following method to both of the previous examples:

```
public void update (Graphics g) {
    paint (g);
}
```

Overriding `update()` helps, but the real solution to our problem is double buffering, which we'll turn to next.

Double Buffering

Double buffering means drawing to an offscreen graphics context and then displaying this graphics context to the screen in a single operation. So far, we have done all our drawing directly on the screen--that is, to the graphics context provided by the `paint()` method. As your programs grow more complex, `paint()` gets bigger and bigger, and it takes more time and resources to update the entire drawing area. On a slow machine, the user will see the individual drawing operations take place, which will make your program look slow and clunky. By using the double buffering technique, you can take your time drawing to another graphics context that isn't displayed. When you are ready, you tell the system to display the completely new image at once. Doing so eliminates the possibility of seeing partial screen updates and flickering.

The first thing you need to do is create an image as your drawing canvas. To get an image object, call the `createImage()` method. `createImage()` is a method of the `Component` class, which we will discuss in Chapter 5, *Components*. Since `Applet` extends `Component`, you can call `createImage()` within an applet. When creating an application and extending `Frame`, `createImage()` returns `null` until the `Frame`'s peer exists. To make sure that the peer exists, call `addNotify()` in the constructor, or make sure you call `show()` before calling `createImage()`. Here's the call to the `createImage()` method that we'll use to get an `Image` object:

```
Image im = createImage (300, 300); // width and height
```

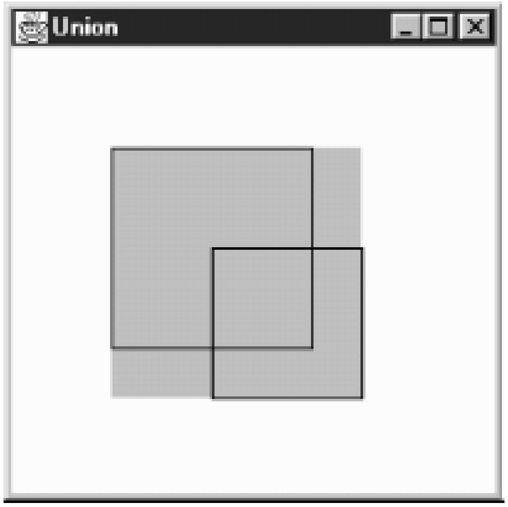
Once you have an `Image` object, you have an area you can draw on. But how do you draw on it? There are no drawing methods associated with `Image`; they're all in the `Graphics` class. So we need to get a `Graphics` context from the `Image`. To do so, call the `getGraphics()` method of the `Image` class, and use that `Graphics` context for your drawing:

```
Graphics buf = im.getGraphics();
```

Now you can do all your drawings with `buf`. To display the drawing, the `paint()` method only needs to call `drawImage(im, . . .)`. Note the hidden connection between the `Graphics` object, `buf`, and the `Image` you are creating, `im`. You draw onto `buf`; then you use `drawImage()` to render the image on the on-screen `Graphics` context within `paint()`.

Another feature of buffering is that you do not have redraw the entire image with each call to `paint()`. The buffered image you're working on remains in memory, and you can add to it at will. If you are drawing directly to the screen, you would have to recreate the entire drawing each time `paint()` is called; remember, `paint()` always hands you a completely new `Graphics` object. Figure 2.16 shows how double buffering works.

Figure 2.16: Double buffering



Example 2.3 puts it all together for you. It plays a game, with one move drawn to the screen each cycle. We still do the drawing within `paint()`, but we draw into an offscreen buffer; that buffer is copied onto the screen by `g.drawImage(im, 0, 0, this)`. If we were doing a lot of drawing, it would be a good idea to move the drawing operations into a different thread, but that would be overkill for this simple applet.

Example 2.3: Double Buffering--Who Won?

```
import java.awt.*;
import java.applet.*;
public class buffering extends Applet {
    Image im;
    Graphics buf;
    int pass=0;
```

```
public void init () {
    // Create buffer
    im = createImage (size().width, size().height);
    // Get its graphics context
    buf = im.getGraphics();
    // Draw Board Once
    buf.setColor (Color.red);
    buf.drawLine (  0,  50, 150,  50);
    buf.drawLine (  0, 100, 150, 100);
    buf.drawLine ( 50,   0,  50, 150);
    buf.drawLine (100,   0, 100, 150);
    buf.setColor (Color.black);
}
public void paint (Graphics g) {
    // Draw image - changes are written onto buf
    g.drawImage (im, 0, 0, this);
    // Make a move
    switch (pass) {
        case 0:
            buf.drawLine (50, 50, 100, 100);
            buf.drawLine (50, 100, 100, 50);
            break;
        case 1:
            buf.drawOval (0, 0, 50, 50);
            break;
        case 2:
            buf.drawLine (100, 0, 150, 50);
            buf.drawLine (150, 0, 100, 50);
            break;
        case 3:
            buf.drawOval (0, 100, 50, 50);
            break;
        case 4:
            buf.drawLine (0, 50, 50, 100);
            buf.drawLine (0, 100, 50, 50);
            break;
        case 5:
            buf.drawOval (100, 50, 50, 50);
            break;
        case 6:
            buf.drawLine (50,  0, 100, 50);
            buf.drawLine (50, 50, 100,  0);
            break;
        case 7:
            buf.drawOval (50, 100, 50, 50);
            break;
        case 8:
                buf.drawLine (100, 100, 150, 150);
                buf.drawLine (150, 100, 100, 150);
                break;
        }
        pass++;
        if (pass <= 9)
            repaint (500);
    }
}
```

2.8 MediaTracker

The `MediaTracker` class assists in the loading of multimedia objects across the network. Tracking is necessary because Java loads images in separate threads. Calls to `getImage()` return immediately; image loading starts only when you call the method `drawImage()`. `MediaTracker` lets you force images to start loading before you try to display them; it also gives you information about the loading process, so you can wait until an image is fully loaded before displaying it.

Currently, `MediaTracker` can monitor the loading of images, but not audio, movies, or anything else. Future versions are rumored to be able to monitor other media types.

MediaTracker Methods

Constants

The `MediaTracker` class defines four constants that are used as return values from the class's methods. These values serve as status indicators.

public static final int LOADING

> The `LOADING` variable indicates that the particular image being checked is still loading.

public static final int ABORTED

> The `ABORTED` variable indicates that the loading process for the image being checked aborted. For example, a timeout could have happened during the download. If something `ABORTED` during loading, it is possible to `flush()` the image to force a retry.

public static final int ERRORED

The ERRORED variable indicates that an error occurred during the loading process for the image being checked. For instance, the image file might not be available from the server (invalid URL) or the file format could be invalid. If an image has ERRORED, retrying it will fail.

public static final int COMPLETE

The COMPLETE flag means that the image being checked successfully loaded.

If COMPLETE, ABORTED, or ERRORED is set, the image has stopped loading. If you are checking multiple images, you can OR several of these values together to form a composite. For example, if you are loading several images and want to find out about any malfunctions, call statusAll() and check for a return value of ABORTED | ERRORED. Constructors

public MediaTracker (Component component)

The MediaTracker constructor creates a new MediaTracker object to track images to be rendered onto component.

Adding images

The addImage() methods add objects for the MediaTracker to track. When placing an object under a MediaTracker's control, you must provide an identifier for grouping purposes. When multiple images are grouped together, you can perform operations on the entire group with a single request. For example, you might want to wait until all the images in an animation sequence are loaded before starting the animation; in this case, assigning the same ID to all the images makes good sense. However, when multiple images are grouped together, you cannot check on the status of a single image. The moral is: if you care about the status of individual images, put each into a group by itself.

Folklore has it that the identifier also serves as a loading priority, with a lower ID meaning a higher priority. This is not completely true. Current implementations start loading lower IDs first, but at most, this is implementation-specific functionality for the JDK. Furthermore, although an object with a lower identifier might be told to start loading first, the MediaTracker does nothing to ensure that it finishes first.

public synchronized void addImage (Image image, int id, int width, int height)

The addImage() method tells the MediaTracker instance that it needs to track the loading of image. The id is used as a grouping. Someone will eventually render the image at a scaled size of width x height. If width and height are both -1, the image will be rendered unscaled. If you forget to notify the MediaTracker that the image will be scaled and ask the MediaTracker to waitForID (id), it is possible that the image may not be fully ready when you try to draw it.

public void addImage (Image image, int id)

The `addImage()` method tells the `MediaTracker` instance that it needs to track the loading of `image`. The `id` is used as a grouping. The `image` will be rendered at its actual size, without scaling.

Removing images

Images that have finished loading are still watched by the `MediaTracker`. The `removeImage()` methods, added in Java 1.1, allow you to remove objects from the `MediaTracker`. Once you no longer care about an image (usually after waiting for it to load), you can remove it from the tracker. Getting rid of loaded images results in better performance because the tracker has fewer objects to check. In Java 1.0, you can't remove an image from `MediaTracker`.

public void removeImage (Image image) ★

The `removeImage()` method tells the `MediaTracker` to remove all instances of `image` from its tracking list.

public void removeImage (Image image, int id) ★

The `removeImage()` method tells the `MediaTracker` to remove all instances of `image` from group `id` of its tracking list.

public void removeImage (Image image, int id, int width, int height) ★

This `removeImage()` method tells the `MediaTracker` to remove all instances of `image` from group `id` and scale `width` x `height` of its tracking list.

Waiting

A handful of methods let you wait for a particular image, image group, all images, or a particular time period. They enable you to be sure that an image has finished trying to load prior to continuing. The fact that an image has finished loading does not imply it has successfully loaded. It is possible that an error condition arose, which caused loading to stop. You should check the status of the image (or group) for actual success.

public void waitForID (int id) throws InterruptedException

The `waitForID()` method blocks the current thread from running until the images added with `id` finish loading. If the wait is interrupted, `waitForID()` throws an `InterruptedException`.

public synchronized boolean waitForID (int id, long ms) throws InterruptedException

> The `waitForID()` method blocks the current thread from running until the images added with `id` finish loading or until `ms` milliseconds have passed. If all the images have loaded, `waitForID()` returns `true`; if the timer has expired, it returns `false`, and one or more images in the `id` set have not finished loading. If `ms` is 0, it waits until all images added with `id` have loaded, with no timeout. If the wait is interrupted, `waitForID()` throws an `InterruptedException`.

public void waitForAll () throws InterruptedException

> The `waitForAll()` method blocks the current thread from running until all images controlled by this `MediaTracker` finish loading. If the wait is interrupted, `waitForAll()` throws an `InterruptedException`.

public synchronized boolean waitForAll (long ms) throws InterruptedException

> The `waitForAll()` method blocks the current thread from running until all images controlled by this `MediaTracker` finish loading or until `ms` milliseconds have passed. If all the images have loaded, `waitForAll()` returns `true`; if the timer has expired, it returns `false`, and one or more images have not finished loading. If `ms` is 0, it waits until all images have loaded, with no timeout. When you interrupt the waiting, `waitForAll()` throws an `InterruptedException`.

Checking status

Several methods are available to check on the status of images loading. None of these methods block, so you can continue working while images are loading.

public boolean checkID (int id)

> The `checkID()` method determines if all the images added with the `id` tag have finished loading. The method returns `true` if all images have completed loading (successfully or unsuccessfully). Since this can return `true` on error, you should also use `isErrorID()` to check for errors. If loading has not completed, `checkID()` returns `false`. This method does not force images to start loading.

public synchronized boolean checkID (int id, boolean load)

The `checkID()` method determines if all the images added with the `id` tag have finished loading. If the `load` flag is `true`, any images in the `id` group that have not started loading yet will start. The method returns `true` if all images have completed loading (successfully or unsuccessfully). Since this can return `true` on error, you should also use `isErrorID()` to check for errors. If loading has not completed, `checkID()` returns `false`.

public boolean checkAll ()

The `checkAll()` method determines if all images associated with the `MediaTracker` have finished loading. The method returns `true` if all images have completed loading (successfully or unsuccessfully). Since this can return `true` on error, you should also use `isErrorAny()` to check for errors. If loading has not completed, `checkAll()` returns `false`. This method does not force images to start loading.

public synchronized boolean checkAll (boolean load)

The `checkAll()` method determines if all images associated with the `MediaTracker` have finished loading. If the `load` flag is `true`, any image that has not started loading yet will start. The method returns `true` if all images have completed loading (successfully or unsuccessfully). Since this can return `true` on error, you should also use `isErrorAny()` to check for errors. If loading has not completed, `checkAll()` returns `false`.

public int statusID (int id, boolean load)

The `statusID()` method checks on the load status of the images in the `id` group. If there are multiple images in the group, the results are ORed together. If the `load` flag is `true`, any image in the `id` group that has not started loading yet will start. The return value is some combination of the class constants `LOADING`, `ABORTED`, `ERRORED`, and `COMPLETE`.

public int statusAll (boolean load)

The `statusAll()` method determines the load status of all the images associated with the `MediaTracker`. If this `MediaTracker` is watching multiple images, the results are ORed together. If the `load` flag is `true`, any image that has not started loading yet will start. The return value is some combination of the class constants `LOADING`, `ABORTED`, `ERRORED`, and `COMPLETE`.

public synchronized boolean isErrorID (int id)

The `isErrorId()` method checks whether any media in the `id` group encountered an error while loading. If any image resulted in an error, `isErrorId()` returns `true`; if there were no errors, it returns `false`.

public synchronized boolean isErrorAny ()

> The `isErrorAny()` method checks to see if any image associated with the `MediaTracker` encountered an error. If there was an error, the method returns `true`; if none, `false`.

public synchronized Object[] getErrorsID (int id)

> The `getErrorsID()` method returns an array of the objects that encountered errors in the group ID during loading. If loading caused no errors, the method returns `null`. The return type is an `Object` array instead of an `Image` array because `MediaTracker` will eventually support additional media types.

public synchronized Object[] getErrorsAny ()

> The `getErrorsAny()` method returns an array of all the objects that encountered an error during loading. If there were no errors, the method returns `null`. The return type is an `Object` array instead of an `Image` array because `MediaTracker` will eventually support additional media types.

Using a MediaTracker

The `init()` method improves the `AnimateApplet` from Example 2.2 to ensure that images load before the animation sequence starts. Waiting for images to load is particularly important if there is a slow link between the computer on which the applet is running and the server for the image files. Note that in a few cases, like interlaced GIF files, you might be willing to display an image before it has completely loaded. However, judicious use of `MediaTracker` will give you much more control over your program's behavior.

The new `init()` method creates a `MediaTracker`, puts all the images in the animation sequence under the tracker's control, and then calls `waitForAll()` to wait until the images are loaded. Once the images are loaded, it calls `isErrorsAny()` to make sure that the images loaded successfully.

```
public void init () {
    MediaTracker mt = new MediaTracker (this);
    im = new Image[numImages];
    for (int i=0;i<numImages;i++) {
        im[i] = getImage (getDocumentBase(), "clock"+i+".jpg");
        mt.addImage (im[i], i);
    }
    try {
        mt.waitForAll();
        if (mt.isErrorAny())
            System.out.println ("Error loading images");
    } catch (Exception e) {
        e.printStackTrace ();
    }
}
```

PREVIOUS | HOME | NEXT
Image | BOOK INDEX | Fonts and Colors

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

3.2 FontMetrics

The abstract `FontMetrics` class provides the tools for calculating the actual width and height of text when displayed on the screen. You can use the results to position objects around text or to provide special effects like shadows and underlining.

Like the `Graphics` class, `FontMetrics` is abstract. The run-time Java platform provides a concrete implementation of `FontMetrics`. You don't have to worry about the actual class; it is guaranteed to implement all the methods of `FontMetrics`. In case you're curious, on a Windows 95 platform, either the class `sun.awt.win32.Win32FontMetrics` (JDK1.0) or the class `sun.awt.windows.WFontMetrics` (JDK1.1) extends `FontMetrics`. On a UNIX/Motif platform, the class is `sun.awt.motif.X11FontMetrics`. With the Macintosh, the class is `sun.awt.macos.MacFontMetrics`. If you're not using the JDK, the class names may be different, but the principle still applies: you don't have to worry about the concrete class.

The FontMetrics Class

Variables

protected Font font

> The font whose metrics are contained in this `FontMetrics` object; use the `getFont()` method to get the value.

Constructors

protected FontMetrics (Font font)

> There is no visible constructor for `FontMetrics`. Since the class is abstract, you cannot create a `FontMetrics` object. The way to get the `FontMetrics` for a font is to ask for it. Through the current graphics context, call the method `getGraphics().getFontMetrics()` to retrieve the `FontMetrics` for the current font. If a graphics context isn't available, you can get a `FontMetrics` object from the default `Toolkit` by calling the method `Toolkit.getDefaultToolkit().getFontMetrics (aFontObject)`.

Font height

Four variables describe the height of a font: leading (pronounced like the metal), ascent, descent, and height. Leading is the amount of space required between lines of the same font. Ascent is the space above the baseline required by the tallest character in the font. Descent is the space required below the baseline by the lowest descender (the "tail" of a character like "y"). Height is the total of the three: ascent, baseline, and descent. Figure 3.1 shows these values graphically.

Figure 3.1: Font height metrics

[Graphic: Figure 3-1]

If that were the entire story, it would be simple. Unfortunately, it isn't. Some special characters (for example, capitals with umlauts or accents) are taller than the "tallest" character in the font; so Java defines a value called `maxAscent` to account for these. Similarly, some characters descend below the "greatest" descent, so Java defines a `maxDescent` to handle these cases.

NOTE:

It seems that on Windows and Macintosh platforms there is no difference between the return values of `getMaxAscent()` and `getAscent()`, or between `getMaxDescent()` and `getDescent()`. On UNIX platforms, they sometimes differ. For developing truly portable applications, the `max` methods should be used where necessary.

public int getLeading ()

The `getLeading()`method retrieves the leading required for the `FontMetrics` of the font. The units for this measurement are pixels.

public int getAscent ()

The `getAscent()`method retrieves the space above the baseline required for the tallest character in the font. The units for this measurement are pixels. You cannot get the ascent value for a specific character.

public int getMaxAscent ()

`getMaxAscent()` retrieves the height above the baseline for the character that's really the tallest character in the font, taking into account accents, umlauts, tildes, and other special marks. The units for this measurement are pixels. If you are using only ordinary ASCII characters below 128 (i.e., the English language character set), `getMaxAscent()` is not necessary.

If you're using `getMaxAscent()`, avoid `getHeight()`; `getHeight()` is based on `getAscent()` and doesn't account for extra space.

For some fonts and platforms, `getAscent()` may include the space for the diacritical marks.

public int getDescent ()

The `getDescent()` method retrieves the space below the baseline required for the deepest character for the font. The units for this measurement are pixels. You cannot get the descent value for a specific character.

public int getMaxDescent ()
public int getMaxDecent ()

Some fonts may have special characters that extend farther below the baseline than the value returned by `getDescent()`. `getMaxDescent()` returns the real maximum descent for the font, in pixels. In most cases, you can still use the `getDescent()` method; visually, it is okay for an occasional character to extend into the space between lines. However, if it is absolutely, positively necessary that the descent space does not overlap with the next line's ascent requirements, use `getMaxDescent()` and avoid `getDescent()` and `getHeight()`.

An early beta release of the AWT API included the method `getMaxDecent()`. It is left for compatibility with early beta code. Avoid using it; it is identical to `getMaxDescent()` in every way except spelling. Unfortunately, it is not flagged as deprecated.

public int getHeight ()

The `getHeight()` method returns the sum of `getDescent()`, `getAscent()`, and `getLeading()`. In most cases, this will be the distance between successive baselines when you are displaying multiple lines of text. The height of a font in pixels is not necessarily the size of a font in points.

Don't use `getHeight()` if you are displaying characters with accents, umlauts, and other marks that increase the character's height. In this case, compute the height yourself using the `getMaxAscent()` method. Likewise, you shouldn't use the method `getHeight()` if you are using `getMaxDescent()` instead of `getDescent()`.

Character width

In the horizontal dimension, positioning characters is relatively simple: you don't have to worry about ascenders and descenders, you only have to worry about how far ahead to draw the next character after you have drawn the current one. The "how far" is called the *advance width* of a character. For most cases, the advance width is the actual width plus the intercharacter space. However, it's not a good idea to think in these terms; in many cases, the intercharacter space is actually negative (i.e., the bounding boxes for two adjacent characters overlap). For example, consider an italic font. The top right corner of one character probably extends beyond the character's advance width, overlapping the next character's bounding box. (To see this, look back at Figure 3.1; in particular, look at the *ll* in *O'Reilly*.) If you think purely in terms of the advance width (the amount to move horizontally after drawing a character), you won't run into trouble. Obviously, the advance width depends on the character, unless you're using a fixed width font.

public int charWidth (char character)

This version of the `charWidth()` method returns the advance width of the given `character` in pixels.

public int charWidth (int character)

The `charWidth()` method returns the advance width of the given `character` in pixels. Note that the argument has type `int` rather than `char`. This version is useful when overriding the `Component.keyDown()` method, which gets the integer value of the character pressed as a parameter. With the `KeyEvent` class, you should use the previous version with its `getKeyChar()` method.

public int stringWidth (String string)

The `stringWidth()` method calculates the advance width of the entire `string` in pixels. Among other things, you can use the results to underline or center text within an area of the screen. Example 3.1 and Figure 3.2 show an example that centers several text strings (taken from the command-line arguments) in a `Frame`.

Example 3.1: Centering Text in a Frame

```
import java.awt.*;
public class Center extends Frame {
    static String text[];
    private Dimension dim;
    static public void main (String args[]) {
        if (args.length == 0) {
            System.err.println ("Usage: java Center <some text>");
            return;
        }
        text = args;
        Center f = new Center();
        f.show();
    }
    public void addNotify() {
        super.addNotify();
        int maxWidth = 0;
        FontMetrics fm = getToolkit().getFontMetrics(getFont());
        for (int i=0;i<text.length;i++) {
            maxWidth = Math.max (maxWidth, fm.stringWidth(text[i]));
        }
        Insets inset = insets();
        dim = new Dimension (maxWidth + inset.left + inset.right,
            text.length*fm.getHeight() + inset.top + inset.bottom);
        resize (dim);
    }
    public void paint (Graphics g) {
        g.translate(insets().left, insets().top);
        FontMetrics fm = g.getFontMetrics();
        for (int i=0;i<text.length;i++) {
            int x,y;
            x = (size().width - fm.stringWidth(text[i]))/2;
            y = (i+1)*fm.getHeight()-1;
            g.drawString (text[i], x, y);
        }
    }
}
```

Figure 3.2: Centering text in a frame

[Graphic: Figure 3-2]

This application extends the `Frame` class. It stores its command-line arguments in the `String` array `text[]`. The `addNotify()` method sizes the frame appropriately. It computes the size needed to display the arguments and resizes the `Frame` accordingly. To compute the width, it takes the longest `stringWidth()` and adds the left and right insets. To compute the height, it takes the current font's height, multiplies it by the number of lines to display, and adds insets. Then it is up to the `paint()` method to use `stringWidth()` and `getHeight()` to figure out where to put each string.

public int charsWidth (char data[], int offset, int length)

> The `charsWidth()` method allows you to calculate the advance width of the char array `data`, without first converting `data` to a `String` and calling the `stringWidth()` method. The `offset` specifies the element of `data` to start with; `length` specifies the number of elements to use. The first element of the array has an `offset` of zero. If `offset` or `length` is invalid, `charsWidth()` throws the run-time exception `ArrayIndexOutOfBoundsException`.

public int bytesWidth (byte data[], int offset, int length)

> The `bytesWidth()` method allows you to calculate the advance width of the byte array `data`, without first converting `data` to a `String` and calling the `stringWidth()`method. The `offset` specifies the element of `data` to start with; `length` specifies the number of elements to use. The first element of the array has an `offset` of zero. If `offset` or `length` is invalid, `bytesWidth()` throws the run-time exception `ArrayIndexOutOfBoundsException`.

public int[] getWidths ()

> The `getWidths()` method returns an integer array of the advance widths of the first 255 characters in the `FontMetrics` font. `getWidths()` is very useful if you are continually looking up the widths of ASCII characters. Obtaining the widths as an array and looking up individual character widths yourself results in less method invocation overhead than making many calls to `charWidth()`.

public int getMaxAdvance ()

> The `getMaxAdvance()` method returns the advance pixel width of the widest character in the font. This allows you to reserve enough space for characters before you know what they are. If you know you are going to display only ASCII characters, you are better off calculating the maximum value returned from `getWidths()`. When unable to determine the width in advance, the method `getMaxAdvance()` returns -1.

Miscellaneous methods

public Font getFont ()

> The `getFont()` method returns the specific font for this `FontMetrics` instance.

public String toString ()

> The `toString()` method of `FontMetrics` returns a string displaying the current font, ascent, descent, and height. For example:

```
sun.awt.win32.Win32FontMetrics[font=java.awt.Font[family=TimesRoman,
name=TimesRoman,style=bolditalic,size=20]ascent=17, descent=6, height=24]
```

Because this is an abstract class, the concrete implementation could return something different.

Font Display Example

Example 3.2 displays all the available fonts in the different styles at 12 points. The code uses the `FontMetrics` methods to ensure that there is enough space for each line. Figure 3.3 shows the results, using the Java 1.0 font names, on several platforms.

Example 3.2: Font Display

```
import java.awt.*;
public class Display extends Frame {
    static String[] fonts;
    private Dimension dim;
    Display () {
        super ("Font Display");
        fonts = Toolkit.getDefaultToolkit().getFontList();
    }
    public void addNotify() {
        Font f;
        super.addNotify();
        int height   = 0;
        int maxWidth = 0;
        final int vMargin  = 5, hMargin = 5;
        for (int i=0;i<fonts.length;i++) {
            f = new Font (fonts[i], Font.PLAIN, 12);
            height += getHeight (f);
            f = new Font (fonts[i], Font.BOLD, 12);
            height += getHeight (f);
            f = new Font (fonts[i], Font.ITALIC, 12);
            height += getHeight (f);
            f = new Font (fonts[i], Font.BOLD | Font.ITALIC, 12);
            height += getHeight (f);
            maxWidth = Math.max (maxWidth, getWidth (f, fonts[i] + " BOLDITALIC"));
        }
        Insets inset = insets();
        dim = new Dimension (maxWidth + inset.left + inset.right + hMargin,
                        height + inset.top + inset.bottom + vMargin);
        resize (dim);
    }
    static public void main (String args[]) {
        Display f = new Display();
        f.show();
    }
    private int getHeight (Font f) {
        FontMetrics fm = Toolkit.getDefaultToolkit().getFontMetrics(f);
        return fm.getHeight();
    }
    private int getWidth (Font f, String s) {
        FontMetrics fm = Toolkit.getDefaultToolkit().getFontMetrics(f);
        return fm.stringWidth(s);
    }
    public void paint (Graphics g) {
        int x = 0;
        int y = 0;
        g.translate(insets().left, insets().top);
        for (int i=0;i<fonts.length;i++) {
            Font plain = new Font (fonts[i], Font.PLAIN, 12);
            Font bold = new Font (fonts[i], Font.BOLD, 12);
            Font italic = new Font (fonts[i], Font.ITALIC, 12);
            Font bolditalic = new Font (fonts[i], Font.BOLD | Font.ITALIC, 12);
            g.setFont (plain);
            y += getHeight (plain);
            g.drawString (fonts[i] + " PLAIN", x, y);
            g.setFont (bold);
            y += getHeight (bold);
            g.drawString (fonts[i] + " BOLD", x, y);
            g.setFont (italic);
            y += getHeight (italic);
            g.drawString (fonts[i] + " ITALIC", x, y);
            g.setFont (bolditalic);
            y += getHeight (bolditalic);
            g.drawString (fonts[i] + " BOLDITALIC", x, y);
        }
        resize (dim);
    }
}
```

Figure 3.3: Fonts available with the Netscape Navigator 3.0 and Internet Explorer 3.0

[Graphic: Figure 3-3]

PREVIOUS

NEXT

3.3 Color

Not so long ago, color was a luxury; these days, color is a requirement. A program that uses only black and white seems hopelessly old fashioned. AWT's `Color` class lets you define and work with `Color` objects. When we discuss the `Component` class (see Chapter 5, *Components*), you will see how to use these color objects, and our discussion of the `SystemColor` subclass (new to Java 1.1; discussed later in this chapter) shows you how to control the colors that are painted on the screen.

A few words of warning: while colors give you the opportunity to make visually pleasing applications, they also let you do things that are incredibly ugly. Resist the urge to go overboard with your use of color; it's easy to make something hideous when you are trying to use every color in the palette. Also, realize that colors are fundamentally platform dependent, and in a very messy way. Java lets you use the same `Color` objects on any platform, but it can't guarantee that every display will treat the color the same way; the result depends on everything from your software to the age of your monitor. What looks pink on one monitor may be red on another. Furthermore, when running in an environment with a limited palette, AWT picks the available color that is closest to what you requested. If you really care about appearance, there is no substitute for testing.

Color Methods

Constants

The `Color` class has predefined constants (all of type `public static final Color`) for frequently used colors. These constants, their RGB values, and their HSB values (hue, saturation, brightness) are given in Table 3.1.

Table 3.1: Comparison of RGB and HSB Colors

Color	Red	Green	Blue	Hue	Saturation	Brightness
black	0	0	0	0	0	0
blue	0	0	255	.666667	1	1
cyan	0	255	255	.5	1	1
darkGray	64	64	64	0	0	.25098

gray	128	128	128	0	0	.501961
green	0	255	0	.333333	1	1
lightGray	192	192	192	0	0	.752941
magenta	255	0	255	.833333	1	1
orange	255	200	0	.130719	1	1
pink	255	175	175	0	.313726	1
red	255	0	0	0	1	1
white	255	255	255	0	0	1
yellow	255	255	0	.166667	1	1

These constants are used like any other class variable: for example, `Color.red` is a constant `Color` object representing the color red. Many other color constants are defined in the `SystemColor` class. Constructors

When you're not using a predefined constant, you create `Color` objects by specifying the color's red, green, and blue components. Depending on which constructor you use, you can specify the components as integers between 0 and 255 (most intense) or as floating point intensities between 0.0 and 1.0 (most intense). The result is a 24-bit quantity that represents a color. The remaining 8 bits are used to represent transparency: that is, if the color is painted on top of something, does whatever was underneath show through? The `Color` class doesn't let you work with the transparency bits; all `Color` objects are opaque. However, you can use transparency when working with images; this topic is covered in Chapter 12, *Image Processing*.

public Color (int red, int green, int blue)

> This constructor is the most commonly used. You provide the specific `red`, `green`, and `blue` values for the color. Valid values for `red`, `green`, and `blue` are between 0 and 255. The constructor examines only the low-order byte of the integer and ignores anything outside the range, including the sign bit.

public Color (int rgb)

> This constructor allows you to combine all three variables in one parameter, `rgb`. Bits 16-23 represent the red component, and bits 8-15 represent the green component. Bits 0-7 represent the blue component. Bits 24-31 are ignored. Going from three bytes to one integer is fairly easy:

```
(((red & 0xFF) << 16 ) | ((green & 0xFF) << 8) | ((blue & 0xFF) << 0))
```

public Color (float red, float green, float blue)

> This final constructor allows you to provide floating point values between 0.0 and 1.0 for each of `red`, `green`, and `blue`. Values outside of this range yield unpredictable results.

Settings

public int getRed ()

> The `getRed()` method retrieves the current setting for the red component of the color.

public int getGreen ()

> The `getGreen()` method retrieves the current setting for the green component of the color.

public int getBlue ()

> The `getBlue()` method retrieves the current setting for the blue component of the color.

public int getRGB ()

> The `getRGB()` method retrieves the current settings for red, green, and blue in one combined value. Bits 16-23 represent the red component. Bits 8-15 represent the green component. Bits 0-7 represent the blue component. Bits 24-31 are the transparency bits; they are always `0xff` (opaque) when using the default RGB `ColorModel`.

public Color brighter ()

> The `brighter()` method creates a new `Color` that is somewhat brighter than the current color. This method is useful if you want to highlight something on the screen.
>
> **NOTE:**

Black does not get any brighter.

public Color darker ()

> The `darker()` method returns a new `Color` that is somewhat darker than the current color. This method is useful if you are trying to de-emphasize an object on the screen. If you are creating your own `Component`, you can use a `darker()` `Color` to mark it inactive.

Color properties

`Color` properties are very similar to `Font` properties. You can use system properties (or resource files) to allow users to select colors for your programs. The settings have the form `0xRRGGBB`, where `RR` is the red component of the color, `GG` represents the green component, and `BB` represents the blue component. `0x` indicates that the number is in hexadecimal. If you (or your user) are comfortable using decimal values for colors (0x112233 is 1122867 in decimal), you can, but then it is harder to see the values of the different components.

NOTE:

The location of the system properties file depends on the run-time environment and version you are using. Ordinarily, the file will go into a subdirectory of the installation directory or, for environment's where users have home directories, in a subdirectory for the user. Sun's HotJava, JDK, and *appletviewer* tools use the *properties* file in the *.hotjava* directory.

Most browsers do not permit modifying properties, so there is no file.

Java 1.1 adds the idea of "resource files," which are syntactically similar to properties files. Resource files are then placed on the server or within a directory found in the `CLASSPATH`. Updating the properties file is no longer recommended.

For example, consider a screen that uses four colors: one each for the foreground, the background, inactive components, and highlighted text. In the system properties file, you allow users to select colors by setting the following properties:

```
myPackage.myClass.foreground
myPackage.myClass.background
myPackage.myClass.inactive
myPackage.myClass.highlight
```

One particular user set two:

```
myPackage.myClass.foreground=0xff00ff        #magenta
myPackage.myClass.background=0xe0e0e0        #light gray
```

These lines tell the program to use magenta as the foreground color and light gray for the background. The program will use its default colors for inactive components and highlighted text.

public static Color getColor (String name)

> The `getColor()` method gets the color specified by the system property `name`. If `name` is not a valid system property, `getColor()` returns `null`. If the property value does not convert to an integer, `getColor()` returns `null`.
>
> For the properties listed above, if you call `getColor()` with `name` set to the property `myPackage.myClass.foreground`, it returns a magenta `Color` object. If called with `name` set to `myPackage.myClass.inactive`, `getColor()` returns `null`.

public static Color getColor (String name, Color defaultColor)

The `getColor()` method gets the color specified by the system property `name`. This version of the `getColor()` method returns `defaultColor` if `name` is not a valid system property or the property's value does not convert to an integer.

For the previous example, if `getColor()` is called with `name` set to `myPackage.myClass.inactive`, the `getColor()` method returns the value of `defaultColor`. This allows you to provide defaults for properties the user doesn't wish to set explicitly.

public static Color getColor (String name, int defaultColor)

This `getColor()` method gets the color specified by the system property `name`. This version of the `getColor()` method returns `defaultColor` if `name` is not a valid system property or the property's value does not convert to an integer. The default color is specified as an integer in which bits 16-23 represent the red component, 8-15 represent the green component, and 0-7 represent the blue component. Bits 24-31 are ignored. If the property value does not convert to an integer, `defaultColor` is returned.

public static Color decode (String name) ★

The `decode()` method provides an explicit means to decipher color property settings, regardless of where the setting comes from. (The `getColor()` method can decipher settings but only if they're in the system properties file.) In particular, you can use `decode()` to look up color settings in a resource file. The format of `name` is the same as that used by `getColor()`. If the contents of `name` do not translate to a 24-bit integer, the `NumberFormatException` run-time exception is thrown. To perform the equivalent of ``getColor(`myPackage.myClass.foreground`)``, without using system properties, see the following example. For a more extensive example using resource files, see Appendix A.

```
// Java 1.1 only
InputStream is = instance.getClass().getResourceAsStream("propfile");
Properties p = new Properties();
try {
    p.load (is);
    Color c = Color.decode(p.getProperty("myPackage.myClass.foreground"));
} catch (IOException e) {
    System.out.println ("error loading props...");
}
```

Hue, saturation, and brightness

So far, the methods we have seen work with a color's red, green, and blue components. There are many other ways to represent colors. This group of methods allows you to work in terms of the HSB (hue, saturation, brightness) model. Hue represents the base color to work with: working through the colors of the rainbow, red is represented by numbers immediately above 0; magenta is represented by numbers below 1; white is 0; and black is 1. Saturation represents the color's purity, ranging from completely unsaturated (either white or black depending upon brightness) to totally saturated (just the base color present). Brightness is the desired level of luminance, ranging from black (0) to the maximum amount determined by the saturation level.

public static float[] RGBtoHSB (int red, int green, int blue, float[] hsbvalues)

> The `RGBtoHSB()` method allows you to convert a specific `red`, `green`, `blue` value to the hue, saturation, and brightness equivalent. `RGBtoHSB()` returns the results in two different ways: the parameter `hsbvalues` and the method's return value. The values of these are the same. If you do not want to pass an `hsbvalues` array parameter, pass `null`. In both the parameter and the return value, the three components are placed in the array as follows:
>
> `hsbvalues[0]` *contains hue*
> `hsbvalues[1]` *contains saturation*
> `hsbvalues[2]` *contains brightness*

public static Color getHSBColor (float hue, float saturation, float brightness)

> The `getHSBColor()` method creates a `Color` object by using `hue`, `saturation`, and `brightness` instead of `red`, `green`, and `blue` values.

public static int HSBtoRGB (float hue, float saturation, float brightness)

> The `HSBtoRGB()` method converts a specific `hue`, `saturation`, and `brightness` to a `Color` and returns the `red`, `green`, and `blue` values as an integer. As with the constructor, bits 16-23 represent the red component, 8-15 represent the green component, and 0-7 represent the blue component. Bits 24-31 are ignored.

Miscellaneous methods

public int hashCode ()

> The `hashCode()` method returns a hash code for the color. The hash code is used whenever a color is used as a key in a `Hashtable`.

public boolean equals (Object o)

> The `equals()` method overrides the `equals()` method of the `Object` to define equality for `Color` objects. Two `Color` objects are equivalent if their `red`, `green`, and `blue` values are equal.

public String toString ()

The `toString()` method of `Color` returns a string showing the color's red, green, and blue settings. For example `System.out.println (Color.orange)` would result in the following:

```
java.awt.Color[r=255,g=200,b=0]
```

3.4 SystemColor

In Java 1.1, AWT provides access to desktop color schemes, or *themes*. To give you an idea of how these themes work, with the Windows Standard scheme for the Windows 95 desktop, buttons have a gray background with black text. If you use the control panel to change to a High Contrast Black scheme, the button's background becomes black and the text white. Prior to 1.1, Java didn't know anything about desktop colors: all color values were hard coded. If you asked for a particular shade of gray, you got that shade, and that was it; applets and applications had no knowledge of the desktop color scheme in effect, and therefore, wouldn't change in response to changes in the color scheme.

Starting with Java 1.1, you can write programs that react to changes in the color scheme: for example, a button's color will change automatically when you use the control panel to change the color scheme. To do so, you use a large number of constants that are defined in the `SystemColor` class. Although these constants are `public static final`, they actually have a very strange behavior. Your program is not allowed to modify them (like any other constant). However, their initial values are loaded at run-time, and their values may change, corresponding to changes in the color scheme. This has one important consequence for programmers: you should not use `equals()`to compare a `SystemColor` with a "regular" `Color`; use the `getRGB()` methods of the colors you are comparing to ensure that you compare the current color value.[1] Using Desktop Colors contains a usage example.

> [1] The omission of an `equals()` method that can properly compare a `SystemColor` with a `Color` is unfortunate.

Because `SystemColor` is a subclass of `Color`, you can use a `SystemColor` anywhere you can use a `Color` object. You will never create your own `SystemColor` objects; there is no public constructor. The only objects in this class are the twenty or so `SystemColor` constants.

SystemColor Methods

Constants

There are two sets of constants within `SystemColor`. The first set provides names for indices into the internal system color lookup table; you will probably never need to use these. All of them have corresponding constants in the second set, except `SystemColor.NUM_COLORS`, which tells you how many `SystemColor` constants are in the second set.

public final static int ACTIVE_CAPTION ★
public final static int ACTIVE_CAPTION_BORDER ★
public final static int ACTIVE_CAPTION_TEXT ★
public final static int CONTROL ★
public final static int CONTROL_DK_SHADOW ★
public final static int CONTROL_HIGHLIGHT ★
public final static int CONTROL_LT_HIGHLIGHT ★
public final static int CONTROL_SHADOW ★
public final static int CONTROL_TEXT ★
public final static int DESKTOP ★
public final static int INACTIVE_CAPTION ★
public final static int INACTIVE_CAPTION_BORDER ★
public final static int INACTIVE_CAPTION_TEXT ★
public final static int INFO ★
public final static int INFO_TEXT ★
public final static int MENU ★
public final static int MENU_TEXT ★
public final static int NUM_COLORS ★
public final static int SCROLLBAR ★
public final static int TEXT ★
public final static int TEXT_HIGHLIGHT ★
public final static int TEXT_HIGHLIGHT_TEXT ★
public final static int TEXT_INACTIVE_TEXT ★
public final static int TEXT_TEXT ★
public final static int WINDOW ★
public final static int WINDOW_BORDER ★
public final static int WINDOW_TEXT ★

The second set of constants is the set of `SystemColor`s you use when creating `Component` objects, to ensure they appear similar to other objects in the user's desktop environment. By using these symbolic constants, you can create new objects that are well integrated into the user's desktop environment, making it easier for the user to work with your program.

public final static SystemColor activeCaption ★

> The `activeCaption` color represents the background color for the active window's title area. This is automatically set for you when you use `Frame`.

public final static SystemColor activeCaptionBorder ★

> The `activeCaptionBorder` color represents the border color for the active window.

public final static SystemColor activeCaptionText ★

> The `activeCaptionText` color represents the text color to use for the active window's title.

public final static SystemColor control ★

> The `control` color represents the background color for the different components. If you are creating your own `Component` by subclassing `Canvas`, this should be the background color of the new object.

public final static SystemColor controlDkShadow ★

> The `controlDkShadow` color represents a dark shadow color to be used with `control` and `controlShadow` to simulate a three-dimensional appearance. Ordinarily, when not depressed, the `controlDkShadow` should be used for the object's bottom and right edges. When depressed, `controlDkShadow` should be used for the top and left edges.

public final static SystemColor controlHighlight ★

> The `controlHighlight` color represents an emphasis color for use in an area or an item of a custom component.

public final static SystemColor controlLtHighlight ★

> The `controlLtHighlight` color represents a lighter emphasis color for use in an area or an item of a custom component.

public final static SystemColor controlShadow ★

The controlShadow color represents a light shadow color to be used with control and controlDkShadow to simulate a three-dimensional appearance. Ordinarily, when not depressed, the controlShadow should be used for the top and left edges. When depressed, controlShadow should be used for the bottom and right edges.

public final static SystemColor controlText ★

The controlText color represents the text color of a component. Before drawing any text in your own components, you should change the color to controlText with a statement like this:

```
g.setColor(SystemColor.controlText);
```

public final static SystemColor desktop ★

The desktop color represents the background color of the desktop workspace.

public final static SystemColor inactiveCaption ★

The inactiveCaption color represents the background color for an inactive window's title area.

public final static SystemColor inactiveCaptionBorder ★

The inactiveCaptionBorder color represents the border color for an inactive window.

public final static SystemColor inactiveCaptionText ★

The inactiveCaptionText color represents the text color to use for each inactive window's title.

public final static SystemColor info ★

The info color represents the background color for mouse-over help text. When a mouse dwells over an object, any pop-up help text should be displayed in an area of this color. In the Microsoft Windows world, these are also called "tool tips."

public final static SystemColor infoText ★

The infoText color represents the text color for mouse-over help text.

public final static SystemColor menu ★

The `menu` color represents the background color of deselected `MenuItem`-like objects. When the menu is selected, the `textHighlight` color is normally the background color.

public final static SystemColor menuText ★

The `menuText` color represents the color of the text on deselected `MenuItem`-like objects. When a menu is selected, the `textHighlightText` color is normally the text color. If the menu happens to be inactive, `textInactiveText` would be used.

public final static SystemColor scrollbar ★

The `scrollbar` color represents the background color for scrollbars. This color is used by default with `Scrollbar`, `ScrollPane`, `TextArea`, and `List` objects.

public final static SystemColor textHighlight ★

The `textHighlight` color represents the background color of highlighted text; for example, it is used for the selected area of a `TextField` or a selected `MenuItem`.

public final static SystemColor textHighlightText ★

The `textHighlightText` color represents the text color of highlighted text.

public final static SystemColor textInactiveText ★

The `textInactiveText` color represents the text color of an inactive component.

public final static SystemColor textText ★

The `textText` color represents the color of text in `TextComponent` objects.

public final static SystemColor window ★

The `window` color represents the background color of the window's display area. For an applet, this would be the display area specified by the `WIDTH` and `HEIGHT` values of the `<APPLET>` tag (`setBackground(SystemColor.window)`), although you would probably use it more for the background of a `Frame`.

public final static SystemColor windowBorder ★

The `windowBorder` color represents the color of the borders around a window. With AWT, instances of `Window` do not have borders, but instances of `Frame` and `Dialog` do.

public final static SystemColor windowText ★

The `windowText` color represents the color of the text drawn within the window.

NOTE:

Every platform does not fully support every system color. However, on platforms that do not provide natural values for some constants, Java selects reasonable alternate colors.

If you are going to be working only with Java's prefabricated components (`Button`, `List`, etc.), you don't have to worry about system colors; the component's default colors will be set appropriately. You are most likely to use system colors if you are creating your own components. In this case, you will use system colors to make your component emulate the behavior of other components; for example, you will use `controlText` as the color for drawing text, `activeCaption` as the background for the caption of an active window, and so on. Constructors

There are no public constructors for `SystemColor`. If you need to create a new color, use the `Color` class described previously. Miscellaneous methods

public int getRGB ()

The `getRGB()` method retrieves the current settings for red, green, and blue in one combined value, like `Color`. However, since the color value is dynamic, `getRGB()` needs to look up the value in an internal table. Therefore, `SystemColor` overrides `Color.getRGB()`.

public String toString ()

The `toString()` method of `SystemColor` returns a string showing the system color's index into its internal table. For example, the following string is returned by `SystemColor.text.toString()`:

```
java.awt.SystemColor[i=12]
```

3.6 Using Desktop Colors

Example 3.4 demonstrates how to use the desktop color constants introduced in Java 1.1. If you run this example under an earlier release, an uncatchable class verifier error will occur.

NOTE:

Notice that the border lines are drawn from `0` to `width-1` or `height-1`. This is to draw lines of length `width` and `height`, respectively.

Example 3.4: Desktop Color Usage

```
// Java 1.1 only
import java.awt.*;
public class TextBox3D extends Canvas {
    String text;
    public TextBox3D (String s, int width, int height) {
        super();
        text=s;
        setSize(width, height);
    }
    public synchronized void paint (Graphics g) {
        FontMetrics fm = g.getFontMetrics();
        Dimension size=getSize();
        int x = (size.width - fm.stringWidth(text))/2;
        int y = (size.height - fm.getHeight())/2;
        g.setColor (SystemColor.control);
        g.fillRect (0, 0, size.width, size.height);
        g.setColor (SystemColor.controlShadow);
        g.drawLine (0, 0, 0, size.height-1);
        g.drawLine (0, 0, size.width-1, 0);
        g.setColor (SystemColor.controlDkShadow);
        g.drawLine (0, size.height-1, size.width-1, size.height-1);
        g.drawLine (size.width-1, 0, size.width-1, size.height-1);
        g.setColor (SystemColor.controlText);
        g.drawString (text, x, y);
    }
}
```

PREVIOUS Displaying Colors | HOME BOOK INDEX | NEXT Events

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

4.2 The Event Class

An instance of the Event class is a platform-independent representation that encapsulates the specifics of an event that happens within the Java 1.0 model. It contains everything you need to know about an event: who, what, when, where, and why the event happened. Note that the Event class is not used in the Java 1.1 event model; instead, Java 1.1 has an AWTEvent class, with subclasses for different event types.

When an event occurs, you decide whether or not to process the event. If you decide against reacting, the event passes through your program quickly without anything happening. If you decide to handle the event, you must deal with it quickly so the system can process the next event. If handling the event requires a lot of work, you should move the event-handling code into its own thread. That way, the system can process the next event while you go off and process the first. If you do not multithread your event processing, the system becomes slow and unresponsive and could lose events. A slow and unresponsive program frustrates users and may convince them to find another solution for their problems.

Variables

Event contains ten instance variables that offer all the specific information for a particular event.
Instance variables

public Object arg

> The arg field contains some data regarding the event, to be interpreted by the recipient. For example, if the user presses Return within a TextField, an Event with an id of ACTION_EVENT is generated with the TextField as the target and the string within it as the arg. See a description of each specific event to find out what its arg means.

public int clickCount

The `clickCount` field allows you to check for double clicking of the mouse. This field is relevant only for `MOUSE_DOWN` events. There is no way to specify the time delta used to determine how quick a double-click needs to be, nor is there a maximum value for `clickCount`. If a user quickly clicks the mouse four times, `clickCount` is four. Only the passage of a system-specific time delta will reset the value so that the next `MOUSE_DOWN` is the first click. The incrementing of `clickCount` does not care which mouse button is pressed.

public Event evt

The `evt` field does not appear to be used anywhere but is available if you wish to pass around a linked list of events. Then your program can handle this event and tell the system to deal with the next one (as demonstrated in the following code), or you can process the entire chain yourself.

```
public boolean mouseDown (Event e, int x, int y) {
    System.out.println ("Coordinates: " + x + "-" + y);
    if (e.evt != null)
        postEvent (e.evt);
    return true;
}
```

public int id

The `id` field of `Event` contains the identifier of the event. The system-generated events are the following `Event` constants:

`WINDOW_DESTROY`
`WINDOW_EXPOSE`
`WINDOW_ICONIFY`
`WINDOW_DEICONIFY`
`KEY_PRESS`
`KEY_RELEASE`
`KEY_ACTION`
`KEY_ACTION_RELEASE`
`MOUSE_DOWN`
`MOUSE_UP`
`MOUSE_MOVE`
`MOUSE_ENTER`
`MOUSE_EXIT`
`MOUSE_DRAG`
`SCROLL_LINE_UP`
`SCROLL_LINE_DOWN`
`SCROLL_PAGE_UP`
`SCROLL_PAGE_DOWN`
`SCROLL_ABSOLUTE`
`LIST_SELECT`
`LIST_DESELECT`
`ACTION_EVENT`

As a user, you can create your own event types and store your own unique event ID here. In Java 1.0, there is no formal way to prevent conflicts between your events and system events, but using a negative IO is a good ad-hoc method. It is up to you to check all the user events generated in your program in order to avoid conflicts among user events.

public int key

For keyboard-related events, the `key` field contains the integer representation of the keyboard element that caused the event. Constants are available for the keypad keys. To examine `key` as a character, just cast it to a `char`. For nonkeyboard-related events, the value is zero.

pubic int modifiers

The `modifiers` field shows the state of the modifier keys when the event happened. A flag is set for each modifier key pressed by the user when the event happened. Modifier keys are Shift, Control, Alt, and Meta. Since the middle and right mouse key are indicated in a Java event by a modifier key, one reason to use the `modifiers` field is to determine which mouse button triggered an event. See Working With Mouse Buttons in Java 1.0 for an example.

public Object target

The `target` field contains a reference to the object that is the cause of the event. For example, if the user selects a button, the button is the target of the event. If the user moves the mouse into a `Frame`, the `Frame` is the target. The `target` indicates where the event happened, not the component that is dealing with it.

public long when

The `when` field contains the time of the event in milliseconds. The following code converts this `long` value to a `Date` to examine its contents:

```
Date d = new Date (e.when);
```

public int x
public int y

The `x` and `y` fields show the coordinates where the event happened. The coordinates are always relative to the top left corner of the target of the event and get translated based on the top left corner of the container as the event gets passed through the containing components. (See the previous Identifying the Target for an example of this translation.) It is possible for either or both of these to be outside the coordinate space of the applet (e.g., if user quickly moves the mouse outside the applet).

Constants

Numerous constants are provided with the `Event` class. Several designate which event happened (the why). Others are available to help in determining the function key a user pressed (the what). And yet more are available to make your life easier.

When the system generates an event, it calls a handler method for it. To deal with the event, you have to override the appropriate method. The different event type sections describe which methods you override.
Key constants

These constants are set when a user presses a key. Most of them correspond to function and keypad keys; since such keys are generally used to invoke an action from the program or the system, Java calls them *action keys* and causes them to generate a different `Event` type (`KEY_ACTION`) from regular alphanumeric keys (`KEY_PRESS`).

Table 4.2 shows the constants used to represent keys and the event type that uses each constant. The values, which are all declared `public static final int`, appear in the `key` variable of the event instance. A few keys represent ASCII characters that have string equivalents such as `\n`. Black stars (★) mark the constants that are new in Java 1.1; they can be used with the 1.0 event model, provided that you are running Java 1.1. Java 1.1 events use a different set of key constants defined in the `KeyEvent` class.

Table 4.2: Constants for Keys in Java 1.0

Constant	Event Type		Constant	Event Type
HOME	KEY_ACTION		F9	KEY_ACTION
END	KEY_ACTION		F10	KEY_ACTION
PGUP	KEY_ACTION		F11	KEY_ACTION
PGDN	KEY_ACTION		F12	KEY_ACTION
UP	KEY_ACTION		PRINT_SCREEN★	KEY_ACTION
DOWN	KEY_ACTION		SCROLL_LOCK★	KEY_ACTION
LEFT	KEY_ACTION		CAPS_LOCK★	KEY_ACTION
RIGHT	KEY_ACTION		NUM_LOCK★	KEY_ACTION
F1	KEY_ACTION		PAUSE★	KEY_ACTION
F2	KEY_ACTION		INSERT★	KEY_ACTION
F3	KEY_ACTION		ENTER (\n)★	KEY_PRESS

F4	KEY_ACTION		BACK_SPACE (\b)★	KEY_PRESS
F5	KEY_ACTION		TAB (\t)★	KEY_PRESS
F6	KEY_ACTION		ESCAPE★	KEY_PRESS
F7	KEY_ACTION		DELETE★	KEY_PRESS
F8	KEY_ACTION			

Modifiers

Modifiers are keys like Shift, Control, Alt, or Meta. When a user presses any key or mouse button that generates an `Event`, the `modifiers` field of the `Event` instance is set. You can check whether any modifier key was pressed by ANDing its constant with the `modifiers` field. If multiple modifier keys were down at the time the event occurred, the constants for the different modifiers are ORed together in the field.

```
public static final int ALT_MASK
public static final int CTRL_MASK
public static final int META_MASK
public static final int SHIFT_MASK
```

When reporting a mouse event, the system automatically sets the `modifiers` field. Since Java is advertised as supporting the single-button mouse model, all buttons generate the same mouse events, and the system uses the `modifiers` field to differentiate between mouse buttons. That way, a user with a one- or two-button mouse can simulate a three-button mouse by clicking on his mouse while holding down a modifier key. Table 4.3 lists the mouse modifier keys; an applet in Working With Mouse Buttons in Java 1.0 demonstrates how to differentiate between mouse buttons.

Table 4.3: Mouse Button Modifier Keys

Mouse Button	**Modifier Key**
Left mouse button	None
Middle mouse button	`ALT_MASK`
Right mouse button	`META_MASK`

For example, if you have a three-button mouse, and click the right button, Java generates some kind of mouse event with the `META_MASK` set in the `modifiers` field. If you have a one-button mouse, you can generate the same event by clicking the mouse while depressing the Meta key.

NOTE:

If you have a multibutton mouse and do an Alt+right mouse or Meta+left mouse, the results are platform specific. You should get a mouse event with two masks set.

Key events

The component peers deliver separate key events when a user presses and releases nearly any key. `KEY_ACTION` and `KEY_ACTION_RELEASE` are for the function and arrow keys, while `KEY_PRESS` and `KEY_RELEASE` are for the remaining control and alphanumeric keys.

public static final int KEY_ACTION

> The peers deliver the `KEY_ACTION` event when the user presses a function or keypad key. The default `Component.handleEvent()` method calls the `keyDown()` method for this event. If the user holds down the key, this event is generated multiple times. If you are using the 1.1 event model, the interface method `KeyListener.keyPressed()` handles this event.

public static final int KEY_ACTION_RELEASE

> The peers deliver the `KEY_ACTION_RELEASE` event when the user releases a function or keypad key. The default `handleEvent()` method for `Component` calls the `keyUp()` method for this event. If you are using the 1.1 event model, the `KeyListener.keyReleased()` interface method handles this event.

public static final int KEY_PRESS

> The peers deliver the `KEY_PRESS` event when the user presses an ordinary key. The default `Component.handleEvent()` method calls the `keyDown()` method for this event. Holding down the key causes multiple `KEY_PRESS` events to be generated. If you are using the 1.1 event model, the interface method `KeyListener.keyPressed()` handles this event.

public static final int KEY_RELEASE

> The peers deliver `KEY_RELEASE` events when the user releases an ordinary key. The default `handleEvent()` method for `Component` calls the `keyUp()` method for this event. If you are using the 1.1 event model, the interface method `KeyListener.keyReleased()` handles this event.

NOTE:

If you want to capture arrow and keypad keys under the X Window System, make sure the key codes are set up properly, using the *xmodmap* command.

> **NOTE:**

Some platforms generate events for the modifier keys by themselves, whereas other platforms require modifier keys to be pressed with another key. For example, on a Windows 95 platform, if Ctrl+A is pressed, you would expect one `KEY_PRESS` and one `KEY_RELEASE`. However, there is a second `KEY_RELEASE` for the Control key. Under Motif, you get only a single `KEY_RELEASE`.

Window events

Window events happen only for components that are children of `Window`. Several of these events are available only on certain platforms. Like other event types, the `id` variable holds the value of the specific event instance.

public static final int WINDOW_DESTROY

> The peers deliver the `WINDOW_DESTROY` event whenever the system tells a window to destroy itself. This is usually done when the user selects the window manager's Close or Quit window menu option. By default, `Frame` instances do not deal with this event, and you must remember to catch it yourself. If you are using the 1.1 event model, the `WindowListener.windowClosing()` interface method handles this event.

public static final int WINDOW_EXPOSE

> The peers deliver the `WINDOW_EXPOSE` event whenever all or part of a window becomes visible. To find out what part of the window has become uncovered, use the `getClipRect()` method (or `getClipBounds()` in Java version 1.1) of the `Graphics` parameter to the `paint()` method. If you are using the 1.1 event model, the `WindowListener.windowOpening()` interface method most closely corresponds to the handling of this event.

public static final int WINDOW_ICONIFY

> The peers deliver the `WINDOW_ICONIFY` event when the user iconifies the window. If you are using the 1.1 event model, the interface method `WindowListener.windowIconified()` handles this event.

public static final int WINDOW_DEICONIFY

> The peers deliver the `WINDOW_DEICONIFY` event when the user de-iconifies the window. If you

are using the 1.1 event model, the interface method `WindowListener.windowDeiconified()` handles this event.

public static final int WINDOW_MOVED

The `WINDOW_MOVED` event signifies that the user has moved the window. If you are using the 1.1 event model, the `ComponentListener.componentMoved()` interface method handles this event.

Mouse events

The component peers deliver mouse events when a user presses or releases a mouse button. Events are also delivered whenever the mouse moves. In order to be platform independent, Java pretends that all mice have a single button. If you press the second or third button, Java generates a regular mouse event but sets the event's `modifers` field with a flag that indicates which button was pressed. If you press the left button, no `modifiers` flags are set. Pressing the center button sets the `ALT_MASK` flag; pressing the right button sets the `META_MASK` flag. Therefore, you can determine which mouse button was pressed by looking at the `Event.modifiers` attribute. Furthermore, users with a one-button or two-button mouse can generate the same events by pressing a mouse button while holding down the Alt or Meta keys.

NOTE:

Early releases of Java (1.0.2 and earlier) only propagated mouse events from `Canvas` and `Container` objects. With the 1.1 event model, the events that different components process are better defined.

public static final int MOUSE_DOWN

The peers deliver the `MOUSE_DOWN` event when the user presses any mouse button. This action must occur over a component that passes along the `MOUSE_DOWN` event. The default `Component.handleEvent()` method calls the `mouseDown()` method for this event. If you are using the 1.1 event model, the `MouseListener.mousePressed()` interface method handles this event.

public static final int MOUSE_UP

The peers deliver the `MOUSE_UP` event when the user releases the mouse button. This action must occur over a component that passes along the `MOUSE_UP` event. The default `handleEvent()` method for `Component` calls the `mouseUp()` method for this event. If you are using the 1.1 event model, the interface method `MouseListener.mouseReleased()` handles this event.

public static final int MOUSE_MOVE

The peers deliver the `MOUSE_MOVE` event whenever the user moves the mouse over any part of the applet. This can happen many, many times more than you want to track, so make sure you really want to do something with this event before trying to capture it. (You can also capture `MOUSE_MOVE` events and without losing much, choose to deal with only every third or fourth movement.) The default `handleEvent()` method calls the `mouseMove()` method for the event. If you are using the 1.1 event model, the interface method `MouseMotionListener.mouseMoved()` handles this event.

public static final int MOUSE_DRAG

The peers deliver the `MOUSE_DRAG` event whenever the user moves the mouse over any part of the applet with a mouse button depressed. The default method `handleEvent()` calls the `mouseDrag()` method for the event. If you are using the 1.1 event model, the interface method `MouseMotionListener.mouseDragged()` handles this event.

public static final int MOUSE_ENTER

The peers deliver the `MOUSE_ENTER` event whenever the cursor enters a component. The default `handleEvent()` method calls the `mouseEnter()` method for the event. If you are using the 1.1 event model, the interface method `MouseListener.mouseEntered()` handles this event.

public static final int MOUSE_EXIT

The peers deliver the `MOUSE_EXIT` event whenever the cursor leaves a component. The default `handleEvent()` method calls the `mouseExit()` method for the event. If you are using the 1.1 event model, the interface method `MouseListener.mouseExited()` handles this event.

Scrolling events

The peers deliver scrolling events for the `Scrollbar` component. The objects that have a built-in scrollbar (like `List`, `ScrollPane`, and `TextArea`) do not generate these events. No default methods are called for any of the scrolling events. They must be dealt with in the `handleEvent()` method of the `Container` or a subclass of the `Scrollbar`. You can determine which particular event occurred by checking the `id` variable of the event, and find out the new position of the thumb by looking at the `arg` variable or calling `getValue()` on the scrollbar. See also the description of the `AdjustmentListener` interface later in this chapter.

public static final int SCROLL_LINE_UP

The scrollbar peers deliver the `SCROLL_LINE_UP` event when the user presses the arrow pointing up for the vertical scrollbar or the arrow pointing left for the horizontal scrollbar. This decreases the scrollbar setting by one back toward the minimum value. If you are using the 1.1 event model, the interface method `AdjustmentListener.adjustmentValueChanged()` handles this event.

public static final int SCROLL_LINE_DOWN

The peers deliver the `SCROLL_LINE_DOWN` event when the user presses the arrow pointing down for the vertical scrollbar or the arrow pointing right for the horizontal scrollbar. This increases the scrollbar setting by one toward the maximum value. If you are using the 1.1 event model, the interface method `AdjustmentListener.adjustmentValueChanged()` handles this event.

public static final int SCROLL_PAGE_UP

The peers deliver the `SCROLL_PAGE_UP` event when the user presses the mouse with the cursor in the area between the slider and the decrease arrow. This decreases the scrollbar setting by the paging increment, which defaults to 10, back toward the minimum value. If you are using the 1.1 event model, the interface method `AdjustmentListener.adjustmentValueChanged()` handles this event.

public static final int SCROLL_PAGE_DOWN

The peers deliver the `SCROLL_PAGE_DOWN` event when the user presses the mouse with the cursor in the area between the slider and the increase arrow. This increases the scrollbar setting by the paging increment, which defaults to 10, toward the maximum value. If you are using the 1.1 event model, the interface method `AdjustmentListener.adjustmentValueChanged()` handles this event.

public static final int SCROLL_ABSOLUTE

The peers deliver the `SCROLL_ABSOLUTE` event when the user drags the slider part of the scrollbar. There is no set time period or distance between multiple `SCROLL_ABSOLUTE` events. If you are using the Java version 1.1 event model, the `AdjustmentListener.adjustmentValueChanged()` interface method handles this event.

public static final int SCROLL_BEGIN ★

The `SCROLL_BEGIN` event is not delivered by peers, but you may wish to use it to signify when a user drags the slider at the beginning of a series of `SCROLL_ABSOLUTE` events.

`SCROLL_END`, described next, would then be used to signify the end of the series.

public static final int SCROLL_END ★

The `SCROLL_END` event is not delivered by peers, but you may wish to use it to signify when a user drags the slider at the end of a series of `SCROLL_ABSOLUTE` events. `SCROLL_BEGIN`, described previously, would have been used to signify the beginning of the series.

List events

Two events specific to the `List` class are passed along by the peers. They signify when the user has selected or deselected a specific choice in the `List`. It is not ordinarily necessary to capture these events, because the peers deliver the `ACTION_EVENT` when the user double-clicks on a specific item in the `List` and it is this `ACTION_EVENT` that triggers something to happen. However, if there is reason to do something when the user has just single-clicked on a choice, these events may be useful. An example of how they would prove useful is if you are displaying a list of filenames with the ability to preview files before loading. Single selection would preview, double-click would load, and deselect would stop previewing.

No default methods are called for any of the list events. They must be dealt with in the `handleEvent()` method of the `Container` of the `List` or a subclass of the `List`. You can determine which particular event occurred by checking the `id` variable of the event.

public static final int LIST_SELECT

The peers deliver the `LIST_SELECT` event when the user selects an item in a `List`. If you are using the 1.1 event model, the interface method `ItemListener.itemStateChanged()` handles this event.

public static final int LIST_DESELECT

The peers deliver the `LIST_DESELECT` event when an item in a `List` has been deselected. This is generated only if the `List` permits multiple selections. If you are using the 1.1 event model, the `ItemListener.itemStateChanged()` interface method handles this event.

Focus events

The peers deliver focus events when a component gains (`GOT_FOCUS`) or loses (`LOST_FOCUS`) the input focus. No default methods are called for the focus events. They must be dealt with in the `handleEvent()` method of the `Container` of the component or a subclass of the component. You can determine which particular event occurred by checking the `id` variable of the event.

NOTE:

Early releases of Java (1.0.2 and before) did not propagate focus events on all platforms. This is fixed in release 1.1 of Java. Still, you should avoid capturing focus events if you want to write portable 1.0 code.

public static final int GOT_FOCUS

> The peers deliver the `GOT_FOCUS` event when a component gets the input focus. If you are using the 1.1 event model, the `FocusListener.focusGained()` interface method handles this event.

public static final int LOST_FOCUS

> The peers deliver the `LOST_FOCUS` event when a component loses the input focus. If you are using the 1.1 event model, the `FocusListener.focusLost()` interface method handles this event.

FileDialog events

The `FileDialog` events are another set of nonportable events. Ordinarily, the `FileDialog` events are completely dealt with by the system, and you never see them. Refer to Chapter 6, *Containers* for exactly how to work with the `FileDialog` object. If you decide to create a generic `FileDialog` object, you can use these events to indicate file loading and saving. These constants would be used in the `id` variable of the specific event instance:

public static final int LOAD_FILE
public static final int SAVE_FILE

Miscellaneous events

`ACTION_EVENT` is probably the event you deal with most frequently. It is generated when the user performs the desired action for a specific component type (e.g., when a user selects a button or toggles a checkbox). This constant would be found in the `id` variable of the specific event instance.

public static final int ACTION_EVENT

> The circumstances that lead to the peers delivering the `ACTION_EVENT` event depend upon the component that is the target of the event and the user's platform. Although the event can be passed along differently on different platforms, users will be accustomed to how the peers work on their specific platforms and will not care that it is different on the other platforms. For example, a Java 1.0 `List` component on a Microsoft Windows platform allows the user to select an item by

pressing the first letter of the choice, whereupon the `List` tries to find an item that starts with the letter. The X Window System `List` component does not provide this capability. It works like a normal X `List`, where the user must scroll to locate the item and then select it.

When the `ACTION_EVENT` is generated, the `arg` variable of the specific `Event` instance is set based upon the component type. In Chapters 5-11, which describe Java's GUI components, the description of each component contains an "Events" subsection that describes the value of the event's `arg` field. If you are using the 1.1 event model, the `ActionListener.actionPerformed()` and `ItemListener.itemStateChanged()` interface methods handle this event, depending upon the component type.

Event Methods

Constructors

Ordinarily, the peers deliver all your events for you. However, if you are creating your own components or want to communicate across threads, it may be necessary to create your own events. You can also create your own events to notify your component's container of application-specific occurrences. For example, if you were implementing your own tab sequencing for text fields, you could create a "next text field" event to tell your container to move to the next text field. Once you create the event, you send it through the system using the `Component.postEvent()` method.

public Event (Object target, long when, int id, int x, int y, int key, int modifiers, Object arg)

The first version of the constructor is the most complete and is what the other two call. It initializes all the fields of the `Event` to the parameters passed and sets `clickCount` to 0. See the descriptions of the instance variables Variables for the meanings of the arguments.

public Event (Object target, long when, int id, int x, int y, int key, int modifiers)

The second constructor version calls the first with `arg` set to null.

public Event (Object target, int id, Object arg)

The final version calls the first constructor with the `when`, `x`, `y`, `key`, and `modifiers` parameters set to 0.

Modifier methods

The modifier methods check to see if the different modifier mask values are set. They report the state of each modifier key at the moment an event occurred. It is possible for multiple masks to be set if multiple modifiers are pressed when the event occurs.

There is no `altDown()` method; to check whether the Alt key is pressed you must directly compare the event's `modifiers` against the `Event.ALT_MASK` constant. The `metaDown()` method is helpful when dealing with mouse events to see if the user pressed the right mouse button.

public boolean shiftDown ()

> The `shiftDown()` method returns `true` if the Shift key was pressed and `false` otherwise. There is no way to differentiate left and right shift keys.

public boolean controlDown ()

> The `controlDown()` method returns `true` if the Control key was pressed and `false` otherwise.

public boolean metaDown ()

> The `metaDown()` method returns `true` if the Meta key was pressed and `false` otherwise.

Miscellaneous methods

public void translate (int x, int y)

> The `translate()` method translates the x and y coordinates of the `Event` instance by `x` and `y`. The system does this so that the coordinates of the event are relative to the component receiving the event, rather than the container of the component. The system takes care of all this for you when passing the event through the containment hierarchy (not the object hierarchy), so you do not have to bother with translating them yourself. Figure 4.3 shows how this method would change the location of an event from a container down to an internal component.

Figure 4.3: Translating an event's location relative to a component

[Graphic: Figure 4-3]

protected String paramString ()

When you call the `toString()` method of `Event`, the `paramString()` method is called in turn to build the string to display. In the event you subclass `Event` to add additional information, instead of having to provide a whole new `toString()` method, you need only add the new information to the string already generated by `paramString()`. Assuming the new information is `foo`, this would result in the following method declaration:

```
protected String paramString() {
    return super.paramString() + ",foo=" + foo;
}
```

public String toString ()

The `toString()` method of `Event` returns a string with numerous components. The only variables that will always be in the output will be the event ID and the x and y coordinates. The others will be present if necessary (i.e., non-null): key (as the integer corresponding to a keyboard event), shift when `shiftDown()` is true; control, when `controlDown()` is true; meta, when `metaDown()` is true; target (if it was a `Component`); and arg (the value depends on the target and ID). `toString()` does not display all pieces of the `Event` information. An event when moving a `Scrollbar` might result in the following:

```
java.awt.Event[id=602,x=374,y=110,target=java.awt.Scrollbar[374,
110,15x50,val=1,vis=true,min=0,max=255,vert],arg=1]
```

Working With Mouse Buttons in Java 1.0

As stated earlier, the `modifiers` component of `Event` can be used to differentiate the different mouse buttons. If the user has a multibutton mouse, the `modifiers` field is set automatically to indicate which button was pressed. If the user does not own a multibutton mouse, he or she can press the mouse button in combination with the Alt or Meta keys to simulate a three-button mouse. Example 4.2 is a sample program called `mouseEvent` that displays the mouse button selected.

Example 4.2: Differentiating Mouse Buttons in Java 1.0

```
import java.awt.*;
import java.applet.*;
public class mouseEvent extends Applet {
    String theString = "Press a Mouse Key";
    public synchronized void setString (String s) {
        theString = s;
    }
    public synchronized String getString () {
        return theString;
    }
    public synchronized void paint (Graphics g) {
        g.drawString (theString, 20, 20);
    }
    public boolean mouseDown (Event e, int x, int y) {
        if (e.modifiers == Event.META_MASK) {
            setString ("Right Button Pressed");
        } else if (e.modifiers == Event.ALT_MASK) {
            setString ("Middle Button Pressed");
        } else {
            setString ("Left Button Pressed");
        }
        repaint ();
        return true;
    }
    public boolean mouseUp (Event e, int x, int y) {
        setString ("Press a Mouse Key");
        repaint ();
        return true;
    }
}
```

Unfortunately, this technique does not always work. With certain components on some platforms, the peer captures the mouse event and does not pass it along; for example, on Windows, the display-edit menu of a `TextField` appears when you select the right mouse button. Be cautious about relying on multiple mouse buttons; better yet, if you want to ensure absolute portability, stick to a single button.

Comprehensive Event List

Unfortunately, there are many platform-specific differences in the way event handling works. It's not clear whether these differences are bugs or whether vendors think they are somehow improving their product by introducing portability problems. We hope that as Java matures, different platforms will gradually come into synch. Until that happens, you might want your programs to assume the lowest common denominator. If you are willing to take the risk, you can program for a specific browser or platform, but should be aware of the possibility of changes.

Appendix C, *Platform-Specific Event Handling*, includes a table that shows which components pass along which events by default in the most popular environments. This table was developed using an interactive program called `compList`, which generates a list of supported events for each component. You can find `compList` on this book's Web site, http://www.ora.com/catalog/javawt. If you want to check the behavior of some new platform, or a newer version of one of the platforms in Appendix C, *Platform-Specific Event Handling*, feel free to use `compList`. It does require a little bit of work on your part. You have to click, toggle, type, and mouse over every object. Hopefully, as Java matures, this program will become unnecessary.

4.3 The Java 1.1 Event Model

Now it's time to discuss the new event model that is implemented by the 1.1 release of the JDK. Although this model can seem much more complex (it does have many more pieces), it is really much simpler and more efficient. The new event model does away with the process of searching for components that are interested in an event--`deliverEvent()`, `postEvent()`, `handleEvent()`--and all that. The new model requires objects be registered to receive events. Then, only those objects that are registered are told when the event actually happens.

This new model is called "delegation"; it implements the `Observer`-`Observable` design pattern with events. It is important in many respects. In addition to being much more efficient, it allows for a much cleaner separation between GUI components and event handling. It is important that any object, not just a `Component`, can receive events. Therefore, you can separate your event-handling code from your GUI code. One set of classes can implement the user interface; another set of classes can respond to the events generated by the interface. This means that if you have designed a good interface, you can reuse it in different applications by changing the event processing. The delegation model is essential to JavaBeans, which allows interaction between Java and other platforms, like OpenDoc or ActiveX. To allow such interaction, it was essential to separate the source of an event from the recipient.[1]

[1] For more information about JavaBeans, see http://splash.javasoft.com/beans/.

The delegation model has several other important ramifications. First, event handlers no longer need to worry about whether or not they have completely dealt with an event; they do what they need to, and return. Second, events can be broadcast to multiple recipients; any number of classes can be registered to receive an event. In the old model, broadcasting was possible only in a very limited sense, if at all. An event handler could declare that it hadn't completely processed an event, thus letting its container receive the event when it was done, or an event handler could generate a new event and deliver it to some other component. In any case, developers had to plan how to deliver events to other recipients. In Java 1.1, that's no longer necessary. An event will be delivered to every object that is registered as a listener for that event, regardless of what other objects do with the event. Any listener can mark an event "consumed," so it will be ignored by the peer or (if they care) other listeners.

Finally, the 1.1 event model includes the idea of an event queue. Instead of having to override `handleEvent()` to see all events, you can peek into the system's event queue by using the `EventQueue` class. The details of this class are discussed at the end of this chapter.

In Java 1.1, each component is an event *source* that can generate certain types of events, which are all subclasses of `AWTEvent`. Objects that are interested in an event are called *listeners*. Each event type corresponds to a listener interface that specifies the methods that are called when the event occurs. To receive an event, an object must implement the appropriate listener interface and must be registered with the event's source, by a call to an "add listener" method of the component that generates the event. Who calls the "add listener" method can vary; it is probably the best design for the component to register any listeners for the events that it generates, but it is also possible for the event handler to register itself, or for some third object to handle registration (for example, one object could call the constructor for a component, then call the constructor for an event handler, then register the event handler as a listener for the component's events).

This sounds complicated, but it really isn't that bad. It will help to think in concrete terms. A `TextField` object can generate action events, which in Java 1.1 are of the class `ActionEvent`. Let's say we have an object of class `TextActionHandler` that is called `myHandler` that is interested in receiving action events from a text field named `inputBuffer`. This means that our object must implement the `ActionListener` interface, and this in turn, means that it must include an `actionPerformed()` method, which is called when an action event occurs. Now, we have to register our object's interest in action events generated by `inputBuffer`; to do so, we need a call to `inputBuffer.addActionListener(myHandler)`. This call would probably be made by the object that is creating the `TextField` but could also be made by our event handler itself. The code might be as simple as this:

```
...
public void init(){
    ...
    inputBuffer = new TextField();
    myHandler = new TextActionHandler();
    inputBuffer.addActionListener(myHandler); // register the handler for the
                                              // buffer's events
    add (inputBuffer);  // add the input buffer to the display
    ...
}
```

Once our object has been registered, `myHandler.actionPerformed()` will be called whenever a user does anything in the text field that generates an action event, like typing a carriage return. In a way, `actionPerformed()` is very similar to the `action()` method of the old event model--except that it is not tied to the `Component` hierarchy; it is part of an interface that can be implemented by any object that cares about events.

Of course, there are many other kinds of events. Figure 4.4 shows the event hierarchy for Java 1.1. Figure 4.5 shows the different listener interfaces, which are all subinterfaces of `EventListener`, along with the related adapter classes.

Figure 4.4: AWTEvent class hierarchy

[Graphic: Figure 4-4]

Figure 4.5: AWT EventListener and Adapter class hierarchies

[Graphic: Figure 4-5]

Some of the listener interfaces are constructed to deal with multiple events. For instance, the `MouseListener` interface declares five methods to handle different kinds of mouse events: mouse down, mouse up, click (both down and up), mouse enter, and mouse exit. Strictly speaking, this means that an object interested in mouse events must implement `MouseListener` and must therefore implement five methods to deal with all possible mouse actions. This sounds like a waste of the programmer's effort; most of the time, you're only interested in one or two of these events. Why should you have to implement all five methods? Fortunately, you don't. The `java.awt.event` package also includes a set of *adapter classes*, which are shorthands that make it easier to write event handlers. The adapter class for any listener interface provides a `null` implementation of all the methods in that interface. For example, the `MouseAdapter` class provides `stub` implementations of the methods `mouseEntered()`, `mouseExited()`, `mousePressed()`, `mouseReleased()`, and `mouseClicked()`. If you want to write an event-handling class that deals with mouse clicks only, you can declare that your class extends `MouseAdapter`. It then inherits all five of these methods, and your only programming task is to override the single method you care about: `mouseClicked()`.

A particularly convenient way to use the adapters is to write an anonymous inner class. For example, the following code deals with the `MOUSE_PRESSED` event without creating a separate listener class:

```
addMouseListener (new MouseAdapter()   {
  public void mousePressed (MouseEvent e)  {
    // do what's needed to handle the event
    System.out.println ("Clicked at: " + e.getPoint());
  }
});
```

This code creates a `MouseAdapter`, overrides its `mousePressed()` method, and registers the resulting unnamed object as a listener for mouse events. Its `mousePressed()` method is called when `MOUSE_PRESSED` events occur. You can also use the adapter classes to implement something similar to a callback. For example, you could override `mousePressed()` to call one of your own methods, which would then be called whenever a `MOUSE_PRESSED` event occurs.

There are adapter classes for most of the listener interfaces; the only exceptions are the listener interfaces that contain only one method (for example, there's no `ActionAdapter` to go with `ActionListener`). When the listener interface contains only one method, an adapter class is superfluous. Event handlers may as well implement the listener interface directly, because they will have to override the only method in the interface; creating a dummy class with the interface method stubbed out doesn't accomplish anything. The different adapter classes are discussed with their related

EventListener interfaces.

With all these adapter classes, listener interfaces, and event classes, it's easy to get confused. Here's a quick summary of the different pieces involved and the roles they play:

- Components generate AWTEvents when something happens. Different subclasses of AWTEvent represent different kinds of events. For example, mouse events are represented by the MouseEvent class. Each component can generate certain subclasses of AWTEvent.

- Event handlers are registered to receive events by calls to an "add listener" method in the component that generates the event. There is a different "add listener" method for every kind of AWTEvent the component can generate; for example, to declare your interest in a mouse event, you call the component's addMouseListener() method.

- Every event type has a corresponding listener interface that defines the methods that are called when that event occurs. To be able to receive events, an event handler must therefore implement the appropriate listener interface. For example, MouseListener defines the methods that are called when mouse events occur. If you create a class that calls addMouseListener(), that class had better implement the MouseListener interface.

- Most event types also have an adapter class. For example, MouseEvents have a MouseAdapter class. The adapter class implements the corresponding listener interface but provides a stub implementation of each method (i.e., the method just returns without taking any action). Adapter classes are shorthand for programs that only need a few of the methods in the listener interface. For example, instead of implementing all five methods of the MouseListener interface, a class can extend the MouseAdapter class and override the one or two methods that it is interested in.

Using the 1.1 Event Model

Before jumping in and describing all the different pieces in detail, we will look at a simple applet that uses the Java 1.1 event model. Example 4.3 is equivalent to Example 4.2, except that it uses the new event model; when you press a mouse button, it just tells you what button you pressed. Notice how the new class, mouseEvent11, separates the user interface from the actual work. The class mouseEvent11 implements a very simple user interface. The class UpDownCatcher handles the events, figures out what to do, and calls some methods in mouseEvent11 to communicate the results. I added a simple interface that is called GetSetString to define the communications between the user interface and the event handler; strictly speaking, this isn't necessary, but it's a good programming practice.

Example 4.3: Handling Mouse Events in Java 1.1

```
// Java 1.1 only
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
interface GetSetString {
    public void setString (String s);
    public String getString ();
}
```

The `UpDownCatcher` class is responsible for handling events generated by the user interface. It extends `MouseAdapter` so that it needs to implement only the `MouseListener` methods that we care about (such as `mousePressed()` and `mouseReleased()`).

```
class UpDownCatcher extends MouseAdapter {
    GetSetString gss;
    public UpDownCatcher (GetSetString s) {
        gss = s;
    }
```

The constructor simply saves a reference to the class that is using this handler.

```
    public void mousePressed (MouseEvent e) {
        int mods = e.getModifiers();
        if ((mods & MouseEvent.BUTTON3_MASK) != 0) {
            gss.setString ("Right Button Pressed");
        } else if ((mods & MouseEvent.BUTTON2_MASK) != 0) {
            gss.setString ("Middle Button Pressed");
        } else {
            gss.setString ("Left Button Pressed");
        }
        e.getComponent().repaint();
    }
```

The `mousePressed` method overrides one of the methods of the `MouseAdapter` class. The method `mousePressed()` is called whenever a user presses any mouse button. This method figures out which button on a three-button mouse was pressed and calls the `setString()` method in the user interface to inform the user of the result.

```
    public void mouseReleased (MouseEvent e) {
        gss.setString ("Press a Mouse Key");
        e.getComponent().repaint();
    }
}
```

The `mouseReleased` method overrides another of the methods of the `MouseAdapter` class. When the user releases the mouse button, it calls `setString()` to restore the user interface to the original message.

```
public class mouseEvent11 extends Applet implements GetSetString {
    private String theString = "Press a Mouse Key";
    public synchronized void setString (String s) {
        theString = s;
    }
    public synchronized String getString () {
        return theString;
    }
    public synchronized void paint (Graphics g) {
        g.drawString (theString, 20, 20);
    }
    public void init () {
        addMouseListener (new UpDownCatcher(this));
    }
}
```

`mouseEvent11` is a very simple applet that implements our user interface. All it does is draw the desired string on the screen; the event handler tells it what string to draw. The `init()` method creates an instance of the event handler, which is `UpDownCatcher`, and registers it as interested in mouse events.

Because the user interface and the event processing are in separate classes, it would be easy to use this user interface for another purpose. You would have to replace only the `UpDownCatcher` class with something else--perhaps a more complex class that reported when the mouse entered and exited the area.

AWTEvent and Its Children

Under the 1.1 delegation event model, all system events are instances of `AWTEvent` or its subclasses. The model provides two sets of event types. The first set are fairly raw events, such as those indicating when a component gets focus, a key is pressed, or the mouse is moved. These events exist in `ComponentEvent` and its subclasses, along with some new events previously available only by overriding non-event-related methods. In addition, higher-level event types (for example, selecting a button) are encapsulated in other subclasses of `AWTEvent` that are not children of `ComponentEvent`.

AWTEvent

Variables

protected int id ★

> The `id` field of `AWTEvent` is protected and is accessible through the `getID()` method. It serves as the identifier of the event type, such as the `ACTION_PERFORMED` type of `ActionEvent` or the `MOUSE_MOVE` type of `Event`. With the delegation event model, it is usually not necessary to look at the event `id` unless you are looking in the event queue; just register the appropriate event listener.

Constants

The constants of `AWTEvent` are used in conjunction with the internal method `Component.eventEnabled()`. They are used to help the program determine what style of event handling (true/false--containment or listening--delegation) the program uses and which events a component processes. If you want to process 1.1 events without providing a listener, you need to set the mask for the type of event you want to receive. Look in Chapter 5, *Components*, for more information on the use of these constants:

public final static long ACTION_EVENT_MASK ★
public final static long ADJUSTMENT_EVENT_MASK ★
public final static long COMPONENT_EVENT_MASK ★

public final static long CONTAINER_EVENT_MASK ★
public final static long FOCUS_EVENT_MASK ★
public final static long ITEM_EVENT_MASK ★
public final static long KEY_EVENT_MASK ★
public final static long MOUSE_EVENT_MASK ★
public final static long MOUSE_MOTION_EVENT_MASK ★
public final static long TEXT_EVENT_MASK ★
public final static long WINDOW_EVENT_MASK ★

In addition to the mask constants, the constant `RESERVED_ID_MAX` is the largest event ID reserved for "official" events. You may use ID numbers greater than this value to create your own events, without risk of conflicting with standard events.

public final static long RESERVED_ID_MAX ★

Constructors

Since `AWTEvent` is an abstract class, you cannot call the constructors directly. They are automatically called when an instance of a child class is created.

public AWTEvent(Event event) ★

The first constructor creates an `AWTEvent` from the parameters of a 1.0 `Event`. The `event.target` and `event.id` are passed along to the second constructor.

public AWTEvent(Object source, int id) ★

This constructor creates an `AWTEvent` with the given `source`; the source is the object generating the event. The `id` field serves as the identifier of the event type. It is protected and is accessible through the `getID()` method. With the delegation event model, it is usually not necessary to look at the event `id` unless you are looking in the event queue or in the `processEvent()` method of a component; just register the appropriate event listener.

Methods

public int getID() ★

The `getID()` method returns the `id` from the constructor, thus identifying the event type.

protected void consume() ★

The `consume()` method is called to tell an event that it has been handled. An event that has been marked "consumed" is still delivered to the source component's peer and to all other registered listeners. However, the peer will ignore the event; other listeners may also choose to ignore it, but that's up to them. It isn't possible for a listener to "unconsume" an event that has already been marked "consumed."

Noncomponent events cannot be consumed. Only keyboard and mouse event types can be flagged as consumed. Marking an event "consumed" is useful if you are capturing keyboard input and need to reject a character; if you call `consume()`, the key event never makes it to the peer, and the keystroke isn't displayed. In Java 1.0, you would achieve the same effect by writing an event handler (e.g., `keyDown()`) that returns `true`.

You can assume that an event won't be delivered to the peer until all listeners have had a chance to consume it. However, you should not make any other assumptions about the order in which listeners are called.

protected boolean isConsumed() ★

The `isConsumed()` method returns whether the event has been consumed. If the event has been consumed, either by default or through `consume()`, this method returns `true`; otherwise, it returns `false`.

public String paramString() ★

When you call the `toString()` method of an `AWTEvent`, the `paramString()` method is called in turn to build the string to display. Since you are most frequently dealing with children of `AWTEvent`, the children need only to override `paramString()` to add their specific information.

public String toString() ★

The `toString()` method of `AWTEvent` returns a string with the name of the event, specific information about the event, and the source. In the method `MouseAdapter.mouseReleased()`, printing the parameter would result in something like the following:

```
java.awt.event.MouseEvent[MOUSE_RELEASED,(69,107),mods=0,clickCount=1] on panel1
```

ComponentEvent

Constants

public final static int COMPONENT_FIRST ★
public final static int COMPONENT_LAST ★

The `COMPONENT_FIRST` and `COMPONENT_LAST` constants hold the endpoints of the range of identifiers for `ComponentEvent` types.

public final static int COMPONENT_HIDDEN ★

The `COMPONENT_HIDDEN` constant identifies component events that occur because a component was hidden. The interface method `ComponentListener.componentHidden()` handles this event.

public final static int COMPONENT_MOVED ★

The `COMPONENT_MOVED` constant identifies component events that occur because a component has moved. The `ComponentListener.componentMoved()` interface method handles this event.

public final static int COMPONENT_RESIZED ★

The `COMPONENT_RESIZED` constant identifies component events that occur because a component has changed size. The interface method `ComponentListener.componentResized()` handles this event.

public final static int COMPONENT_SHOWN ★

The `COMPONENT_SHOWN` constant identifies component events that occur because a component has been shown (i.e., made visible). The interface method `ComponentListener.componentShown()` handles this event.

Constructors

public ComponentEvent(Component source, int id) ★

This constructor creates a `ComponentEvent` with the given `source`; the source is the object generating the event. The `id` field identifies the event type. If system generated, the `id` will be one of the last four constants above. However, nothing stops you from creating your own `id` for your event types.

Methods

public Component getComponent() ★

The `getComponent()` method returns the `source` of the event--that is, the component initiating the event.

public String paramString() ★

When you call the `toString()` method of an `AWTEvent`, the `paramString()` method is called in turn to build the string to display. At the `ComponentEvent` level, `paramString()` adds a string containing the event `id` (if available) and the bounding rectangle for the `source` (if appropriate). For example:

```
java.awt.event.ComponentEvent[COMPONENT_RESIZED (0, 0, 100x100)] on button0
```

ContainerEvent

The `ContainerEvent` class includes events that result from specific container operations. Constants

public final static int CONTAINER_FIRST ★
public final static int CONTAINER_LAST ★

The `CONTAINER_FIRST` and `CONTAINER_LAST` constants hold the endpoints of the range of identifiers for `ContainerEvent` types.

public final static int COMPONENT_ADDED ★

> The `COMPONENT_ADDED` constant identifies container events that occur because a component has been added to the container. The interface method `ContainerListener.componentAdded()` handles this event. Listening for this event is useful if a common listener should be attached to all components added to a container.

public final static int COMPONENT_REMOVED ★

> The `COMPONENT_REMOVED` constant identifies container events that occur because a component has been removed from the container. The interface method `ContainerListener.componentRemoved()` handles this event.

Constructors

public ContainerEvent(Container source, int id, Component child) ★

> The constructor creates a `ContainerEvent` with the given `source` (the container generating the event), to which the given `child` has been added or removed. The `id` field serves as the identifier of the event type. If system generated, the `id` will be one of the constants described previously. However, nothing stops you from creating your own `id` for your event types.

Methods

public Container getContainer() ★

> The `getContainer()` method returns the container that generated the event.

public Component getComponent() ★

> The `getComponent()` method returns the component that was added to or removed from the container.

public String paramString() ★

> When you call the `toString()` method of an `AWTEvent`, the `paramString()` method is in turn called to build the string to display. At the `ContainerEvent` level, `paramString()` adds a string containing the event `id` (if available) along with the name of the child.

FocusEvent

The `FocusEvent` class contains the events that are generated when a component gets or loses focus. These may be either temporary or permanent focus changes. A temporary focus change is the result of something else happening, like a window appearing in front of you. Once the window is removed, focus is restored. A permanent focus change is usually the result of focus traversal, using the keyboard or the mouse: for example, you clicked in a text field to type in it, or used Tab to move to the next component. More programmatically, permanent focus changes are the result of calls to `Component.requestFocus()`. Constants

public final static int FOCUS_FIRST ★
public final static int FOCUS_LAST ★

> The `FOCUS_FIRST` and `FOCUS_LAST` constants hold the endpoints of the range of identifiers for `FocusEvent` types.

public final static int FOCUS_GAINED ★

> The `FOCUS_GAINED` constant identifies focus events that occur because a component gains input focus. The `FocusListener.focusGained()` interface method handles this event.

public final static int FOCUS_LOST ★

> The `FOCUS_LOST` constant identifies focus events that occur because a component loses input focus. The `FocusListener.focusLost()` interface method handles this event.

Constructors

public FocusEvent(Component source, int id, boolean temporary) ★

> This constructor creates a `FocusEvent` with the given `source`; the source is the object generating the event. The `id` field serves as the identifier of the event type. If system generated, the `id` will be one of the two constants described previously. However, nothing stops you from creating your own `id` for your event types. The `temporary` parameter is `true` if this event represents a temporary focus change.

public FocusEvent(Component source, int id) ★

> This constructor creates a `FocusEvent` by calling the first constructor with the `temporary` parameter set to `false`; that is, it creates an event for a permanent focus change.

Methods

public boolean isTemporary() ★

> The `isTemporary()` method returns `true` if the focus event describes a temporary focus change, `false` if the event describes a permanent focus change. Once set by the constructor, the setting is permanent.

public String paramString() ★

> When you call the `toString()` method of an `AWTEvent`, the `paramString()` method is in turn called to build the string to display. At the `FocusEvent` level, `paramString()` adds a string showing the event `id` (if available) and whether or not it is temporary.

WindowEvent

The `WindowEvent` class encapsulates the window-oriented events. Constants

public final static int WINDOW_FIRST ★
public final static int WINDOW_LAST ★

The `WINDOW_FIRST` and `WINDOW_LAST` constants hold the endpoints of the range of identifiers for `WindowEvent` types.

public final static int WINDOW_ICONIFIED ★

The `WINDOW_ICONIFIED` constant identifies window events that occur because the user iconifies a window. The `WindowListener.windowIconified()` interface method handles this event.

public final static int WINDOW_DEICONIFIED ★

The `WINDOW_DEICONIFIED` constant identifies window events that occur because the user de-iconifies a window. The interface method `WindowListener.windowDeiconified()` handles this event.

public final static int WINDOW_OPENED ★

The `WINDOW_OPENED` constant identifies window events that occur the first time a `Frame` or `Dialog` is made visible with `show()`. The interface method `WindowListener.windowOpened()` handles this event.

public final static int WINDOW_CLOSING ★

The `WINDOW_CLOSING` constant identifies window events that occur because the user wants to close a window. This is similar to the familiar event `Event.WINDOW_DESTROY` dealt with under 1.0 with frames. The `WindowListener.windowClosing()` interface method handles this event.

public final static int WINDOW_CLOSED ★

The `WINDOW_CLOSED` constant identifies window events that occur because a `Frame` or `Dialog` has finally closed, after `hide()` or `destroy()`. This comes after `WINDOW_CLOSING`, which happens when the user wants the window to close. The `WindowListener.windowClosed()` interface method handles this event.

NOTE:

If there is a call to `System.exit()` in the `windowClosing()` listener, the window will not be around to call `windowClosed()`, nor will other listeners know.

public final static int WINDOW_ACTIVATED ★

The `WINDOW_ACTIVATED` constant identifies window events that occur because the user brings the window to the front, either after showing the window, de-iconifying, or removing whatever was in front. The interface method `WindowListener.windowActivated()` handles this event.

public final static int WINDOW_DEACTIVATED ★

> The `WINDOW_DEACTIVATED` constant identifies window events that occur because the user makes another window the active window. The interface method `WindowListener.windowDeactivated()` handles this event.

Constructors

public WindowEvent(Window source, int id) ★

> This constructor creates a `WindowEvent` with the given `source`; the source is the object generating the event. The `id` field serves as the identifier of the event type. If system generated, the `id` will be one of the seven constants described previously. However, nothing stops you from creating your own `id` for your event types.

Methods

public Window getWindow() ★

> The `getWindow()` method returns the `Window` that generated the event.

public String paramString() ★

> When you call the `toString()` method of an `AWTEvent`, the `paramString()` method is in turn called to build the string to display. At the `WindowEvent` level, `paramString()` adds a string containing the event `id` (if available). In a call to `windowClosing()`, printing the parameter would yield:

```
java.awt.event.WindowEvent[WINDOW_CLOSING] on frame0
```

PaintEvent

The `PaintEvent` class encapsulates the paint-oriented events. There is no corresponding `PaintListener` class, so you cannot listen for these events. To process them, override the `paint()` and `update()` routines of `Component`. The `PaintEvent` class exists to ensure that events are serialized properly through the event queue.
Constants

public final static int PAINT_FIRST ★
public final static int PAINT_LAST ★

> The `PAINT_FIRST` and `PAINT_LAST` constants hold the endpoints of the range of identifiers for `PaintEvent` types.

public final static int PAINT ★

The `PAINT` constant identifies paint events that occur because a component needs to be repainted. Override the `Component.paint()` method to handle this event.

public final static int UPDATE ★

The `UPDATE` constant identifies paint events that occur because a component needs to be updated before painting. This usually refreshes the display. Override the `Component.update()` method to handle this event.

Constructors

public PaintEvent(Component source, int id, Rectangle updateRect) ★

This constructor creates a `PaintEvent` with the given `source`. The source is the object whose display needs to be updated. The `id` field identifies the event type. If system generated, the `id` will be one of the two constants described previously. However, nothing stops you from creating your own `id` for your event types. `updateRect` represents the rectangular area of `source` that needs to be updated.

Methods

public Rectangle getUpdateRect()

The `getUpdateRect()` method returns the rectangular area within the `PaintEvent`'s source component that needs repainting. This area is set by either the constructor or the `setUpdateRect()` method.

public void setUpdateRect(Rectangle updateRect)

The `setUpdateRect()` method changes the area of the `PaintEvent`'s source component that needs repainting.

public String paramString() ★

When you call the `toString()` method of an `AWTEvent`, the `paramString()` method is called in turn to build the string to display. At the `PaintEvent` level, `paramString()` adds a string containing the event `id` (if available) along with the area requiring repainting (a clipping rectangle). If you peek in the event queue, one possible result may yield:

```
java.awt.event.PaintEvent[PAINT,updateRect=java.awt.Rectangle[x=0,y=0,
width=192,height=173]] on frame0
```

InputEvent

The `InputEvent` class provides the basis for the key and mouse input and movement routines. `KeyEvent` and `MouseEvent` provide the specifics of each. Constants

The constants of `InputEvent` help identify which modifiers are present when an input event occurs, as shown in

Example 4.3. To examine the event modifiers and test for the presence of these masks, call `getModifiers()` to get the current set of modifiers.

public final static int ALT_MASK ★
public final static int CTRL_MASK ★
public final static int META_MASK ★
public final static int SHIFT_MASK ★

> The first set of `InputEvent` masks are for the different modifier keys on the keyboard. They are often set to indicate which button on a multibutton mouse has been pressed.

public final static int BUTTON1_MASK ★
public final static int BUTTON2_MASK ★
public final static int BUTTON3_MASK ★

> The button mask constants are equivalents for the modifier masks, allowing you to write more intelligible code for dealing with button events. `BUTTON2_MASK` is the same as `ALT_MASK`, and `BUTTON3_MASK` is the same as `META_MASK`; `BUTTON1_MASK` currently isn't usable and is never set. For example, if you want to check whether the user pressed the second (middle) mouse button, you can test against `BUTTON2_MASK` rather than `ALT_MASK`. Example 4.3 demonstrates how to use these constants.

Constructors

`InputEvent` is an abstract class with no public constructors. Methods

Unlike the `Event` class, `InputEvent` has an `isAltDown()` method to check the `ALT_MASK` setting.

public boolean isAltDown() ★

> The `isAltDown()` method checks to see if `ALT_MASK` is set. If so, `isAltDown()` returns `true`; otherwise, it returns `false`.

public boolean isControlDown() ★

> The `isControlDown()` method checks to see if `CONTROL_MASK` is set. If so, `isControlDown()` returns `true`; otherwise, it returns `false`.

public boolean isMetaDown() ★

> The `isMetaDown()` method checks to see if `META_MASK` is set. If so, the method `isMetaDown()` returns `true`; otherwise, it returns `false`.

public boolean isShiftDown() ★

The `isShiftDown()` method checks to see if `SHIFT_MASK` is set. If so, the method `isShiftDown()` returns `true`; otherwise, it returns `false`.

public int getModifiers() ★

The `getModifiers()` method returns the current state of the modifier keys. For each modifier key pressed, a different flag is raised in the return argument. To check if a modifier is set, AND the return value with a flag and check for a nonzero value.

```
if ((ie.getModifiers() & MouseEvent.META_MASK) != 0) {
    System.out.println ("Meta is set");
}
```

public long getWhen() ★

The `getWhen()` method returns the time at which the event occurred. The return value is in milliseconds. Convert the `long` value to a `Date` to examine the contents. For example:

```
Date d = new Date (ie.getWhen());
```

public void consume() ★

This class overrides the `AWTEvent.consume()` method to make it public. Anyone, not just a subclass, can mark an `InputEvent` as consumed.

public boolean isConsumed() ★

This class overrides the `AWTEvent.isconsumed()` method to make it public. Anyone can find out if an `InputEvent` has been consumed.

KeyEvent

The `KeyEvent` class is a subclass of `InputEvent` for dealing with keyboard events. There are two fundamental key actions: key presses and key releases. These are represented by `KEY_PRESSED` and `KEY_RELEASED` events. Of course, it's inconvenient to think in terms of all these individual actions, so Java also keeps track of the "logical" keys you type. These are represented by `KEY_TYPED` events. For every keyboard key pressed, a `KeyEvent.KEY_PRESSED` event occurs; the key that was pressed is identified by one of the virtual keycodes from Table 4.4 and is available through the `getKeyCode()` method. For example, if you type an uppercase A, you will get two `KEY_PRESSED` events, one for shift (`VK_SHIFT`) and one for the "a" (`VK_A`). You will also get two `KeyEvent.KEY_RELEASED` events. However, there will only be one `KeyEvent.KEY_TYPED` event; if you call `getKeyChar()` for the `KEY_TYPED` event, the result will be the Unicode character "A" (type `char`). `KEY_TYPED` events do not happen for action-oriented keys like function keys. Constants

Like the `Event` class, numerous constants help you identify all the keyboard keys. Table 4.4 shows the constants that refer to these keyboard keys. The values are all declared `public static final int`. A few keys represent ASCII characters that have string equivalents like `\n`.

Table 4.4: Key Constants in Java 1.1

VK_ENTER	VK_0	VK_A	VK_F1	VK_ACCEPT
VK_BACK_SPACE	VK_1	VK_B	VK_F2	VK_CONVERT
VK_TAB	VK_2	VK_C	VK_F3	VK_FINAL
VK_CANCEL	VK_3	VK_D	VK_F4	VK_KANA
VK_CLEAR	VK_4	VK_E	VK_F5	VK_KANJI
VK_SHIFT	VK_5	VK_F	VK_F6	VK_MODECHANGE
VK_CONTROL	VK_6	VK_G	VK_F7	VK_NONCONVERT
VK_ALT	VK_7	VK_H	VK_F8	
VK_PAUSE	VK_8	VK_I	VK_F9	
VK_CAPS_LOCK	VK_9	VK_J	VK_F10	
VK_ESCAPE	VK_NUMPAD0	VK_K	VK_F11	
VK_SPACE	VK_NUMPAD1	VK_L	VK_F12	
VK_PAGE_UP	VK_NUMPAD2	VK_M	VK_DELETE	
VK_PAGE_DOWN	VK_NUMPAD3	VK_N	VK_NUM_LOCK	
VK_END	VK_NUMPAD4	VK_O	VK_SCROLL_LOCK	
VK_HOME	VK_NUMPAD5	VK_P	VK_PRINTSCREEN	
VK_LEFT	VK_NUMPAD6	VK_Q	VK_INSERT	
VK_UP	VK_NUMPAD7	VK_R	VK_HELP	
VK_RIGHT	VK_NUMPAD8	VK_S	VK_META	
VK_DOWN	VK_NUMPAD9	VK_T	VK_BACK_QUOTE	
VK_COMMA	VK_MULTIPLY	VK_U	VK_QUOTE	
VK_PERIOD	VK_ADD	VK_V	VK_OPEN_BRACKET	
VK_SLASH	VK_SEPARATER[1]	VK_W	VK_CLOSE_BRACKET	
VK_SEMICOLON	VK_SUBTRACT	VK_X		
VK_EQUALS	VK_DECIMAL	VK_Y		
VK_BACK_SLASH	VK_DIVIDE	VK_Z		

Footnotes:

[1] Expect VK_SEPARATOR to be added at some future point. This constant represents the numeric separator key on your keyboard.

public final static int VK_UNDEFINED ★

When a KEY_TYPED event happens, there is no keycode. If you ask for it, the getKeyCode() method returns VK_UNDEFINED.

public final static char CHAR_UNDEFINED ★

> For `KEY_PRESSED` and `KEY_RELEASED` events that do not have a corresponding Unicode character to display (like Shift), the `getKeyChar()` method returns `CHAR_UNDEFINED`.

Other constants identify what the user did with a key.

public final static int KEY_FIRST ★
public final static int KEY_LAST ★

> The `KEY_FIRST` and `KEY_LAST` constants hold the endpoints of the range of identifiers for `KeyEvent` types.

public final static int KEY_PRESSED ★

> The `KEY_PRESSED` constant identifies key events that occur because a keyboard key has been pressed. To differentiate between action and non-action keys, call the `isActionKey()` method described later. The `KeyListener.keyPressed()` interface method handles this event.

public final static int KEY_RELEASED ★

> The `KEY_RELEASED` constant identifies key events that occur because a keyboard key has been released. The `KeyListener.keyReleased()` interface method handles this event.

public final static int KEY_TYPED ★

> The `KEY_TYPED` constant identifies a combination of a key press followed by a key release for a non-action oriented key. The `KeyListener.keyTyped()` interface method handles this event.

Constructors

public KeyEvent(Component source, int id, long when, int modifiers, int keyCode, char keyChar) ★

> This constructor[2] creates a `KeyEvent` with the given `source`; the source is the object generating the event. The `id` field identifies the event type. If system-generated, the `id` will be one of the constants above. However, nothing stops you from creating your own `id` for your event types. The `when` parameter represents the time the event happened. The `modifiers` parameter holds the state of the various modifier keys; masks to represent these keys are defined in the `InputEvent` class. Finally, `keyCode` is the virtual key that triggered the event, and `keyChar` is the character that triggered it.
>
> > [2] Beta releases of Java 1.1 have an additional constructor that lacks the `keyChar` parameter. Comments in the code indicate that this constructor will be deleted prior to the 1.1.1 release.
>
> The `KeyEvent` constructor throws the `IllegalArgumentException` run-time exception in two situations. First, if the `id` is `KEY_TYPED` and `keyChar` is `CHAR_UNDEFINED`, it throws an exception because if a key has been typed, it must be associated with a character. Second, if the `id` is `KEY_TYPED` and `keyCode` is not `VK_UNDEFINED`, it throws an exception because typed keys frequently represent

combinations of key codes (for example, Shift struck with "a"). It is legal for a `KEY_PRESSED` or `KEY_RELEASED` event to contain both a `keyCode` and a `keyChar`, though it's not clear what such an event would represent.

Methods

public char getKeyChar() ★

The `getKeyChar()` method retrieves the Unicode character associated with the key in this `KeyEvent`. If there is no character, `CHAR_UNDEFINED` is returned.

public void setKeyChar(char KeyChar) ★

The `setKeyChar()` method allows you to change the character for the `KeyEvent`. You could use this method to convert characters to uppercase.

public int getKeyCode() ★

The `getKeyCode()` method retrieves the virtual keycode (i.e., one of the constants in Table 4.4) of this `KeyEvent`.

public void setKeyCode(int keyCode) ★

The `setKeyCode()` method allows you to change the keycode for the `KeyEvent`. Changes you make to the `KeyEvent` are seen by subsequent listeners and the component's peer.

public void setModifiers(int modifiers) ★

The `setModifiers()` method allows you to change the modifier keys associated with a `KeyEvent` to `modifiers`. The parent class `InputEvent` already has a `getModifiers()` method that is inherited. Since this is your own personal copy of the `KeyEvent`, no other listener can find out about the change.

public boolean isActionKey() ★

The `isActionKey()` method allows you to check whether the key associated with the `KeyEvent` is an action key (e.g., function, arrow, keypad) or not (e.g., an alphanumeric key). For action keys, this method returns `true`; otherwise, it returns `false`. For action keys, the `keyChar` field usually has the value `CHAR_UNDEFINED`.

public static String getKeyText (int keyCode) ★

The static `getKeyText()` method returns the localized textual string for `keyCode`. For each nonalphanumeric virtual key, there is a key name (the "key text"); these names can be changed using the AWT properties. Table 4.5 shows the properties used to redefine the key names and the default name for each key.

Table 4.5: Key Text Properties

Property	Default	Property	Default
AWT.accept	Accept	AWT.f8	F8
AWT.add	NumPad +	AWT.f9	F9
AWT.alt	Alt	AWT.help	Help
AWT.backQuote	Back Quote	AWT.home	Home
AWT.backSpace	Backspace	AWT.insert	Insert
AWT.cancel	Cancel	AWT.kana	Kana
AWT.capsLock	Caps Lock	AWT.kanji	Kanji
AWT.clear	Clear	AWT.left	Left
AWT.control	Control	AWT.meta	Meta
AWT.decimal	NumPad .	AWT.modechange	Mode Change
AWT.delete	Delete	AWT.multiply	NumPad *
AWT.divide	NumPad /	AWT.noconvert	No Convert
AWT.down	Down	AWT.numLock	Num Lock
AWT.end	End	AWT.numpad	NumPad
AWT.enter	Enter	AWT.pause	Pause
AWT.escape	Escape	AWT.pgdn	Page Down
AWT.final	Final	AWT.pgup	Page Up
AWT.f1	F1	AWT.printScreen	Print Screen
AWT.f10	F10	AWT.quote	Quote
AWT.f11	F11	AWT.right	Right
AWT.f12	F12	AWT.scrollLock	Scroll Lock
AWT.f2	F2	AWT.separator	NumPad ,
AWT.f3	F3	AWT.shift	Shift
AWT.f4	F4	AWT.space	Space
AWT.f5	F5	AWT.subtract	NumPad -
AWT.f6	F6	AWT.tab	Tab
AWT.f7	F7	AWT.unknown	Unknown keyCode
AWT.up	Up		

public static String getKeyModifiersText (int modifiers) ★

The static getKeyModifiersText() method returns the localized textual string for modifiers. The parameter modifiers is a combination of the key masks defined by the InputEvent class. As with the keys themselves, each modifier is associated with a textual name. If multiple modifiers are set, they are concatenated with a plus sign (+) separating them. Similar to getKeyText(), the strings are localized because for each modifier, an awt property is available to redefine the string. Table 4.6 lists the properties and the default modifier names.

Table 4.6: Key Modifiers Text Properties

Property	Default
`AWT.alt`	Alt
`AWT.control`	Ctrl
`AWT.meta`	Meta
`AWT.shift`	Shift

public String paramString() ★

When you call the `toString()` method of an `AWTEvent`, the `paramString()` method is called in turn to build the string to display. At the `KeyEvent` level, `paramString()` adds a textual string for the `id` (if available), the text for the key (if available from `getKeyText()`), and modifiers (from `getKeyModifiersText()`). A key press event would result in something like the following:

```
java.awt.event.KeyEvent[KEY_PRESSED,keyCode=118,
F7,modifiers=Ctrl+Shift] on textfield0
```

MouseEvent

The `MouseEvent` class is a subclass of `InputEvent` for dealing with mouse events. Constants

public final static int MOUSE_FIRST ★
public final static int MOUSE_LAST ★

The `MOUSE_FIRST` and `MOUSE_LAST` constants hold the endpoints of the range of identifiers for `MouseEvent` types.

public final static int MOUSE_CLICKED ★

The `MOUSE_CLICKED` constant identifies mouse events that occur when a mouse button is clicked. A mouse click consists of a mouse press and a mouse release. The `MouseListener.mouseClicked()` interface method handles this event.

public final static int MOUSE_DRAGGED ★

The `MOUSE_DRAGGED` constant identifies mouse events that occur because the mouse is moved over a component with a mouse button pressed. The interface method `MouseMotionListener.mouseDragged()` handles this event.

public final static int MOUSE_ENTERED ★

The `MOUSE_ENTERED` constant identifies mouse events that occur when the mouse first enters a component. The `MouseListener.mouseEntered()` interface method handles this event.

public final static int MOUSE_EXITED ★

The `MOUSE_EXISTED` constant identifies mouse events that occur because the mouse leaves a component's space. The `MouseListener.mouseExited()` interface method handles this event.

public final static int MOUSE_MOVED ★

The `MOUSE_MOVED` constant identifies mouse events that occur because the mouse is moved without a mouse button down. The interface method `MouseMotionListener.mouseMoved()` handles this event.

public final static int MOUSE_PRESSED ★

The `MOUSE_PRESSED` constant identifies mouse events that occur because a mouse button has been pressed. The `MouseListener.mousePressed()` interface method handles this event.

public final static int MOUSE_RELEASED ★

The `MOUSE_RELEASED` constant identifies mouse events that occur because a mouse button has been released. The `MouseListener.mouseReleased()` interface method handles this event.

Constructors

public MouseEvent(Component source, int id, long when, int modifiers, int x, int y, int clickCount, boolean popupTrigger) ★

This constructor creates a `MouseEvent` with the given `source`; the source is the object generating the event. The `id` field serves as the identifier of the event type. If system-generated, the `id` will be one of the constants described in the previous section. However, nothing stops you from creating your own `id` for your event types. The `when` parameter represents the time the event happened. The `modifiers` parameter holds the state of the various modifier keys, using the masks defined for the `InputEvent` class, and lets you determine which button was pressed. (`x`, `y`) represents the coordinates of the event relative to the origin of `source`, while `clickCount` designates the number of consecutive times the mouse button was pressed within an indeterminate time period. Finally, the `popupTrigger` parameter signifies whether this mouse event should trigger the display of a `PopupMenu`, if one is available. (The `PopupMenu` class is discussed in Chapter 10, *Would You Like to Choose from the Menu?*)

Methods

public int getX() ★

The `getX()` method returns the current x coordinate of the event relative to the source.

public int getY() ★

The `getY()` method returns the current y coordinate of the event relative to the source.

public synchronized Point getPoint() ★

The `getPoint()` method returns the current x and y coordinates of the event relative to the event source.

public synchronized void translatePoint(int x, int y) ★

The `translatePoint()` method translates the x and y coordinates of the `MouseEvent` instance by `x` and `y`. This method functions similarly to the `Event.translate()` method.

public int getClickCount() ★

The `getClickCount()` method retrieves the current `clickCount` setting for the event.

public boolean isPopupTrigger() ★

The `isPopupTrigger()` method retrieves the state of the `popupTrigger` setting for the event. If this method returns `true` and the source of the event has an associated `PopupMenu`, the event should be used to display the menu, as shown in the following code. Since the action the user performs to raise a pop-up menu is platform specific, this method lets you raise a pop-up menu without worrying about what kind of event took place. You only need to call `isPopupTrigger()` and show the menu if it returns `true`.

```
public void processMouseEvent(MouseEvent e) {
    if (e.isPopupTrigger())
        aPopup.show(e.getComponent(), e.getX(), e.getY());
    super.processMouseEvent(e);
}
```

public String paramString() ★

When you call the `toString()` method of an `AWTEvent`, the `paramString()` method is called in turn to build the string to display. At the `MouseEvent` level, a textual string for the `id` (if available) is tacked on to the coordinates, modifiers, and click count. A mouse down event would result in something like the following:

```
java.awt.event.MouseEvent[MOUSE_PRESSED,(5,7),mods=0,clickCount=2] on textfield0
```

ActionEvent

The `ActionEvent` class is the first higher-level event class. It encapsulates events that signify that the user is doing something with a component. When the user selects a button, list item, or menu item, or presses the Return key in a text field, an `ActionEvent` passes through the event queue looking for listeners. Constants

public final static int ACTION_FIRST ★
public final static int ACTION_LAST ★

> The `ACTION_FIRST` and `ACTION_LAST` constants hold the endpoints of the range of identifiers for `ActionEvent` types.

public final static int ACTION_PERFORMED ★

> The `ACTION_PERFORMED` constant represents when a user activates a component. The `ActionListener.actionPerformed()` interface method handles this event.

public static final int ALT_MASK ★
public static final int CTRL_MASK ★
public static final int META_MASK ★
public static final int SHIFT_MASK ★

> Similar to the mouse events, action events have `modifiers`. However, they are not automatically set by the system, so they don't help you see what modifiers were pressed when the event occurred. You may be able to use these constants if you are generating your own action events. To see the value of an action event's modifiers, call `getModifiers()`.

Constructors

public ActionEvent(Object source, int id, String command) ★

> This constructor creates an `ActionEvent` with the given `source`; the source is the object generating the event. The `id` field serves as the identifier of the event type. If system-generated, the `id` will be `ACTION_PERFORMED`. However, nothing stops you from creating your own `id` for your event types. The `command` parameter is the event's action command. Ideally, the action command should be some locale-independent string identifying the user's action. Most components that generate action events set this field to the selected item's label by default.

public ActionEvent(Object source, int id, String command, int modifiers) ★

> This constructor adds `modifiers` to the settings for an `ActionEvent`. This allows you to define action-oriented events that occur only if certain modifier keys are pressed.

Methods

public String getActionCommand() ★

> The `getActionCommand()` method retrieves the `command` field from the event. It represents the command associated with the object that triggered the event. The idea behind the action command is to differentiate the command associated with some event from the displayed content of the event source. For example, the action command for a button may be Help. However, what the user sees on the label of the button could be a string

localized for the environment of the user. Instead of having your event handler look for 20 or 30 possible labels, you can test whether an event has the action command Help.

public int getModifiers() ★

The `getModifiers()` method returns the state of the modifier keys. For each one set, a different flag is raised in the method's return value. To check if a modifier is set, AND the return value with a flag, and check for a nonzero value.

public String paramString() ★

When you call the `toString()` method of an `AWTEvent`, the `paramString()` method is called in turn to build the string to display. At the `ActionEvent` level, `paramString()` adds a textual string for the event `id` (if available), along with the `command` from the constructor. When the user selects a `Button` with the action command Help, printing the resulting event yields:

```
java.awt.event.ActionEvent[ACTION_PERFORMED,cmd=Help] on button0
```

AdjustmentEvent

The `AdjustmentEvent` class is another higher-level event class. It encapsulates events that represent scrollbar motions. When the user moves the slider of a scrollbar or scroll pane, an `AdjustmentEvent` passes through the event queue looking for listeners. Although there is only one type of adjustment event, there are five subtypes represented by constants `UNIT_DECREMENT`, `UNIT_INCREMENT`, and so on. Constants

public final static int ADJUSTMENT_FIRST ★
public final static int ADJUSTMENT_LAST ★

The `ADJUSTMENT_FIRST` and `ADJUSTMENT_LAST` constants hold the endpoints of the range of identifiers for `AdjustmentEvent` types.

public final static int ADJUSTMENT_VALUE_CHANGED ★

The `ADJUSTMENT_VALUE_CHANGED` constant identifies adjustment events that occur because a user moves the slider of a `Scrollbar` or `ScrollPane`. The `AdjustmentListener.adjustmentValueChanged()` interface method handles this event.

public static final int UNIT_DECREMENT ★

`UNIT_DECREMENT` identifies adjustment events that occur because the user selects the increment arrow.

public static final int UNIT_INCREMENT ★

`UNIT_INCREMENT` identifies adjustment events that occur because the user selects the decrement arrow.

public static final int BLOCK_DECREMENT ★

> `BLOCK_DECREMENT` identifies adjustment events that occur because the user selects the block decrement area, between the decrement arrow and the slider.

public static final int BLOCK_INCREMENT ★

> `BLOCK_INCREMENT` identifies adjustment events that occur because the user selects the block increment area, between the increment arrow and the slider.

public static final int TRACK ★

> `TRACK` identifies adjustment events that occur because the user selects the slider and drags it. Multiple adjustment events of this subtype usually occur consecutively.

Constructors

public AdjustmentEvent(Adjustable source, int id, int type, int value) ★

> This constructor creates an `AdjustmentEvent` with the given `source`; the source is the object generating the event. The `id` field serves as the identifier of the event type. If system-generated, the `id` of the `AdjustmentEvent` will be `ADJUSTMENT_VALUE_CHANGED`. However, nothing stops you from creating your own `id` for your event types. The `type` parameter is normally one of the five subtypes, with `value` being the current setting of the slider, but is not restricted to that.

Methods

public Adjustable getAdjustable() ★

> The `getAdjustable()` method retrieves the `Adjustable` object associated with this event--that is, the event's `source`.

public int getAdjustmentType() ★

> The `getAdjustmentType()` method retrieves the `type` parameter from the constructor. It represents the subtype of the current event and, if system-generated, is one of the following constants: `UNIT_DECREMENT`, `UNIT_INCREMENT`, `BLOCK_DECREMENT`, `BLOCK_INCREMENT`, or `TRACK`.

public int getValue() ★

> The `getValue()` method retrieves the `value` parameter from the constructor. It represents the current setting of the adjustable object.

public String paramString() ★

When you call the `toString()` method of an `AWTEvent`, the `paramString()` method is called to help build the string to display. At the `AdjustableEvent` level, `paramString()` adds a textual string for the event `id` (if available), along with a textual string of the `type` (if available), and `value`. For example:

```
java.awt.event.AdjustableEvent[ADJUSTMENT_VALUE_CHANGED,
adjType=TRACK,value=27] on scrollbar0
```

ItemEvent

The `ItemEvent` class is another higher-level event class. It encapsulates events that occur when the user selects a component, like `ActionEvent`. When the user selects a checkbox, choice, list item, or checkbox menu item, an `ItemEvent` passes through the event queue looking for listeners. Although there is only one type of `ItemEvent`, there are two subtypes represented by the constants `SELECTED` and `DESELECTED`. Constants

public final static int ITEM_FIRST ★
public final static int ITEM_LAST ★

The `ITEM_FIRST` and `ITEM_LAST` constants hold the endpoints of the range of identifiers for `ItemEvent` types.

public final static int ITEM_STATE_CHANGED ★

The `ITEM_STATE_CHANGED` constant identifies item events that occur because a user selects a component, thus changing its state. The interface method `ItemListener.itemStateChanged()` handles this event.

public static final int SELECTED ★

`SELECTED` indicates that the user selected the item.

public static final int DESELECTED ★

`DESELECTED` indicates that the user deselected the item.

Constructors

public ItemEvent(ItemSelectable source, int id, Object item, int stateChange) ★

This constructor creates a `ItemEvent` with the given `source`; the source is the object generating the event. The `id` field serves as the identifier of the event type. If system-generated, the `id` will be `ITEM_STATE_CHANGE`. However, nothing stops you from creating your own `id` for your event types. The `item` parameter represents the text of the item selected: for a `Checkbox`, this would be its label, for a `Choice` the current selection. For your own events, this parameter could be virtually anything, since its type is `Object`.

Methods

public ItemSelectable getItemSelectable() ★

The `getItemSelectable()` method retrieves the `ItemSelectable` object associated with this event--that is, the event's source.

public Object getItem() ★

The `getItem()` method returns the `item` that was selected. This usually represents some text to help identify the source but could be nearly anything for user-generated events.

public int getStateChange() ★

The `getStateChange()` method returns the `stateChange` parameter from the constructor and, if system generated, is either `SELECTED` or `DESELECTED`.

public String paramString() ★

When you call the `toString()` method of an `AWTEvent`, the `paramString()` method is called in turn to build the string to display. At the `ItemEvent` level, `paramString()` adds a textual string for the event `id` (if available), along with a textual string indicating the value of `stateChange` (if available) and `item`. For example:

```
java.awt.event.ItemEvent[ITEM_STATE_CHANGED,item=Help,
stateChange=SELECTED] on checkbox1
```

TextEvent

The `TextEvent` class is yet another higher-level event class. It encapsulates events that occur when the contents of a `TextComponent` have changed, although is not required to have a `TextComponent` source. When the contents change, either programmatically by a call to `setText()` or because the user typed something, a `TextEvent` passes through the event queue looking for listeners. Constants

public final static int TEXT_FIRST ★
public final static int TEXT_LAST ★

The `TEXT_FIRST` and `TEXT_LAST` constants hold the endpoints of the range of identifiers for `TextEvent` types.

public final static int TEXT_VALUE_CHANGED ★

The `TEXT_VALUE_CHANGED` constant identifies text events that occur because a user changes the contents of a text component. The interface method `TextListener.textValueChanged()` handles this event.

Constructors

public TextEvent(Object source, int id) ★

> This constructor creates a `TextEvent` with the given `source`; the source is the object generating the event. The `id` field identifies the event type. If system-generated, the `id` will be `TEXT_VALUE_CHANGE`. However, nothing stops you from creating your own `id` for your event types.

Method

public String paramString() ★

> When you call the `toString()` method of an `AWTEvent`, the `paramString()` method is called in turn to build the string to display. At the `TextEvent` level, `paramString()` adds a textual string for the event `id` (if available).

Event Listener Interfaces and Adapters

Java 1.1 has 11 event listener interfaces, which specify the methods a class must implement to receive different kinds of events. For example, the `ActionListener` interface defines the single method that is called when an `ActionEvent` occurs. These interfaces replace the various event-handling methods of Java 1.0: `action()` is now the `actionPerformed()` method of the `ActionListener` interface, `mouseUp()` is now the `mouseReleased()` method of the `MouseListener` interface, and so on. Most of the listener interfaces have a corresponding adapter class, which is an abstract class that provides a null implementation of all the methods in the interface. (Although an adapter class has no abstract methods, it is declared `abstract` to remind you that it must be subclassed.) Rather than implementing a listener interface directly, you have the option of extending an adapter class and overriding only the methods you care about. (Much more complex adapters are possible, but the adapters supplied with AWT are very simple.) The adapters are available for the listener interfaces with multiple methods. (If there is only one method in the listener interface, there is no need for an adapter.)

This section describes Java 1.1's listener interfaces and adapter classes. It's worth noting here that Java 1.1 does not allow you to modify the original event when you're writing an event handler.

ActionListener

The `ActionListener` interface contains the one method that is called when an `ActionEvent` occurs. It has no adapter class. For an object to listen for action events, it is necessary to call the `addActionListener()` method with the class that implements the `ActionListener` interface as the parameter. The method `addActionListener()` is implemented by `Button`, `List`, `MenuItem`, and `TextField` components. Other components don't generate action events.

public abstract void actionPerformed(ActionEvent e) ★

> The `actionPerformed()` method is called when a component is selected or activated. Every component is activated differently; for a `List`, activation means that the user has double-clicked on an entry. See the appropriate section for a description of each component.
>
> `actionPerformed()` is the Java 1.1 equivalent of the `action()` method in the 1.0 event model.

AdjustmentListener

The `AdjustmentListener` interface contains the one method that is called when an `AdjustmentEvent` occurs. It has no adapter class. For an object to listen for adjustment events, it is necessary to call `addAdjustmentListener()` with the class that implements the `AdjustmentListener` interface as the parameter. The `addAdjustmentListener()` method is implemented by the `Scrollbar` component and the `Adjustable` interface. Other components don't generate adjustment events.

public abstract void adjustmentValueChanged(AdjustmentEvent e) ★

> The `adjustmentValueChanged()` method is called when a slider is moved. The `Scrollbar` and `ScrollPane` components have sliders, and generate adjustment events when the sliders are moved. (The `TextArea` and `List` components also have sliders, but do not generate adjustment events.) See the appropriate section for a description of each component.
>
> There is no real equivalent to `adjustmentValueChanged()` in Java 1.0; to work with scrolling events, you had to override the `handleEvent()` method.

ComponentListener and ComponentAdapter

The `ComponentListener` interface contains four methods that are called when a `ComponentEvent` occurs; component events are used for general actions on components, like moving or resizing a component. The adapter class corresponding to `ComponentListener` is `ComponentAdapter`. If you care only about one or two of the methods in `ComponentListener`, you can subclass the adapter and override only the methods that you are interested in. For an object to listen for component events, it is necessary to call `Component.addComponentListener()` with the class that implements the interface as the parameter.

public abstract void componentResized(ComponentEvent e) ★

> The `componentResized()` method is called when a component is resized (for example, by a call to `Component.setSize()`).

public abstract void componentMoved(ComponentEvent e) ★

> The `componentMoved()` method is called when a component is moved (for example, by a call to `Component.setLocation()`).

public abstract void componentShown(ComponentEvent e) ★

> The `componentShown()` method is called when a component is shown (for example, by a call to `Component.show()`).

public abstract void componentHidden(ComponentEvent e) ★

> The `componentHidden()` method is called when a component is hidden (for example, by a call to `Component.hide()`).

ContainerListener and ContainerAdapter

The `ContainerListener` interface contains two methods that are called when a `ContainerEvent` occurs; container events are generated when components are added to or removed from a container. The adapter class for `ContainerListener` is `ContainerAdapter`. If you care only about one of the two methods in `ContainerListener`, you can subclass the adapter and override only the method that you are interested in. For a container to listen for container events, it is necessary to call `Container.addContainerListener()` with the class that implements the interface as the parameter.

public abstract void componentAdded(ContainerEvent e) ★

> The `componentAdded()` method is called when a component is added to a container (for example, by a call to `Container.add()`).

public abstract void componentRemoved(ContainerEvent e) ★

> The `componentRemoved()` method is called when a component is removed from a container (for example, by a call to `Container.remove()`).

FocusListener and FocusAdapter

The `FocusListener` interface has two methods, which are called when a `FocusEvent` occurs. Its adapter class is `FocusAdapter`. If you care only about one of the methods, you can subclass the adapter and override the method you are interested in. For an object to listen for a `FocusEvent`, it is necessary to call the `Component.addFocusListener()` method with the class that implements the `FocusListener` interface as the parameter.

public abstract void focusGained(FocusEvent e) ★

> The `focusGained()` method is called when a component receives input focus, usually by the user clicking the mouse in the area of the component.
>
> This method is the Java 1.1 equivalent of `Component.gotFocus()` in the Java 1.0 event model.

public abstract void focusLost(FocusEvent e) ★

> The `focusLost()` method is called when a component loses the input focus.
>
> This method is the Java 1.1 equivalent of `Component.lostFocus()` in the Java 1.0 event model.

ItemListener

The `ItemListener` interface contains the one method that is called when an `ItemEvent` occurs. It has no adapter class. For an object to listen for an `ItemEvent`, it is necessary to call `addItemListener()` with the class that implements the `ItemListener` interface as the parameter. The `addItemListener()` method is implemented by the `Checkbox`, `CheckboxMenuItem`, `Choice`, and `List` components. Other components don't generate item events.

public abstract void itemStateChanged(ItemEvent e) ★

> The `itemStateChanged()` method is called when a component's state is modified. Every component is modified differently; for a `List`, modifying the component means single-clicking on an entry. See the appropriate section for a description of each component.

KeyListener and KeyAdapter

The `KeyListener` interface contains three methods that are called when a `KeyEvent` occurs; key events are generated when the user presses or releases keys. The adapter class for `KeyListener` is `KeyAdapter`. If you only care about one or two of the methods in `KeyListener`, you can subclass the adapter and only override the methods that you are interested in. For an object to listen for key events, it is necessary to call `Component.addKeyListener()` with the class that implements the interface as the parameter.

public abstract void keyPressed(KeyEvent e) ★

> The `keyPressed()` method is called when a user presses a key. A key press is, literally, just what it says. A key press event is called for every key that is pressed, including keys like Shift and Control. Therefore, a `KEY_PRESSED` event has a virtual key code identifying the physical key that was pressed; but that's not the same as a typed character, which usually consists of several key presses (for example, Shift+A to type an uppercase A). The `keyTyped()` method reports actual characters.
>
> This method is the Java 1.1 equivalent of `Component.keyDown()` in the Java 1.0 event model.

public abstract void keyReleased(KeyEvent e) ★

> The `keyReleased()` method is called when a user releases a key. Like the `keyPressed()` method, when dealing with `keyReleased()`, you must think of virtual key codes, not characters.
>
> This method is the Java 1.1 equivalent of `Component.keyUp()` in the Java 1.0 event model.

public abstract void keyTyped(KeyEvent e) ★

> The `keyTyped()` method is called when a user types a key. The method `keyTyped()` method reports the actual character typed. Action-oriented keys, like function keys, do not trigger this method being called.

MouseListener and MouseAdapter

The `MouseListener` interface contains five methods that are called when a nonmotion oriented `MouseEvent` occurs; mouse events are generated when the user presses or releases a mouse button. (Separate classes, `MouseMotionListener` and `MouseMotionAdapter`, are used to handle mouse motion events; this means that you can listen for mouse clicks only, without being bothered by thousands of mouse motion events.) The adapter class for `MouseListener` is `MouseAdapter`. If you care about only one or two of the methods in `MouseListener`, you can subclass the adapter and override only the methods that you are interested in. For an object to listen for mouse events, it is necessary to call the method `Window.addWindowListener()` with the class that implements the interface as the parameter.

public abstract void mouseEntered(MouseEvent e) ★

The `mouseEntered()` method is called when the mouse first enters the bounding area of the component.

This method is the Java 1.1 equivalent of `Component.mouseEnter()` in the Java 1.0 event model.

public abstract void mouseExited(MouseEvent e) ★

The `mouseExited()` method is called when the mouse leaves the bounding area of the component.

This method is the Java 1.1 equivalent of `Component.mouseExit()` in the Java 1.0 event model.

public abstract void mousePressed(MouseEvent e) ★

The `mousePressed()` method is called each time the user presses a mouse button within the component's space.

This method is the Java 1.1 equivalent of `Component.mouseDown()` in the Java 1.0 event model.

public abstract void mouseReleased(MouseEvent e) ★

The `mouseReleased()` method is called when the user releases the mouse button after a mouse press. The user does not have to be over the original component any more; the original component (i.e., the component in which the mouse was pressed) is the source of the event.

This method is the Java 1.1 equivalent of `Component.mouseUp()` in the Java 1.0 event model.

public abstract void mouseClicked(MouseEvent e) ★

The `mouseClicked()` method is called once each time the user clicks a mouse button; that is, once for each mouse press/mouse release combination.

MouseMotionListener and MouseMotionAdapter

The `MouseMotionListener` interface contains two methods that are called when a motion-oriented `MouseEvent` occurs; mouse motion events are generated when the user moves the mouse, whether or not a button is pressed. (Separate classes, `MouseListener` and `MouseAdapter`, are used to handle mouse clicks and entering/exiting components. This makes it easy to ignore mouse motion events, which are very frequent and can hurt performance. You should listen only for mouse motion events if you specifically need them.) `MouseMotionAdapter` is the adapter class for `MouseMotionListener`. If you care about only one of the methods in `MouseMotionListener`, you can subclass the adapter and override only the method that you are interested in. For an object to listen for mouse motion events, it is necessary to call `Component.addMouseMotionListener()` with the class that implements the interface as the parameter.

public abstract void mouseMoved(MouseEvent e) ★

The `mouseMoved()` method is called every time the mouse moves within the bounding area of the component, and no mouse button is pressed.

This method is the Java 1.1 equivalent of `Component.mouseMove()` in the Java 1.0 event model.

public abstract void mouseDragged(MouseEvent e) ★

The `mouseDragged()` method is called every time the mouse moves while a mouse button is pressed. The source of the `MouseEvent` is the component that was under the mouse when it was first pressed.

This method is the Java 1.1 equivalent of `Component.mouseDrag()` in the Java 1.0 event model.

TextListener

The `TextListener` interface contains the one method that is called when a `TextEvent` occurs. It has no adapter class. For an object to listen for a `TextEvent`, it is necessary to call `addTextListener()` with the class that implements the `TextListener` interface as the parameter. The `addTextListener()` method is implemented by the `TextComponent` class, and thus the `TextField` and `TextArea` components. Other components don't generate text events.

public abstract void textValueChanged(TextEvent e) ★

The `textValueChanged()` method is called when a text component's contents are modified, either by the user (by a keystroke) or programmatically (by the `setText()` method).

WindowListener and WindowAdapter

The `WindowListener` interface contains seven methods that are called when a `WindowEvent` occurs; window events are generated when something changes the visibility or status of a window. The adapter class for `WindowListener` is `WindowAdapter`. If you care about only one or two of the methods in `WindowListener`, you can subclass the adapter and override only the methods that you are interested in. For an object to listen for window events, it is necessary to call the method `Window.addWindowListener()` or `Dialog.addWindowListener()` with the class that implements the interface as the parameter.

public abstract void windowOpened(WindowEvent e) ★

The `windowOpened()` method is called when a `Window` is first opened.

public abstract void windowClosing(WindowEvent e) ★

The `windowClosing()` method is triggered whenever the user tries to close the `Window`.

public abstract void windowClosed(WindowEvent e) ★

The `windowClosed()` method is called after the `Window` has been closed.

public abstract void windowIconified(WindowEvent e) ★

The `windowIconified()` method is called whenever a user iconifies a `Window`.

public abstract void windowDeiconified(WindowEvent e) ★

The `windowDeiconified()` method is called when the user deiconifies the `Window`.

public abstract void windowActivated(WindowEvent e) ★

The `windowActivated()` method is called whenever a `Window` is brought to the front.

public abstract void windowDeactivated(WindowEvent e) ★

The `windowDeactivated()` method is called when the `Window` is sent away from the front, either through iconification, closing, or another window becoming active.

AWTEventMulticaster

The `AWTEventMulticaster` class is used by AWT to manage the listener queues for the different events, and for sending events to all interested listeners when they occur (multicasting). Ordinarily, you have no need to work with this class or know about its existence. However, if you wish to create your own components that have their own set of listeners, you can use the class instead of implementing your own event-delivery system. See "Constructor methods" in this section for more on how to use the `AWTEventMulticaster`.

`AWTEventMulticaster` looks like a strange beast, and to some extent, it is. It contains methods to add and remove every possible kind of listener and implements all of the listener interfaces (11 as of Java 1.1). Because it implements all the listener interfaces, you can pass an event multicaster as an argument wherever you expect any kind of listener. However, unlike a class you might implement to listen for a specific kind of event, the multicaster includes machinery for maintaining chains of listeners. This explains the rather odd signatures for the `add()` and `remove()` methods. Let's look at one in particular:

```
public static ActionListener add(ActionListener first, ActionListener second)
```

This method takes two `ActionListener`s and returns another `ActionListener`. The returned listener is actually an event multicaster that contains the two listeners given as arguments in a linked list. However, because it implements the `ActionListener` interface, it is just as much an `ActionListener` as any class you might write; the fact that it contains two (or more) listeners inside it is irrelevant. Furthermore, both arguments can also be event multicasters, containing arbitrarily long chains of action listeners; in this case, the returned listener combines the two chains. Most often, you will use add to add a single listener to a chain that you're building, like this:

```
actionListenerChain=AWTEventMulticaster.add(actionListenerChain,
                                            newActionListener);
```

`actionListenerChain` is an `ActionListener`--but it is also a multicaster holding a chain of action listeners. To start a chain, use `null` for the first argument. You rarely need to call the `AWTEventMulticaster` constructor. `add()` is a static method, so you can use it with either argument set to `null` to start the chain.

Now that you can maintain chains of listeners, how do you use them? Simple; just deliver your event to the appropriate method in the chain. The multicaster takes care of sending the event to all the listeners it contains:

```
actionListenerChain.actionPerformed(new ActionEvent(...));
```

Variables

protected EventListener a; ★
protected EventListener b; ★

> The `a` and `b` event listeners each consist of a chain of `EventListeners`.

Constructor methods

protected AWTEventMulticaster(EventListener a, EventListener b) ★

> The constructor is protected. It creates an `AWTEventMulticaster` instance from the two chains of listeners. An instance is automatically created for you when you add your second listener by calling an `add()` method.

Listener methods

These methods implement all of the listener interfaces. Rather than repeating all the descriptions, the methods are just listed.

public void actionPerformed(ActionEvent e) ★
public void adjustmentValueChanged(AdjustmentEvent e) ★
public void componentAdded(ContainerEvent e) ★
public void componentHidden(ComponentEvent e) ★
public void componentMoved(ComponentEvent e) ★
public void componentRemoved(ContainerEvent e) ★
public void componentResized(ComponentEvent e) ★
public void componentShown(ComponentEvent e) ★
public void focusGained(FocusEvent e) ★
public void focusLost(FocusEvent e) ★
public void itemStateChanged(ItemEvent e) ★
public void keyPressed(KeyEvent e) ★
public void keyReleased(KeyEvent e) ★
public void keyTyped(KeyEvent e) ★
public void mouseClicked(MouseEvent e) ★

public void mouseDragged(MouseEvent e) ★
public void mouseEntered(MouseEvent e) ★
public void mouseExited(MouseEvent e) ★
public void mouseMoved(MouseEvent e) ★
public void mousePressed(MouseEvent e) ★
public void mouseReleased(MouseEvent e) ★
public void textValueChanged(TextEvent e) ★
public void windowActivated(WindowEvent e) ★
public void windowClosed(WindowEvent e) ★
public void windowClosing(WindowEvent e) ★
public void windowDeactivated(WindowEvent e) ★
public void windowDeiconified(WindowEvent e) ★
public void windowIconified(WindowEvent e) ★
public void windowOpened(WindowEvent e) ★

> These methods broadcast the event given as an argument to all the listeners.

Support methods

There is an `add()` method for every listener interface. Again, I've listed them with a single description.

public static ActionListener add(ActionListener first, ActionListener second) ★
public static AdjustmentListener add(AdjustmentListener first, AdjustmentListener second) ★
public static ComponentListener add(ComponentListener first, ComponentListener second) ★
public static ContainerListener add(ContainerListener first, ContainerListener second) ★
public static FocusListener add(FocusListener first, FocusListener second) ★
public static ItemListener add(ItemListener first, ItemListener second) ★
public static KeyListener add(KeyListener first, KeyListener second)
public static MouseListener add(MouseListener first, MouseListener second) ★
public static MouseMotionListener add(MouseMotionListener first, MouseMotionListener second) ★
public static TextListener add(TextListener first, TextListener second) ★
public static WindowListener add(WindowListener first, WindowListener second) ★

> These methods combine the listener sets together; they are called by the "add listener" methods of the various components. Usually, the `first` parameter is the initial listener chain, and the `second` parameter is the listener to add. However, nothing forces that. The combined set of listeners is returned.

protected static EventListener addInternal(EventListener first, EventListener second) ★

> The `addInternal()` method is a support routine for the various `add()` methods. The combined set of listeners is returned.

Again, there are `remove()` methods for every listener type, and I've economized on the descriptions.

public static ComponentListener remove(ComponentListener list, ComponentListener oldListener) ★
public static ContainerListener remove(ContainerListener list, ContainerListener oldListener) ★
public static FocusListener remove(FocusListener list, FocusListener oldListener) ★
public static KeyListener remove(KeyListener list, KeyListener oldListener) ★
public static MouseMotionListener remove(MouseMotionListener list, MouseMotionListener oldListener) ★
public static MouseListener remove(MouseListener list, MouseListener oldListener) ★
public static WindowListener remove(WindowListener list, WindowListener oldListener) ★
public static ActionListener remove(ActionListener list, ActionListener oldListener) ★
public static ItemListener remove(ItemListener list, ItemListener oldListener) ★
public static AdjustmentListener remove(AdjustmentListener list, AdjustmentListener oldListener) ★
public static TextListener remove(TextListener list, TextListener oldListener) ★

> These methods remove `oldListener` from the list of listeners, `list`. They are called by the "remove listener" methods of the different components. If `oldListener` is not found in the `list`, nothing happens. All these methods return the new list of listeners.

protected static EventListener removeInternal(EventListener list, EventListener oldListener) ★

> The `removeInternal()` method is a support routine for the various `remove()` methods. It removes `oldListener` from the list of listeners, `list`. Nothing happens if `oldListener` is not found in the `list`. The new set of listeners is returned.

protected EventListener remove(EventListener oldListener) ★

> This `remove()` method removes `oldListener` from the `AWTEventMulticaster`. It is a support routine for `removeInternal()`.

protected void saveInternal(ObjectOutputStream s, String k) throws IOException ★

> The `saveInternal()` method is a support method for serialization.

Using an event multicaster

Example 4.4 shows how to use `AWTEventMulticaster` to create a component that generates `ItemEvents`. The `AWTEventMulticaster` is used in the `addItemListener()` and `removeItemListener()` methods. When it comes time to generate the event in `processEvent()`, the `itemStateChanged()` method is called to notify anyone who might be interested. The item event is generated when a mouse button is clicked; we just count the number of clicks to determine whether an item was selected or deselected. Since we do not have any mouse listeners, we need to enable mouse events with `enableEvents()` in the constructor, as shown in the following example.

Example 4.4: Using an AWTEventMulticaster

```
// Java 1.1 only
import java.awt.*;
import java.awt.event.*;
class ItemEventComponent extends Component implements ItemSelectable {
    boolean selected;
    int i = 0;
    ItemListener itemListener = null;
    ItemEventComponent () {
        enableEvents (AWTEvent.MOUSE_EVENT_MASK);
    }
    public Object[] getSelectedObjects() {
        Object o[] = new Object[1];
        o[0] = new Integer (i);
        return o;
    }
    public void addItemListener (ItemListener l) {
        itemListener = AWTEventMulticaster.add (itemListener, l);
    }
    public void removeItemListener (ItemListener l) {
        itemListener = AWTEventMulticaster.remove (itemListener, l);
    }
    public void processEvent (AWTEvent e) {
        if (e.getID() == MouseEvent.MOUSE_PRESSED) {
            if (itemListener != null) {
                selected = !selected;
                i++;
                itemListener.itemStateChanged (
                    new ItemEvent (this, ItemEvent.ITEM_STATE_CHANGED,
                        getSelectedObjects(),
                        (selected?ItemEvent.SELECTED:ItemEvent.DESELECTED)));
            }
        }
    }
}
public class ItemFrame extends Frame implements ItemListener {
    ItemFrame () {
        super ("Listening In");
        ItemEventComponent c = new ItemEventComponent ();
        add (c, "Center");
        c.addItemListener (this);
        c.setBackground (SystemColor.control);
        setSize (200, 200);
    }
    public void itemStateChanged (ItemEvent e) {
        Object[] o = e.getItemSelectable().getSelectedObjects();
        Integer i = (Integer)o[0];
        System.out.println (i);
    }
    public static void main (String args[]) {
        ItemFrame f = new ItemFrame();
        f.show();
    }
}
```

The `ItemFrame` displays just an `ItemEventComponent` and listens for its item events.

The `EventQueue` class lets you manage Java 1.1 events directly. You don't usually need to manage events yourself; the system takes care of event delivery behind the scene. However, should you need to, you can acquire the system's event queue by calling `Toolkit.getSystemEventQueue()`, peek into the event queue by calling `peekEvent()`, or post new events by calling `postEvent()`. All of these operations may be restricted by the `SecurityManager`. You should not remove the events from the queue (i.e., don't call `getNextEvent()`) unless you really mean to.Constructors

public EventQueue() ★

> This constructor creates an `EventQueue` for those rare times when you need to manage your own queue of events. More frequently, you just work with the system event queue acquired through the `Toolkit`.

Methods

public synchronized AWTEvent peekEvent() ★

> The `peekEvent()` method looks into the event queue and returns the first event, without removing that event. If you modify the event, your modifications are reflected in the event still on the queue. The returned object is an instance of `AWTEvent`. If the queue is empty, `peekEvent()` returns `null`.

public synchronized AWTEvent peekEvent(int id) ★

> This `peekEvent()` method looks into the event queue for the first event of the specified type. `id` is one of the integer constants from an `AWTEvent` subclass or an integer constant of your own. If there are no events of the appropriate type on the queue, `peekEvent()` returns `null`.
>
> Note that a few of the `AWTEvent` classes have both event types and subtypes; `peekEvent()` checks event types only and ignores the subtype. For example, to find an `ItemEvent`, you would call `peekEvent(ITEM_STATE_CHANGED)`. However, a call to `peekEvent(SELECTED)` would return `null`, since `SELECTED` identifies an `ItemEvent` subtype.

public synchronized void postEvent(AWTEvent theEvent) ★

> This version of `postEvent()` puts a new style (Java1.1) event on the event queue.

public synchronized AWTEvent getNextEvent() throws InterruptedException ★

> The `getNextEvent()` method removes an event from the queue. If the queue is empty, the call waits. The object returned is the item taken from the queue; it is either an `Event` or an `AWTEvent`. If the method call is

interrupted, the method getNextEvent() throws an InterruptedException.

PREVIOUS
The Event Class

HOME
BOOK INDEX

NEXT
Components

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

5.4 A Simple Calculator

It is always helpful to see complete and somewhat useful examples after learning something new. Example 5.2 shows a working calculator that performs floating point addition, subtraction, multiplication, and division. Figure 5.4 shows the calculator in operation. The button in the lower left corner is a decimal point. This applet uses a number of classes that will be discussed later in the book (most notably, some layout managers and a `Panel`); try to ignore them for now. Focus on the `action()` and `compute()` methods; `action()` figures out which button was pressed, converting it to a digit (0-9 plus the decimal point) or an operator (=, +, -, *, /). As you build a number, it is displayed in the label `lab`, which conveniently serves to store the number in string form. The `compute()` method reads the label's text, converts it to a floating point number, does the computation, and displays the result in the label. The `addButtons()` method is a helper method to create a group of `Button` objects at one time.

Example 5.2: Calculator Source Code

```
import java.awt.*;
import java.applet.*;
public class JavaCalc extends Applet {
    Label lab;
    boolean firstDigit = true;
    float savedValue = 0.0f;     // Initial value
    String operator = "=";  // Initial operator
    public void addButtons (Panel p, String labels) {
        int count = labels.length();
        for (int i=0;i<count;i++)
            p.add (new Button (labels.substring(i,i+1)));
    }
    public void init () {
        setLayout (new BorderLayout());
        add ("North", lab = new Label ("0", Label.RIGHT));
        Panel p = new Panel();
        p.setLayout (new GridLayout (4, 4));
        addButtons (p, "789/");
        addButtons (p, "456*");
        addButtons (p, "123-");
        addButtons (p, ".0=+");
        add ("Center", p);
    }
    public boolean action (Event e, Object o) {
        if (e.target instanceof Button) {
            String s = (String)o;
            if ("0123456789.".indexOf (s) != -1) { // isDigit
                if (firstDigit) {
                    firstDigit = false;
                    lab.setText (s);
                } else {
                    lab.setText (lab.getText() + s);
                }
            } else { // isOperator
                if (!firstDigit) {
                    compute (lab.getText());
                    firstDigit = true;
                }
                operator = s;
            }
            return true;
        }
        return false;
    }
    public void compute (String s) {
        float sValue = new Float (s).floatValue();
        char c = operator.charAt (0);
        switch (c) {
            case '=':   savedValue  = sValue;
                        break;
            case '+':   savedValue += sValue;
                        break;
            case '-':   savedValue -= sValue;
                        break;
            case '*':   savedValue *= sValue;
                        break;
            case '/':   savedValue /= sValue;
                        break;
        }
        lab.setText (String.valueOf(savedValue));
    }
}
```

Figure 5.4: Calculator applet

[Graphic: Figure 5-4]

PREVIOUS Buttons | HOME | BOOK INDEX | NEXT Canvas

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

5.6 Creating Your Own Component

If you find that no AWT component satisfies your needs, you can create your own. This is usually done either by extending an existing component or by starting from scratch. When extending an existing component, you start with the base functionality of an existing object and add to it. The users will not see anything new or different about the object until they start to interact with it, since it is not a new component. For example, a `TextField` could be subclassed to convert all letters input to uppercase. On the other hand, if you create a new component from scratch, it will appear the same on all platforms (regardless of what the platform's native components look like), and you have to make sure the user can fairly easily figure out how to work with it. Example 5.3 shows how to create your own `Component` by creating a `Label` that displays vertically, as opposed to the standard `Label` `Component` that displays horizontally. The whole process is fairly easy.

The third possibility for creating your own components involves adding functionality to containers. This is fairly easy to do and can be useful if you are constantly grouping components together. For example, if you are always adding a `TextField` or `Label` to go with a `Scrollbar` to display the value, do it once, and call it something meaningful like `LabeledScrollbarPanel`. Then whenever you need it again, reuse your `LabeledScrollbarPanel`. Think about reusability whenever you can.

With Java 1.1, the colors for these new components should be set to color values consistent to the user's platform. This is done through color constants provided in the `SystemColor` class introduced in Chapter 2, *Simple Graphics*.

VerticalLabel

When you create new components, they must meet three requirements:

- In Java 1.0, you must extend a subclass of `Component`, usually `Canvas`. In Java 1.1, you can extend `Component` itself, creating a lightweight component. In many cases, this alternative is more efficient.
- You must provide a constructor for the new component so that you can create new instances of it; if you really don't need a constructor, you can use the default constructor that you inherit from `Canvas` or `Component`.
- You must provide a way to draw the object on the screen by overriding the `paint()` method.

If initializing the component requires information about display characteristics (for example, you need to know the default `Font`), you must wait until the object is displayed on the screen before you initialize it. This is done by overriding the `addNotify()` method. First, call `super.addNotify()` to create the peer; you can now ask for platform-dependent information and initialize your component accordingly. Remember to override `getPreferredSize()` and `getMinimumSize()` (the Java 1.0 names are `preferredSize()` and `minimumSize()`) to return the proper dimensions for the new component, so that layout management works properly. There can be other support methods, depending upon the requirements of the object. For example, it is helpful, but not required, to provide a `toString()` or `paramString()` method.

Creating a new component sounds a lot harder than it is. Example 5.3 contains the source for a new component called `VerticalLabel`. It displays a label that reads from top to bottom, instead of from left to right, and can be configured to display its text right or left justified or centered. Figure 5.5 displays the new component `VerticalLabel` in action.

Example 5.3: Source for VerticalLabel Component

```
import java.awt.*;
public class VerticalLabel extends Canvas {
   public static final int LEFT = 0;
   public static final int CENTER = 1;
   public static final int RIGHT = 2;
   private String text;
   private int    vgap;
   private int    alignment;
   Dimension      mySize;
   int            textLength;
   char           chars[];
   // constructors
   public VerticalLabel () {
        this (null, 0, CENTER);
   }
   public VerticalLabel (String text) {
        this (text, 0, CENTER);
   }
   public VerticalLabel (String text, int vgap, int alignment) {
      this.text = text;
      this.vgap = vgap;
      this.alignment = alignment;
   }
   void init () {
      textLength = text.length();
      chars = new char[textLength];
      text.getChars (0, textLength, chars, 0);
      Font f = getFont();
      FontMetrics fm = getFontMetrics (f);
      mySize = new Dimension(0,0);
      mySize.height = (fm.getHeight() * textLength) + (vgap * 2);
      for (int i=0; i < textLength; i++) {
          mySize.width = Math.max (mySize.width, fm.charsWidth(chars, i, 1));
      }
   }
   public int getAlignment () {
      return alignment;
   }
   public void addNotify () {
       super.addNotify();
       init();  // Component must be visible for init to work
   }
   public void setText (String text)    {this.text = text; init();}
   public String getText ()             {return text; }
   public void setVgap (int vgap)       {this.vgap = vgap; init();}
   public int getVgap ()                {return vgap; }
   public Dimension preferredSize ()    {return mySize; }
   public Dimension minimumSize ()      {return mySize; }
   public void paint (Graphics g) {
      int x,y;
      int xPositions[];
      int yPositions[];
// Must redo this each time since font/screen area might change
// Use actual width for alignment
      Font f = getFont();
      FontMetrics fm = getFontMetrics (f);
      xPositions = new int[textLength];
      for (int i=0; i < textLength; i++) {
         if (alignment == RIGHT) {
            xPositions[i] = size().width - fm.charWidth (chars[i]);
         } else if (alignment == LEFT) {
            xPositions[i] = 0;
         } else {// CENTER
            xPositions[i] = (size().width - fm.charWidth (chars[i])) / 2;
         }
      }
      yPositions = new int[textLength];
      for (int i=0; i < textLength; i++) {
         yPositions[i] = (fm.getHeight() * (i+1)) + vgap;
      }
      for (int i = 0; i < textLength; i++) {
         x = xPositions[i];
         y = yPositions[i];
         g.drawChars (chars, i, 1, x, y);
      }
   }
   protected String paramString () {
      String str=",align=";
      switch (alignment) {
         case LEFT:    str += "left"; break;
         case CENTER:  str += "center"; break;
         case RIGHT:   str += "right"; break;
      }
      if (vgap!=0) str+= ",vgap=" + vgap;
      return super.paramString() + str + ",label=" + text;
   }
}
```

Figure 5.5: Using VerticalLabel

[Graphic: Figure 5-5]

The following code is a simple applet using the `VerticalLabel`. It creates five instances of `VerticalLabel` within a `BorderLayout` panel, with gaps (see Chapter 7, *Layouts* for more on `BorderLayout`). The top and bottom labels are justified to the left and right, respectively, to demonstrate justification.

```
import java.awt.*;
import java.applet.*;
public class vlabels extends Applet {
   public void init () {
        setLayout (new BorderLayout (10, 10));
        setFont (new Font ("TimesRoman", Font.BOLD, 12));
        add ("North",  new VerticalLabel ("One", 10, VerticalLabel.LEFT));
        add ("South",  new VerticalLabel ("Two", 10, VerticalLabel.RIGHT));
        add ("West",   new VerticalLabel ("Three"));
        add ("East",   new VerticalLabel ("Four"));
        add ("Center", new VerticalLabel ("Five"));
        resize (preferredSize());
   }
}
```

Lightweight VerticalLabel

The `VerticalLabel` in Example 5.3 works in both Java 1.0 and 1.1 but is relatively inefficient. When you create one, the system must create a `Canvas` and the peer of the `Canvas`. This work doesn't gain you anything; since this is a new component, it doesn't have to match the native appearance of any other component.

In Java 1.1, there's a way to avoid the overhead if you are creating a component that doesn't have to match a native object. This is called a *lightweight component*. To create one, you just subclass `Component` itself. To make a lightweight version of our `VerticalLabel`, we have to change only one line of code.

```
// Java 1.1 only
public class VerticalLabel extends Component
```

Everything else remains unchanged.

5.7 Cursor

Introduced in Java 1.1, the `Cursor` class provides the different cursors that can be associated with a `Component`. Previously, cursors could only be associated with a whole `Frame`. Now any component can use fancy cursors when the user is interacting with the system.

To change the cursor, a component calls its `setCursor()` method; its argument is a `Cursor` object, which is defined by this class.

NOTE:

There is still no way to assign a user-defined cursor to a `Component`. You are restricted to the 14 predefined cursors.

Cursor Constants

The following is a list of `Cursor` constants. The cursors corresponding to the constants are shown in Figure 5.6.

public final static int DEFAULT_CURSOR
public final static int CROSSHAIR_CURSOR
public final static int TEXT_CURSOR
public final static int WAIT_CURSOR
public final static int HAND_CURSOR
public final static int MOVE_CURSOR
public final static int N_RESIZE_CURSOR
public final static int S_RESIZE_CURSOR
public final static int E_RESIZE_CURSOR
public final static int W_RESIZE_CURSOR
public final static int NE_RESIZE_CURSOR
public final static int NW_RESIZE_CURSOR

public final static int SE_RESIZE_CURSOR
public final static int SW_RESIZE_CURSOR

Figure 5.6: Standard Java cursors

[Graphic: Figure 5-6]

Cursor Methods

public Cursor (int type) ★

The sole constructor creates a `Cursor` of the specified `type`. `type` must be one of the `Cursor` class constants. If `type` is not one of the class constants, the constructor throws the run-time exception `IllegalArgumentException`.

This constructor exists primarily to support object serialization; you don't need to call it in your code. It is more efficient to call `getPredefinedCursor()`, discussed later in this section.

Miscellaneous methods

public int getType() ★

The `getType()` method returns the cursor type. The value returned is one of the class constants.

static public Cursor getPredefinedCursor(int type) ★

The `getPredefinedCursor()` method returns the predefined `Cursor` of the given `type`. If `type` is not one of the class constants, this method throws the run-time exception `IllegalArgumentException`. This method checks what `Cursor` objects already exist and gives you a reference to a preexisting `Cursor` if it can find one with the appropriate type. Otherwise, it creates a new `Cursor` for you. This is more efficient than calling the `Cursor` constructor whenever you need one.

static public Cursor getDefaultCursor() ★

The `getDefaultCursor()` method returns the predefined `Cursor` for the `DEFAULT_CURSOR` type.

PREVIOUS | HOME | NEXT

Creating Your Own Component | BOOK INDEX | Containers

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

6.2 Panel

The `Panel` class provides a generic container within an existing display area. It is the simplest of all the containers. When you load an applet into Netscape Navigator or an *appletviewer*, you have a `Panel` to work with at the highest level.

A `Panel` has no physical appearance. It is just a rectangular display area. The default `LayoutManager` of `Panel` is `FlowLayout`; `FlowLayout` is described in FlowLayout.

Panel Methods

Constructors

public Panel ()

> The first constructor creates a `Panel` with a `LayoutManager` of `FlowLayout`.

public Panel (LayoutManager layout) ★

> This constructor allows you to set the initial `LayoutManager` of the new `Panel` to `layout`. If `layout` is `null`, there is no `LayoutManager`, and you must shape and position the components within the `Panel` yourself.

Miscellaneous methods

public void addNotify ()

> The `addNotify()` method creates the `Panel` peer. If you override this method, first call `super.addNotify()`, then add your customizations for the new class. Then you can do everything you need with the information about the newly created peer.

Panel Events

In Java 1.0, a `Panel` peer generates all the events that are generated by the `Component` class; it does not generate events that are specific to a particular type of component. That is, it generates key events, mouse events, and focus events; it doesn't generate action events or list events. If an event happens within a child component of a `Panel`, the target of the event is the child component, not the `Panel`. There's one exception to this rule: if a component uses the `LightweightPeer` (new to Java 1.1), it cannot be the target of an event.

With Java 1.1, events are delivered to whatever listener is associated with a contained component. The fact that the component is within a `Panel` has no relevance.

PREVIOUS Container | HOME BOOK INDEX | NEXT Insets

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

6.3 Insets

The `Insets` class provides a way to encapsulate the layout margins of the four different sides of a container. The class helps in laying out containers. The `Container` can retrieve their values through the `getInsets()` method, then analyze the settings to position components. The different inset values are measured in pixels. The space reserved by insets can still be used for drawing directly within `paint()`. Also, if the `LayoutManager` associated with the container does not look at the insets, the request will be completely ignored.

Insets Methods

Variables

There are four variables for insets, one for each border.

public int top

> This variable contains the border width in pixels for the top of a container.

public int bottom

> This variable contains the border width in pixels for the bottom of a container.

public int left

> This variable contains the border width in pixels for the left edge of a container.

public int right

> This variable contains the border width in pixels for the right edge of a container.

Constructors

public Insets (int top, int left, int bottom, int right)

> The constructor creates an `Insets` object with `top`, `left`, `bottom`, and `right` being the size of the insets in pixels. If this object was the return object from the `getInsets()` method of a container, these values represent the size of a border inside that container.

Miscellaneous methods

public Object clone ()

> The `clone()` method creates a clone of the `Insets` so the same `Insets` object can be associated with multiple containers.

public boolean equals(Object object) ★

> The `equals()` method defines equality for insets. Two `Insets` objects are equal if the four settings for the different values are equal.

public String toString ()

> The `toString()` method of `Insets` returns the current settings. Using the `new Insets (10, 20, 30, 40)` constructor, the results would be:

```
java.awt.Insets[top=10,left=20,bottom=30,right=40]
```

Insets Example

The following source code demonstrates the use of insets within an applet's `Panel`. The applet displays a button that takes up the entire area of the `Panel`, less the insets, then draws a rectangle around that area. This is shown visually in Figure 6.1. The example demonstrates that if you add components to a container, the `LayoutManager` deals with the insets for you in positioning them. But if you are drawing directly to the `Panel`, you must look at the insets if you want to avoid the requested area within the container.

```
import java.awt.*;
import java.applet.*;
public class myInsets extends Applet {
    public Insets insets () {
        return new Insets (50, 50, 50, 50);
    }
    public void init () {
        setLayout (new BorderLayout ());
        add ("Center", new Button ("Insets"));
    }
    public void paint (Graphics g) {
        Insets i = insets();
        int width  = size().width - i.left - i.right;
        int height = size().height - i.top - i.bottom;
        g.drawRect (i.left-2, i.top-2, width+4, height+4);
        g.drawString ("Insets Example", 25, size().height - 25);
    }
}
```

Figure 6.1: Insets

[Graphic: Figure 6-1]

To change the applet's insets from the default, we override the `insets()` method to return a new `Insets` object, with the new values.

PREVIOUS
Panel

HOME
BOOK INDEX

NEXT
Window

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

6.4 Window

A `Window` is a top-level display area that exists outside the browser or applet area you are working in. It has no adornments, such as the borders, window title, or menu bar that a typical window manager might provide. A `Frame` is a subclass of `Window` that adds these parts (borders, window title). Normally you will work with the children of `Window` and not `Window` directly. However, you might use a `Window` to create your own pop-up menu or some other GUI component that requires its own window and isn't provided by AWT. This technique isn't as necessary in Java 1.1, which has a `PopupMenu` component.

The default `LayoutManager` for `Window` is `BorderLayout`, which is described in BorderLayout.

Window Methods

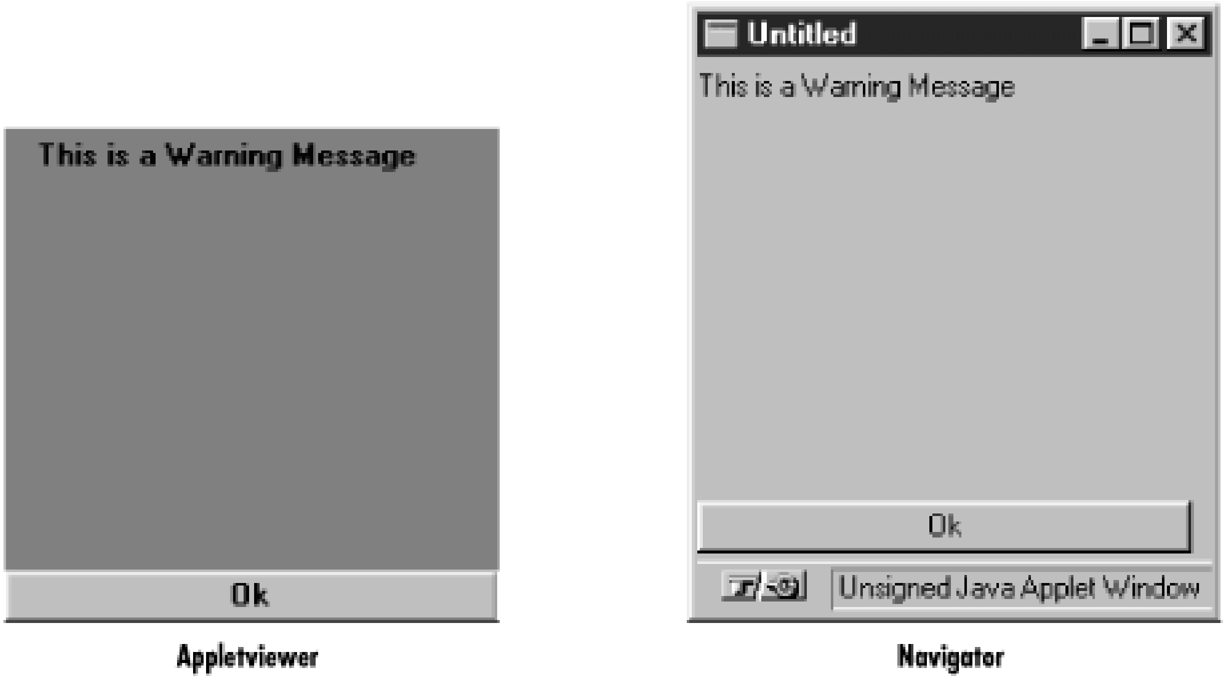
Constructors

public Window (Frame parent)

> There is one public constructor for `Window`. It has one parameter, which specifies the `parent` of the `Window`. When the `parent` is minimized, so is the `Window`. In an application, you must therefore create a `Frame` before you can create a `Window`; this isn't much of an inconvenience since you usually need a `Frame` in which to build your user interface. In an applet, you often do not have access to a `Frame` to use as the parent, so you can pass `null` as the argument.
>
> Figure 6.2 shows a simple `Window` on the left. Notice that there are no borders or window management adornments present. The `Window` on the right was created by an applet loaded over the network. Notice the warning message you get in the status bar at the bottom of the screen. This is to warn users that the `Window` was created by an applet that comes from an untrusted source, and you can't necessarily trust it to do what it says. The warning is particularly appropriate for windows, since a user can't necessarily tell whether a window was created by an applet or any other application. It is therefore possible to write applets that mimic windows from well-known applications, to trick the user into giving away passwords, credit card numbers, or other sensitive

information.

Figure 6.2: Two windows



This is a Warning Message

Appletviewer

Navigator

In some environments, you can get the browser's `Frame` to use with the `Window`'s constructor. This is one way to create a `Dialog`, as we shall see. By repeatedly calling `getParent()` until there are no more parents, you can discover an applet's top-level parent, which should be the browser's `Frame`. Example 6.1 contains the code you would write to do this. You should then check the return value to see if you got a `Frame` or `null`. This code is completely nonportable, but you may happen to be in an environment where it works.

Example 6.1: Finding a Parent Frame

```
import java.awt.*;
public class ComponentUtilities {
    public static Frame getTopLevelParent (Component component) {
        Component c = component;
        while (c.getParent() != null)
            c = c.getParent();
        if (c instanceof Frame)
            return (Frame)c;
        else
            return null;
    }
}
```

Appearance methods

A handful of methods assist with the appearance of the `Window`.

public void pack ()

The `pack()` method resizes the `Window` to the preferred size of the components it contains and validates the `Window`.

public void show ()

The `show()` method displays the `Window`. When a `Window` is initially created it is hidden. If the window is already showing when this method is called, it calls `toFront()` to bring the window to the foreground. To hide the window, just call the `hide()` method of `Component`. After you `show()` a window, it is validated for you.

The first call to `show()` for any `Window` generates a `WindowEvent` with the ID `WINDOW_OPENED`.

public void dispose ()

The `dispose()` method releases the resources of the `Window` by hiding it and removing its peer. Calling this method generates a `WindowEvent` with the ID `WINDOW_CLOSED`.

public void toFront ()

The `toFront()` method brings the `Window` to the foreground of the display. This is automatically called if you call `show()` and the `Window` is already shown.

public void toBack ()

The `toBack()` method puts the `Window` in the background of the display.

public boolean isShowing() ★

The `isShowing()` method returns `true` if the `Window` is visible on the screen.

Miscellaneous methods

public Toolkit getToolkit ()

The `getToolkit()` method returns the current `Toolkit` of the window. The `Toolkit` provides you with information about the native platform. This will allow you to size the `Window` based upon the current screen resolution and get images for an application. See Building a New Component from a Window for a usage example.

public Locale getLocale () ★

The `getLocale()` method retrieves the current `Locale` of the window, if it has one. Using a `Locale` allows you to write programs that can adapt themselves to different languages and different regional variants. If no `Locale` has been set, `getLocale()` returns the default `Locale`. The default `Locale` has a user language of English and no region. To change the default `Locale`, set the system properties `user.language` and `user.region` or call `Locale.setDefault()` (`setDefault()` verifies access rights with the security manager).[1]

[1] For more on the `Locale` class, see the *Java Fundamental Classes Reference* from O'Reilly & Associates.

public final String getWarningString ()

The `getWarningString()` method returns `null` or a string that is displayed on the bottom of insecure `Window` instances. If the `SecurityManager` says that top-level windows do not get a warning message, this method returns `null`. If a message is required, the default text is "Warning: Applet Window". However, Java allows the user to change the warning by setting the system property `awt.appletWarning`. (Netscape Navigator and Internet Explorer do not allow the warning message to be changed. Netscape Navigator's current (V3.0) warning string is "Unsigned Java Applet Window.") The purpose of this string is to warn users that the `Window` was created by an untrusted source, as opposed to a standard application, and should be used with caution.

public Component getFocusOwner () ★

The `getFocusOwner()` method allows you to ask the `Window` which of its components currently has the input focus. This is useful if you are cutting and pasting from the system clipboard; asking who has the input focus tells you where to put the data you get from the clipboard. The system clipboard is covered in Chapter 16, *Data Transfer*. If no component in the `Window` has the focus, `getFocusOwner()` returns `null`.

public synchronized void addNotify ()

The `addNotify()` method creates the `Window` peer. This is automatically done when you call the `show()` method of the `Window`. If you override this method, first call `super.addNotify()`, then add your customizations for the new class. Then you can do everything you need to with the information about the newly created peer.

Window Events

In Java 1.0, a `Window` peer generates all the events that are generated by the `Component` class; it does not generate events that are specific to a particular type of component. That is, it generates key events, mouse events, and focus events; it doesn't generate action events or list events. If an event occurs within a child component of a `Window`, the target of the event is the child component, not the `Window`.

In addition to the `Component` events, five events are specific to windows, none of which are passed on by the window's peer. These events happen at the `Frame` and `Dialog` level. The events are `WINDOW_DESTROY`, `WINDOW_EXPOSE`, `WINDOW_ICONIFY`, `WINDOW_DEICONIFY`, and `WINDOW_MOVED`. The default event handler, `handleEvent()`, doesn't call a convenience method to handle any of these events. If you want to work with them, you must override `handleEvent()`. See Frame Events for an example that catches the `WINDOW_DESTROY` event.

public boolean postEvent (Event e) ☆

The `postEvent()` method tells the `Window` to deal with `Event e`. It calls the `handleEvent()` method, which returns `true` if somebody handled `e` and `false` if no one handles it. This method, which overrides `Component.postEvent()`, is necessary because a `Window` is, by definition, an outermost container, and therefore does not need to post the event to its parent.

Listeners and 1.1 event handling

With the 1.1 event model, you register listeners for different event types; the listeners are told when the event happens. These methods register listeners and let the `Window` component inspect its own events.

public void addWindowListener(WindowListener listener) ★

The `addWindowListener()` method registers `listener` as an object interested in being notified when an `WindowEvent` passes through the `EventQueue` with this `Window` as its target. When such an event occurs, one of the methods in the `WindowListener` interface is called. Multiple listeners can be registered.

public void removeWindowListener(WindowListener listener) ★

The `removeWindowListener()` method removes `listener` as an interested listener. If `listener` is not registered, nothing happens.

protected void processEvent(AWTEvent e) ★

The `processEvent()` method receives every `AWTEvent` with this `Window` as its target. `processEvent()` then passes them along to any listeners for processing. When you subclass `Window`, overriding `processEvent()` allows you to process all events yourself, before sending them to any listeners. In a way, overriding `processEvent()` is like overriding `handleEvent()` using the 1.0 event model.

If you override `processEvent()`, remember to call `super.processEvent(e)` last to ensure that regular event processing can occur. If you want to process your own events, it's a good idea to call `enableEvents()` (inherited from `Component`) to ensure that events are delivered even in the absence of registered listeners.

protected void processWindowEvent(WindowEvent e) ★

The `processWindowEvent()` method receives every `WindowEvent` with this `Window` as its target. `processWindowEvent()` then passes them along to any listeners for processing. When you subclass `Window`, overriding `processWindowEvent()` allows you to process all events yourself, before sending them to any listeners. In a way, overriding `processWindowEvent()` is like overriding `handleEvent()` using the 1.0 event model.

If you override `processWindowEvent()`, you must remember to call `super.processWindowEvent(e)` last to ensure that regular event processing can occur. If you want to process your own events, it's a good idea to call `enableEvents()` (inherited from `Component`) to ensure that events are delivered even in the absence of registered listeners.

6.5 Frames

The `Frame` is a special type of `Window` that looks like other high level programs in your windowing environment. It adds a `MenuBar`, window title, and window gadgets (like resize, maximize, minimize, window menu) to the basic `Window` object. All the menu-related pieces are discussed in Chapter 10, *Would You Like to Choose from the Menu?*

The default layout manager for a `Frame` is `BorderLayout`.

Frame Constants

The `Frame` class includes a number of constants used to specify cursors. These constants are left over from Java 1.0 and maintained for compatibility. In Java 1.1, you should use the new `Cursor` class, introduced in the previous chapter, and the `Component.setCursor()` method to change the cursor over a frame. Avoid using the `Frame` constants for new code. To see these cursors, refer to Figure 5.6.

public final static int DEFAULT_CURSOR
public final static int CROSSHAIR_CURSOR
public final static int TEXT_CURSOR
public final static int WAIT_CURSOR
public final static int SW_RESIZE_CURSOR
public final static int SE_RESIZE_CURSOR
public final static int NW_RESIZE_CURSOR
public final static int NE_RESIZE_CURSOR
public final static int N_RESIZE_CURSOR
public final static int S_RESIZE_CURSOR
public final static int W_RESIZE_CURSOR
public final static int E_RESIZE_CURSOR
public final static int HAND_CURSOR
public final static int MOVE_CURSOR

NOTE:

HAND_CURSOR and MOVE_CURSOR are not available on Windows platforms with Java 1.0. If you ask to use these and they are not available, you get DEFAULT_CURSOR.

Frame Constructors

public Frame ()

The constructor for Frame creates a hidden window with a window title of "Untitled" (Java1.0) or an empty string (Java1.1). Like Window, the default LayoutManager of a Frame is BorderLayout. DEFAULT_CURSOR is the initial cursor. To position the Frame on the screen, call Component.move(). Since the Frame is initially hidden, you need to call the show() method before the user sees the Frame.

public Frame (String title)

This version of Frame's constructor is identical to the first but sets the window title to title. Figure 6.3 shows the results of a call to new Frame("My Frame") followed by resize() and show().

Figure 6.3: A typical Frame

[Graphic: Figure 6-3]

Frame Methods

public String getTitle ()

The getTitle() method returns the current title for the Frame. If there is no title, this method returns null.

public void setTitle (String title)

The setTitle() method changes the Frame's title to title.

public Image getIconImage ()

The `getIconImage()` method returns the image used as the icon. Initially, this returns `null`. For some platforms, the method should not be used because the platform does not support the concept.

public void setIconImage (Image image)

The `setIconImage()` method changes the image to display when the `Frame` is iconified to `image`. Not all platforms utilize this resource.

public MenuBar getMenuBar ()

The `getMenuBar()` method retrieves the `Frame`'s current menu bar.

public synchronized void setMenuBar (MenuBar bar)

The `setMenuBar()` method changes the menu bar of the `Frame` to `bar`. If `bar` is `null`, it removes the menu bar so that none is available. It is possible to have multiple menu bars based upon the context of the application. However, the same menu bar cannot appear on multiple frames and only one can appear at a time. The `MenuBar` class, and everything to do with menus, is covered in Chapter 10, *Would You Like to Choose from the Menu?*.

public synchronized void remove (MenuComponent component)

The `remove()` method removes `component` from `Frame` if `component` is the frame's menu bar. This is equivalent to calling `setMenuBar()` with a parameter of `null` and in actuality is what `remove()` calls.

public synchronized void dispose ()

The `dispose()` method frees up the system resources used by the `Frame`. If any `Dialogs` or `Windows` are associated with this `Frame`, their resources are freed, too. Some people like to call `Component.hide()` before calling the `dispose()` method so users do not see the frame decomposing.

public boolean isResizable ()

The `isResizable()` method will tell you if the current `Frame` is resizable.

public void setResizable (boolean resizable)

The `setResizable()` method changes the resize state of the `Frame`. A `resizable` value of `true` means the user can resize the `Frame`, `false` means the user cannot. This must be set before the `Frame` is shown or the peer created.

public void setCursor (int cursorType)

The `setCursor()` method changes the cursor of the `Frame` to `cursorType`. `cursorType` must be one of the cursor constants provided with the `Frame` class. You cannot create your own cursor image yet. When changing from the `DEFAULT_CURSOR` to another cursor, the mouse must be moved for the cursor icon to change to the new cursor. If `cursorType` is not one of the predefined cursor types, `setCursor()` throws the `IllegalArgumentException` run-time exception.

This method has been replaced by the `Component.setCursor()` method. Both function equivalently, but this method is being phased out.

public int getCursorType ()

The `getCursorType()` method retrieves the current cursor.

This method has been replaced by the `Component.getCursor()` method. Both function equivalently, but this method is being phased out.

Miscellaneous methods

public synchronized void addNotify ()

The `addNotify()` method creates the `Frame` peer. This is automatically done when you call the `show()` method of the `Frame`. If you override this method, first call `super.addNotify()`, then add your customizations for the new class. Then you can do everything you need to do with the information about the newly created peer.

protected String paramString ()

When you call the `toString()` method of `Frame`, the default `toString()` method of `Component` is called. This in turn calls `paramString()`, which builds up the string to display. At the `Frame` level, `paramString()` appends `resizable` (if `true`) and the `title` (if present). Using the default `Frame` constructor, the results would be:

```
java.awt.Frame[0,0,0x0,invalid,hidden,layout=java.awt.BorderLayout,
resizable,title=]
```

Until the `Frame` is shown, via `show()`, the position and size are not known and therefore appear as zeros. After showing the `Frame`, you might see:

```
java.awt.Frame[44,44,300x300,layout=java.awt.BorderLayout,
resizable,title=]
```

Frame Events

In Java 1.0, a `Frame` peer generates all the events that are generated by the `Component` class; it does not generate events that are specific to a particular type of component. That is, it generates key events, mouse events, and focus events; it doesn't generate action events or list events. If an event happens within a child component of a `Frame`, the target of the event is the child component, not the `Frame`.Window

In addition to the `Component` events, `Frame` generates the `WINDOW` events. These events are `WINDOW_DESTROY`, `WINDOW_EXPOSE`, `WINDOW_ICONIFY`, `WINDOW_DEICONIFY`, and `WINDOW_MOVED`.

One common event, `WINDOW_DESTROY`, is generated when the user tries to close the `Frame` by selecting Quit, Close, or Exit (depending on your windowing environment) from the window manager's menu. By default, this event does nothing. You must provide an event handler that explicitly closes the `Frame`. If you do not, your `Frame` will close only when the Java Virtual Machine exits--for example, when you quit Netscape Navigator. The `handleEvent()` method in the following example, or one like it, should therefore be included in all classes that extend `Frame`. If a `WINDOW_DESTROY` event occurs, it gets rid of the `Frame` and exits the program. Make sure your method calls `super.handleEvent()` to process the other events.

```
public boolean handleEvent (Event e) {
    if (e.id == Event.WINDOW_DESTROY) {
        hide();
        dispose();
        System.exit(0);
        return true;          // boolean method, must return something
    } else {
                              // handle other events we find interesting
    }
                              // make sure normal event processing happens
    return super.handleEvent (e);
}
```

Listeners and 1.1 event handling

With the 1.1 event model, you register listeners for different event types; the listeners are told when the event happens. The `Frame` class inherits all its listener handling from `Window`.

Here's the Java 1.1 code necessary to handle `WINDOW_CLOSING` events; it is equivalent to the `handleEvent()` method in the previous example. First, you must add the following line to the `Frame`'s constructor:

```
enableEvents (AWTEvent.WINDOW_EVENT_MASK);
```

This line guarantees that we will receive window events, even if there is no listener. The `processWindowEvent()` method in the following code does the actual work of closing things down:

```
// Java 1.1 only
protected void processWindowEvent(WindowEvent e) {
    if (e.getID() == WindowEvent.WINDOW_CLOSING) {
        // Notify others we are closing
        if (windowListener != null)
            windowListener.windowClosing(e);
        System.exit(0);
    } else {
        super.processEvent(e);
    }
}
```

If you forget to enable events, `processWindowEvent()` may never be called, and your windows will not shut down until the Java Virtual Machine exits. All subclasses of `Frame` should include code like this to make sure they terminate gracefully.

Building a New Component from a Window

Now that we have discussed the `Frame` and `Window` objects, we can briefly investigate some ways to use them together. Previously I said that you can use a `Window` to build your own pop-up menu. That's no longer necessary in Java 1.1, but the same techniques apply to plenty of other objects. In the following example, we build a set of pop-up buttons; it also uses the `Toolkit` of a `Frame` to load images within an application. The pop-up button set appears when the user presses the right mouse button over the image. It is positioned at the coordinates of the `mouseDown()` event; to do so, we add the current `location()` of the `Frame` to the mouse's `x` and `y` coordinates. Figure 6.4 shows what this application looks like when the pop-up button set is on the screen.

```
import java.awt.*;
public class PopupButtonFrame extends Frame {
    Image im;
    Window w = new PopupWindow (this);
    PopupButtonFrame () {
        super ("PopupButton Example");
        resize (250, 100);
        show();
        im = getToolkit().getImage ("rosey.jpg");
        MediaTracker mt = new MediaTracker (this);
        mt.addImage (im, 0);
        try {
            mt.waitForAll();
        } catch (Exception e) {e.printStackTrace(); }
    }
    public static void main (String args[]) {
        Frame f = new PopupMenuFrame ();
    }
    public void paint (Graphics g) {
        if (im != null)
            g.drawImage (im, 20, 20, this);
    }
    public boolean mouseDown (Event e, int x, int y) {
        if (e.modifiers == Event.META_MASK) {
            w.move (location().x+x, location().y+y);
            w.show();
            return true;
        }
        return false;
    }
}
class PopupWindow extends Window {
    PopupWindow (Frame f) {
        super (f);
        Panel p = new Panel ();
        p.add (new Button ("About"));
        p.add (new Button ("Save"));
        p.add (new Button ("Quit"));
        add ("North", p);
        setBackground (Color.gray);
        pack();
    }
    public boolean action (Event e, Object o) {
        if ("About".equals (o))
            System.out.println ("About");
        else if ("Save".equals (o))
            System.out.println ("Save Me");
        else if ("Quit".equals (o))
            System.exit (0);
        hide();
        return true;
    }
}
```

Figure 6.4: Pop-up buttons

[Graphic: Figure 6-4]

The most interesting method in this application is `mouseDown()`. When the user clicks on the mouse, `mouseDown()` checks whether the `META_MASK` is set in the event modifiers; this indicates that the user pressed the right mouse button, or pressed the left button while pressing the Meta key. If this is `true`, `mouseDown()` moves the window to the location of the mouse click, calls `show()` to display the window, and returns `true` to indicate that the event was handled completely. If `mouseDown` were called with any other kind of mouse event, we return `false` to let the event propagate to any other object that might be interested. Remember that the coordinates passed with the mouse event are the coordinates of the mouse click relative to the `Frame`; to find out where to position the pop-up window, we need an absolute location and therefore ask the `Frame` for its location.

`PopupWindow` itself is a simple class. Its constructor simply creates a display with three buttons. The call to `pack()` sizes the window so that it provides a nice border around the buttons but isn't excessively large; you can change the border by playing with the window's insets if you want, but that usually isn't necessary. The class `PopupWindow` has an `action()` method that is called when the user clicks one of the buttons. When the user clicks on a button, `action()` prints a message and hides the window.

6.6 Dialogs

The `Dialog` class provides a special type of display window that is normally used for pop-up messages or input from the user. It should be associated with a `Frame` (a required parameter for the constructor), and whenever anything happens to this `Frame`, the same thing will happen to the `Dialog`. For instance, if the parent `Frame` is iconified, the `Dialog` disappears until the `Frame` is de-iconified. If the `Frame` is destroyed, so are all the associated dialogs. Figure 6.5 and Figure 6.6 show typical dialog boxes.

In addition to being associated with a `Frame`, `Dialog` is either modeless or modal. A modeless `Dialog` means a user can interact with both the `Frame` and the `Dialog` at the same time. A modal `Dialog` is one that blocks input to the remainder of the application, including the `Frame`, until the `Dialog` box is acted upon. Note that the parent `Frame` is still executing; unlike some windowing systems, Java does not suspend the entire application for a modal `Dialog`. Normally, blocking access would be done to get input from the user or to show a warning message. Example 6.2 shows how to create and use a modal `Dialog` box, as we will see later in the chapter.

Since `Dialog` subclasses `Window`, its default `LayoutManager` is `BorderLayout`.

In applets, when you create a `Dialog`, you need to provide a reference to the browser's `Frame`, not the applet. In order to get this, you can try to go up the container hierarchy of the `Applet` with `getParent()` until it returns `null`. (You cannot specify a null parent as you can with a `Window`.) See Example 6.1 for a utility method to do this. Simple include a line like the following in your applet:

```
Frame top = ComponentUtilities.getTopLevelParent (this);
```

Then pass `top` to the `Dialog` constructor. Another alternative is to create a new `Frame` to associate with your dialog.

Dialog Constructors and Methods

Constructors

If any constructor is passed a `null` parent, the constructor throws the run-time exception `IllegalArgumentException`.

public Dialog (Frame parent) ★

> This constructor creates an instance of `Dialog` with no title and with `parent` as the `Frame` owning it. It is not modal and is initially resizable.

public Dialog (Frame parent, boolean modal) ☆

> This constructor creates an instance of `Dialog` with no title and with `parent` as the `Frame` owning it. If `modal` is `true`, the `Dialog` grabs all the user input of the program until it is closed. If `modal` is `false`, there is no special behavior associated with the `Dialog`. Initially, the `Dialog` will be resizable. This constructor is comment-flagged as deprecated.

public Dialog (Frame parent, String title) ★

> This version of the constructor creates an instance of `Dialog` with `parent` as the `Frame` owning it and a window title of `title`. It is not modal and is initially resizable.

public Dialog (Frame parent, String title, boolean modal)

> This version of the constructor creates an instance of `Dialog` with `parent` as the `Frame` owning it and a window title of `title`. If `mode` is `true`, the `Dialog` grabs all the user input of the program until it is closed. If `modal` is `false`, there is no special behavior associated with the `Dialog`. Initially, the `Dialog` will be resizable.

> **NOTE:**

In some 1.0 versions of Java, modal dialogs were not supported properly. You needed to create some multithreaded contraption that simulated modality. Modal dialogs work properly in 1.1.

Figure 6.5: A Dialog in an application or local applet

[Graphic: Figure 6-5]

Figure 6.6: The same Dialog in an applet that came across the network

[Graphic: Figure 6-6]

Appearance methods

public String getTitle ()

> The `getTitle()` method returns the current title for the `Dialog`. If there is no title for the `Dialog`, `getTitle()` returns `null`.

public void setTitle (String title)

> The `setTitle()` method changes the current title of the `Dialog` to `title`. To turn off any title for the `Dialog`, use `null` for `title`.

public boolean isResizable ()

> The `isResizable()` method tells you if the current `Dialog` is resizable.

public void setResizable (boolean resizable)

The `setResizable()` method changes the resize state of the `Dialog`. A `resizable` value of `true` means the user can resize the `Dialog`, while `false` means the user cannot. This must be set before the `Dialog` is shown or the peer created.

Modal methods

public boolean isModal ()

The `isModal()` method returns the current mode of the `Dialog`. `true` indicates the dialog traps all user input.

public void setModal (boolean mode) ★

The `setModal()` method changes the current mode of the `Dialog` to `mode`. The next time the dialog is displayed via `show()`, it will be modal. If the dialog is currently displayed, `setModal()` has no immediate effect. The change will take place the next time `show()` is called.

public void show () ★

The `show()` method brings the `Dialog` to the front and displays it. If the dialog is modal, `show()` takes care of blocking events so that they don't reach the parent `Frame`.

Miscellaneous methods

public synchronized void addNotify ()

The `addNotify()` method creates the `Dialog` peer. The peer is created automatically when you call the dialog's `show()` method. If you override the method `addNotify()`, first call `super.addNotify()`, then add your customizations for the new class. You will then be able to do everything you need with the information about the newly created peer.

protected String paramString ()

When you call the `toString()` method of `Dialog`, the default `toString()` method of `Component` is called. This in turn calls `paramString()` which builds up the string to display. At the `Dialog` level, `paramString()` appends the current mode (modal/modeless) and title (if present). Using the constructor ``Dialog (top, `Help`, true)``, the results would be as follows:

```
java.awt.Dialog[0,0,0x0,invalid,hidden,layout=java.awt.BorderLayout,
    modal,title=Help]
```

Dialog Events

In Java 1.0, a Dialog peer generates all the events that are generated by the Component class; it does not generate events that are specific to a particular type of component. That is, it generates key events, mouse events, and focus events; it doesn't generate action events or list events. If an event happens within a child component of a Dialog, the target of the event is the child component, not the Dialog.Window

In addition to the Component events, Dialog generates the WINDOW events. These events are WINDOW_DESTROY, WINDOW_EXPOSE, WINDOW_ICONIFY, WINDOW_DEICONIFY, and WINDOW_MOVED. Listeners and 1.1 event handling

With the 1.1 event model, you register listeners for different event types; the listeners are told when the event happens. The Dialog class inherits all its listener handling from Window.

Dialog Example

Example 6.2 demonstrates how a modal Dialog tries to work in Java 1.0. In some windowing systems, "modal" means that the calling application, and sometimes the entire system stops, and input to anything other than the Dialog is blocked. With Java 1.0, a modal Dialog acts only on the parent frame and simply prevents it from getting screen-oriented input by disabling all components within the frame. The Java program as a whole continues to execute.

Example 6.2 displays a Dialog window with username and password fields, and an Okay button. When the user selects the Okay button, a realistic application would validate the username and password; in this case, they are just displayed on a Frame. Since the Frame must wait for the Dialog to finish before looking at the values of the two fields, the Dialog must tell the Frame when it can look. This is done through a custom interface implemented by the parent Frame and invoked by the Dialog in its action method.

Figure 6.7 is the initial Dialog; Figure 6.8 shows the result after you click Okay. Example 6.2 contains the source code.

Figure 6.7: Username and password Dialog

[Graphic: Figure 6-7]

Notice the use of the newly created `DialogHandler` interface when the user selects the Okay button. Also, see how the pre- and post-event-handling methods are separated. All the pre-event processing takes place before the `Dialog` is shown. The post-event processing is called by the `Dialog` through the new `DialogHandler` interface method, `dialogDoer()`. The interface provides a common method name for all your `Dialog` boxes to call.

Figure 6.8: Resulting Frame

[Graphic: Figure 6-8]

Example 6.2: Modal Dialog Usage

```
import java.awt.*;
interface DialogHandler {
    void dialogDoer (Object o);
}
class modeTest extends Dialog {
    TextField user;
    TextField pass;
    modeTest (DialogHandler parent) {
        super ((Frame)parent, "Mode Test", true);
        add ("North", new Label ("Please enter username/password"));
        Panel left = new Panel ();
        left.setLayout (new BorderLayout ());
        left.add ("North", new Label ("Username"));
        left.add ("South", new Label ("Password"));
        add ("West", left);
        Panel right = new Panel ();
        right.setLayout (new BorderLayout ());
        user = new TextField (15);
        pass = new TextField (15);
        pass.setEchoCharacter ('*');
        right.add ("North", user);
        right.add ("South", pass);
        add ("East", right);
        add ("South", new Button ("Okay"));
        resize (250, 125);
    }
    public boolean handleEvent (Event e) {
        if (e.id == Event.WINDOW_DESTROY) {
            dispose();
            return true;
        } else if ((e.target instanceof Button) &&
             (e.id == Event.ACTION_EVENT)) {
            ((DialogHandler)getParent ()).dialogDoer(e.arg);
        }
        return super.handleEvent (e);
    }
}
public class modeFrame extends Frame implements DialogHandler {
    modeTest d;
    modeFrame (String s) {
        super (s);
        resize (100, 100);
        d = new modeTest (this);
        d.show ();
    }
    public static void main (String []args) {
        Frame f = new modeFrame ("Frame");
    }
    public boolean handleEvent (Event e) {
        if (e.id == Event.WINDOW_DESTROY) {
            hide();
            dispose();
            System.exit (0);
        }
        return super.handleEvent (e);
    }
    public void dialogDoer(Object o) {
        d.dispose();
        add ("North", new Label (d.user.getText()));
        add ("South", new Label (d.pass.getText()));
        show ();
    }
}
```

Since the Java 1.1 modal `Dialog` blocks the calling `Frame` appropriately, the overhead of the `DialogHandler` interface is not necessary and all the work can be combined into the `main()` method, as shown in the following:

```
// only reliable in Java 1.1
import java.awt.*;
class modeTest11 extends Dialog {
    TextField user;
    TextField pass;
    modeTest11 (Frame parent) {
        super (parent, "Mode Test", true);
        add ("North", new Label ("Please enter username/password"));
        Panel left = new Panel ();
        left.setLayout (new BorderLayout ());
        left.add ("North", new Label ("Username"));
        left.add ("South", new Label ("Password"));
        add ("West", left);
        Panel right = new Panel ();
        right.setLayout (new BorderLayout ());
        user = new TextField (15);
        pass = new TextField (15);
        pass.setEchoCharacter ('*');
        right.add ("North", user);
        right.add ("South", pass);
        add ("East", right);
        add ("South", new Button ("Okay"));
        resize (250, 125);
    }
    public boolean handleEvent (Event e) {
        if (e.id == Event.WINDOW_DESTROY) {
            dispose();
            return true;
        } else if ((e.target instanceof Button) &&
             (e.id == Event.ACTION_EVENT)) {
            hide();
        }
        return super.handleEvent (e);
    }
}
public class modeFrame11 extends Frame {
    modeFrame11 (String s) {
        super (s);
        resize (100, 100);
    }
    public static void main (String []args) {
        Frame f = new modeFrame11 ("Frame");
        modeTest11 d;
        d = new modeTest11 (f);
        d.show ();
        d.dispose();
        f.add ("North", new Label (d.user.getText()));
        f.add ("South", new Label (d.pass.getText()));
        f.show ();
    }
    public boolean handleEvent (Event e) {
        if (e.id == Event.WINDOW_DESTROY) {
            hide();
            dispose();
            System.exit (0);
        }
        return super.handleEvent (e);
    }
}
```

The remainder of the code is virtually identical. The most significant difference is that the dialog's `handleEvent()`method just hides the dialog, rather than calling `DialogHandler.dialogDoer()`.

PREVIOUS HOME NEXT
Frames BOOK INDEX FileDialog

6.7 FileDialog

`FileDialog` is a subclass of `Dialog` that lets the user select files for opening or saving. You must load or save any files yourself. If used in an application or *appletviewer*, the `FileDialog` always looks like the local system's file dialog. The `FileDialog` is always a modal `Dialog`, meaning that the calling program is blocked from continuing (and cannot accept input) until the user responds to the `FileDialog`. Figure 6.9 shows the `FileDialog` component in Motif, Windows NT/95, and the Macintosh.

Unlike the other `Window` subclasses, there is no `LayoutManager` for `FileDialog`, since you are creating the environment's actual file dialog. This means you cannot subclass `FileDialog` to alter its behavior or appearance. However, the class is not "final."

> **NOTE:**

Netscape Navigator throws an `AWTError` when you try to create a `FileDialog` because Navigator does not permit local file system access.

FileDialog Methods

Constants

A `FileDialog` has two modes: one for loading a file (input) and one for saving (output). The following variables provide the mode to the constructor. The `FileDialog` functions the same way in both modes. The only visible difference is whether a button on the screen is labeled Load or Save. You must load or save the requested file yourself. On certain platforms there may be functional differences: in `SAVE` mode, the `FileDialog` may ask if you want to replace a file if it already exists; in `LOAD` mode, the `FileDialog` may not accept a filename that does not exist.

public final static int LOAD

> `LOAD` is the constant for load mode. It is the default mode.

public final static int SAVE

> `SAVE` is the constant for save mode.

Constructors

public FileDialog (Frame parent) ★

The first constructor creates a `FileDialog` for loading with a parent `Frame` of `parent`. The window title is initially empty.

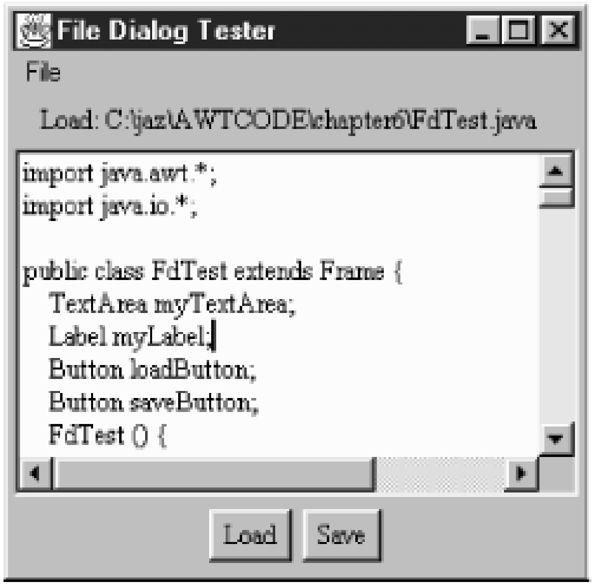
public FileDialog (Frame parent, String title)

This constructor creates a `FileDialog` for loading with a parent `Frame` of `parent`. The window title is `title`.

public FileDialog (Frame parent, String title, int mode)

The final constructor creates a `FileDialog` with an initial mode of `mode`. If `mode` is neither `LOAD` nor `SAVE`, the `FileDialog` is in `SAVE` mode.

Figure 6.9: FileDialogs for Motif, Windows NT/95, and the Macintosh



Load: C:\jaz\AWTCODE\chapter6\FdTest.java

```
import java.awt.*;
import java.io.*;

public class FdTest extends Frame {
    TextArea myTextArea;
    Label myLabel;
    Button loadButton;
    Button saveButton;
    FdTest () {
```

Appearance methods

public String getDirectory ()

`getDirectory()` returns the current directory for the `FileDialog`. Normally, you check this when `FileDialog` returns after a `show()` and a call to `getFile()` returns something other than `null`.

public void setDirectory (String directory)

The `setDirectory()` method changes the initial directory displayed in the `FileDialog` to `directory`. You must call `setDirectory()` prior to displaying the `FileDialog`.

public String getFile ()

The `getFile()` method returns the current file selection from the `FileDialog`. If the user pressed the Cancel button on the `FileDialog`, `getFile()` returns `null`. This is the only way to determine if the user pressed Cancel.

NOTE:

On some platforms in Java 1.0 `getFile()` returns a string that ends in `.*.*` (two periods and two asterisks) if the file does not exist. You need to remove the extra characters before you can create the file.

public void setFile (String file)

The `setFile()` method changes the default file for the `FileDialog` to `file`. Because the `FileDialog` is modal, this must be done before you call `show()`. The string may contain a filename filter like **.java* to show a preliminary list of files to select. This has nothing to do with the use of the `FilenameFilter` class.

public FilenameFilter getFilenameFilter ()

The `getFilenameFilter()` method returns the current `FilenameFilter`. The `FilenameFilter` class is part of the `java.io` package. `FilenameFilter` is an interface that allows you to restrict choices to certain directory and filename combinations. For example, it can be used to limit the user to selecting *.jpg*, *.gif*, and *.xbm* files. The class implementing `FilenameFilter` would not return other possibilities as choices.

public void setFilenameFilter (FilenameFilter filter)

The `setFilenameFilter()` method changes the current filename filter to `filter`. This needs to be done before you `show()` the `FileDialog`.

NOTE:

The JDK does not support the `FilenameFilter` with `FileDialog` boxes. `FilenameFilter` works but can't be used with `FileDialog`.

Miscellaneous methods

public int getMode ()

The `getMode()` method returns the current mode of the `FileDialog`. If an invalid mode was used in the constructor, this method returns an invalid mode here. No error checking is performed.

public void setMode (int mode) ★

The `setMode()` method changes the current mode of the `FileDialog` to `mode`. If `mode` is not one of the class constants `LOAD` or `SAVE`, `setMode()` throws the run-time exception `IllegalArgumentException`.

public synchronized void addNotify ()

The `addNotify()` method creates the `FileDialog` peer. This is automatically done when you call the `show()` method of the `FileDialog`. If you override this method, first call `super.addNotify()`, then add your customizations for the new class. Then you can do everything you need with the information about the newly created peer.

protected String paramString ()

When you call the `toString()` method of `FileDialog`, the default `toString()` method of `Component` is called. This in turn calls `paramString()`, which builds up the string to display. At the `FileDialog` level, `paramString()` appends the directory (if not `null`) and current mode to the return value. Using the constructor `FileDialog(top, `Load Me`)`, the results would be as follows:

```
java.awt.FileDialog[0,0,0x0,invalid,hidden,modal,title=Load Me,load]
```

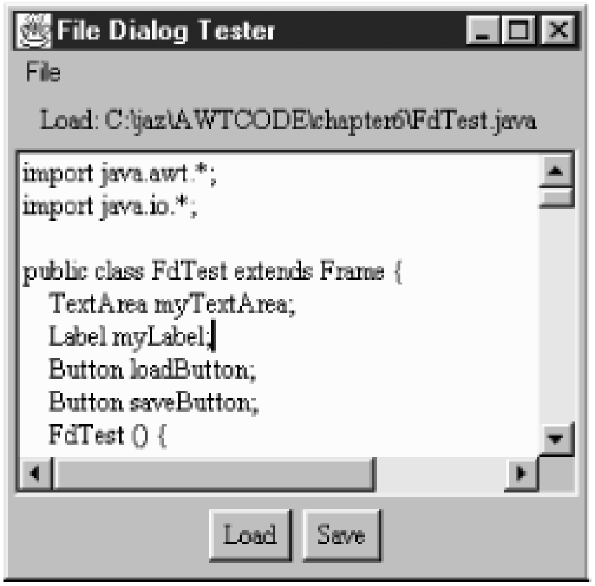
A FileDialog Example

To get a better grasp of how the `FileDialog` works, the following application uses a `FileDialog` to select a file for display in a `TextArea`. You can also use `FileDialog` to save the file back to disk. Figure 6.10 shows the application, with a file displayed in the text area; the `FileDialog` itself looks like any other file dialog on the run-time system. Example 6.3 shows the code.

CAUTION:

This example can overwrite an existing file.

Figure 6.10: FileDialog test program



Load: C:\jaz\AWTCODE\chapter6\FdTest.java

```
import java.awt.*;
import java.io.*;

public class FdTest extends Frame {
    TextArea myTextArea;
    Label myLabel;
    Button loadButton;
    Button saveButton;
    FdTest () {
```

Example 6.3: Complete FileDialog

```
import java.awt.*;
import java.io.*;
public class FdTest extends Frame {
    TextArea myTextArea;
```

```
Label myLabel;
Button loadButton;
Button saveButton;
FdTest () {
    super ("File Dialog Tester");
    Panel p = new Panel ();
    p.add (loadButton = new Button ("Load"));
    p.add (saveButton = new Button ("Save"));
    add ("North", myLabel = new Label ());
    add ("South", p);
    add ("Center", myTextArea = new TextArea (10, 40));
    Menu m = new Menu ("File");
    m.add (new MenuItem ("Quit"));
    MenuBar mb = new MenuBar();
    mb.add (m);
    setMenuBar (mb);
    pack();
}
public static void main (String args[]) {
    FdTest f = new FdTest();
        f.show();
}
public boolean handleEvent (Event e) {
    if (e.id == Event.WINDOW_DESTROY) {
        hide();
        dispose ();
        System.exit(0);
        return true;  // never gets here
    }
    return super.handleEvent (e);
}
public boolean action (Event e, Object o) {
    if (e.target instanceof MenuItem) {
        hide();
        dispose ();
        System.exit(0);
        return true;  // never gets here
    } else if (e.target instanceof Button) {
        int state;
        String msg;
        if (e.target == loadButton) {
            state = FileDialog.LOAD;
            msg = "Load File";
        } else {// if (e.target == saveButton)
            state = FileDialog.SAVE;
            msg = "Save File";
        }
        FileDialog file = new FileDialog (this, msg, state);
        file.setFile ("*.java");  // set initial filename filter
```

```
            file.show(); // Blocks
            String curFile;
            if ((curFile = file.getFile()) != null) {
                String filename = file.getDirectory() + curFile;
                // curFile ends in .*.* if file does not exist
                byte[] data;
                setCursor (Frame.WAIT_CURSOR);
                if (state == FileDialog.LOAD) {
                    File f = new File (filename);
                    try {
                        FileInputStream fin = new FileInputStream (f);
                        int filesize = (int)f.length();
                        data = new byte[filesize];
                        fin.read (data, 0, filesize);
                    } catch (FileNotFoundException exc) {
                        String errorString = "File Not Found: " + filename;
                        data = new byte[errorString.length()];
                        errorString.getBytes (0, errorString.length(), data, 0);
                    } catch (IOException exc) {
                        String errorString = "IOException: " + filename;
                        data = new byte[errorString.length()];
                        errorString.getBytes (0, errorString.length(), data, 0);
                    }
                    myLabel.setText ("Load: " + filename);
                } else {
// Remove trailing ".*.*" if present - signifies file does not exist
                    if (filename.indexOf (".*.*") != -1) {
                        filename = filename.substring (0, filename.length()-4);
                    }
                    File f = new File (filename);
                    try {
                        FileOutputStream fon = new FileOutputStream (f);
                        String text = myTextArea.getText();
                        int textsize = text.length();
                        data = new byte[textsize];
                        text.getBytes (0, textsize, data, 0);
                        fon.write (data);
                        fon.close ();
                    } catch (IOException exc) {
                        String errorString = "IOException: " + filename;
                        data = new byte[errorString.length()];
                        errorString.getBytes (0, errorString.length(), data, 0);
                    }
                    myLabel.setText ("Save: " + filename);
                }
                // Note - on successful save, text is redisplayed
                myTextArea.setText (new String (data, 0));
                setCursor (Frame.DEFAULT_CURSOR);
            }
            return true;
        }
        return false;
    }
}
```

Most of this application is one long `action()` method that handles all the action events that take place within the `Frame`. The constructor doesn't do much besides arrange the display; it includes code to create a File menu with one item, Quit. This menu is visible in the upper left corner of the `Frame`; we'll see more about working with menus in Chapter 10, *Would You Like to Choose from the Menu?* We provide a `main()` method to display the `Frame` and a `handleEvent()` method to shut the application down if the event `WINDOW_DESTROY` occurs.

But the heart of this program is clearly its `action()` method. `action()` starts by checking whether the user selected a menu item; if so, it shuts down the application because the only item on our menu is Quit. It then checks whether the user clicked on one of the buttons and sets the `FileDialog` mode to `LOAD` or `SAVE` accordingly. It then sets a default filename, **.java*, which limits the display to filenames ending in *.java*. Next, `action()` shows the dialog. Because file dialogs are modal, `show()` blocks until the user selects a file or clicks Cancel.

The next line detects whether or not `getFile()` returns `null`. A `null` return indicates that the user selected Cancel; in this case, the dialog disappears, but nothing else happens. We then build a complete filename from the directory name and the name the user selected. If the dialog's state is `LOAD`, we read the file and display it in the text area. Otherwise, the dialog's state must be `SAVE`, so we save the contents of the text area under the given filename. Note that we first check for the string `*.*` and remove it if it is present. In Java 1.1, these two lines are unnecessary, but they don't hurt, either.

7.2 FlowLayout

`FlowLayout` is the default `LayoutManager` for a `Panel`. A `FlowLayout` adds components to the container in rows, working from left to right. When it can't fit any more components in a row, it starts a new row--not unlike a word processor with word wrap enabled. When the container gets resized, the components within it get repositioned based on the container's new size. If sufficient space is available, components within `FlowLayout` containers are given their preferred size. If there is insufficient space, you do not see the components in their entirety.

FlowLayout Methods

Constants

`FlowLayout` defines three constants, all of which are used to specify alignment. The alignment tells `FlowLayout` where to start positioning the components on each row. Each component is still added from left to right, no matter what the alignment setting is.

public final static int LEFT

> `LEFT` is the constant for left alignment.

public final static int CENTER

> `CENTER` is the constant for center alignment and is the default.

public final static int RIGHT

> `RIGHT` is the constant for right alignment.

Constructors

public FlowLayout ()

This constructor creates a `FlowLayout` using default settings: center alignment with a horizontal and vertical gap of five pixels. The gap is the space between the different components in the different directions. By default, there will be five pixels between components. The constructor is usually called within a call to `setLayout()`: `setLayout (new FlowLayout())`. Figure 7.1 shows how the default `FlowLayout` behaves with different screen sizes. As the screen C shows, if the screen is too small, the components will *not* be shrunk so that they can fit better.

Figure 7.1: FlowLayout with six buttons and three different screen sizes

[Graphic: Figure 7-1]

public FlowLayout (int alignment)

This version of the constructor creates a `FlowLayout` using the specified `alignment` and a horizontal and vertical gap of five pixels. Valid alignments are the `FlowLayout` constants, although there is no verification. Figure 7.2 shows the effect of different alignments: `FlowLayout.LEFT` (screen A), `FlowLayout.CENTER` (B), and `FlowLayout.RIGHT` (C).

Figure 7.2: FlowLayout with three different alignments

[Graphic: Figure 7-2]

public FlowLayout (int alignment, int hgap, int vgap)

> The final version of the constructor is called by the other two. It requires you to explicitly specify the alignment, horizontal gap (`hgap`), and vertical gap (`vgap`). This creates a `FlowLayout` with an alignment of `alignment`, horizontal gap of `hgap`, and vertical gap of `vgap`. The units for gaps are pixels. It is possible to have negative gaps if you want components to be placed on top of one another. Figure 7.3 shows the effect of changing the gap sizes.

Figure 7.3: FlowLayout with hgap of 0 and vgap of 20

[Graphic: Figure 7-3]

Informational methods

public int getAlignment () ★

> The `getAlignment()` method retrieves the current alignment of the `FlowLayout`. The return value should equal one of the class constants `LEFT`, `CENTER`, or `RIGHT`.

public void setAlignment (int alignment) ★

The `setAlignment()` method changes the `FlowLayout` alignment to `alignment`. The alignment value should equal one of the class constants `LEFT`, `CENTER`, or `RIGHT`, but this method does not check. After changing the alignment, you must `validate()` the `Container`.

public int getHgap () ★

The `getHgap()` method retrieves the current horizontal gap setting.

public void setHgap (int hgap) ★

The `setHgap()` method changes the current horizontal gap setting to `hgap`. After changing the gaps, you must `validate()` the `Container`.

public int getVgap () ★

The `getVgap()` method retrieves the current vertical gap setting.

public void setVgap (int hgap) ★

The `setVgap()` method changes the current vertical gap setting to `vgap`. After changing the gaps, you must `validate()` the `Container`.

LayoutManager methods

public void addLayoutComponent (String name, Component component)

The `addLayoutComponent()` method of `FlowLayout` does nothing.

public void removeLayoutComponent (Component component)

The `removeLayoutComponent()` method of `FlowLayout` does nothing.

public Dimension preferredLayoutSize (Container target)

The `preferredLayoutSize()` method of `FlowLayout` calculates the preferred dimensions for the `target` container. The `FlowLayout` computes the preferred size by placing all the components in one row and adding their individual preferred sizes along with gaps and insets.

public Dimension minimumLayoutSize (Container target)

The `minimumLayoutSize()` method of `FlowLayout` calculates the minimum dimensions for the container by adding up the sizes of the components. The `FlowLayout` computes the minimum size by placing all the components in one row and adding their individual minimum sizes along with gaps and insets.

public void layoutContainer (Container target)

The `layoutContainer()` method draws `target`'s components on the screen, starting with the first row of the display, going left to right across the screen, based on the current alignment setting. When it reaches the right margin of the container, it skips down to the next row, and continues drawing additional components.

Miscellaneous methods

public String toString ()

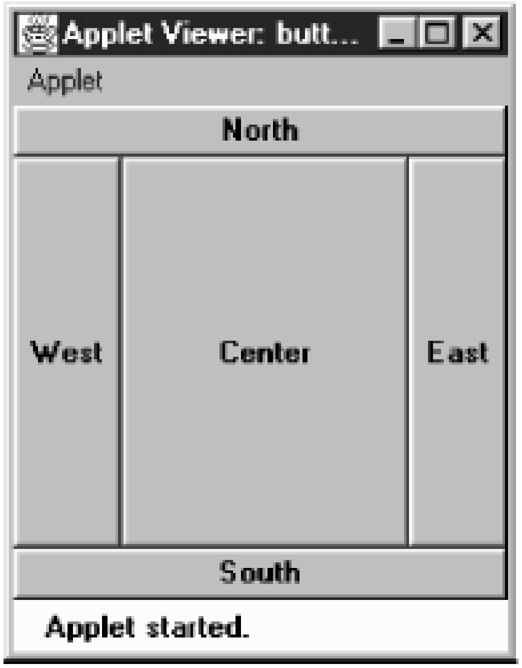
The `toString()` method of `FlowLayout` returns the current horizontal and vertical gap settings along with the alignment (left, center, right). For a `FlowLayout` that uses all the defaults, `toString()` produces:

```
java.awt.FlowLayout[hgap=5,vgap=5,align=center]
```

7.3 BorderLayout

`BorderLayout` is the default `LayoutManager` for a `Window`. It provides a very flexible way of positioning components along the edges of the window. The following call to `setLayout()` changes the `LayoutManager` of the current container to the default `BorderLayout`: `setLayout(new BorderLayout())`. Figure 7.4 shows a typical `BorderLayout`.

Figure 7.4: BorderLayout



Applet started.

`BorderLayout` is the only layout provided that requires you to name components when you add them to the layout; if you're using a `BorderLayout`, you must use `add(String name, Component component)` in Java 1.0 or `add(Component component, String name)` in Java 1.1 (parameter order switched). (The `CardLayout` can use these versions of `add()`, but does not require it.) The `name` parameter of `add()` specifies the region to which the component should be added. The five different regions are "North", "South", "East", and "West" for the edges of the window, and "Center" for any remaining interior space. These names are case sensitive. It is not necessary that a container use all five regions. If a region is not used, it relinquishes its space to the regions around it. If you `add()` multiple objects to a single region, the layout manager only displays the last one. If you want to display multiple objects within a region, group them within a `Panel` first, then `add()` the `Panel`.

NOTE:

In Java 1.1, if you do not provide a name, the component is placed in the "Center" region.

BorderLayout Methods

Constants

Prior to Java 1.1, you had to use string constants to specify the constraints when adding a component to a container whose layout is `BorderLayout`. With Java 1.1, you can use class constants, instead of a literal string, in the following list.

public static final String CENTER ★

The `CENTER` constant represents the "Center" string and indicates that a component should be added to the center region.

public static final String EAST ★

The `EAST` constant represents the "East" string and indicates that a component should be added to the east region.

public static final String NORTH ★

The `NORTH` constant represents the "North" string and indicates that a component should be added to the north region.

public static final String SOUTH ★

The `SOUTH` constant represents the "South" string and indicates that a component should be added to the south region.

public static final String WEST ★

The `WEST` constant represents the "West" string and indicates that a component should be added to the west region.

Constructors

public BorderLayout ()

This constructor creates a `BorderLayout` using a default setting of zero pixels for the horizontal and vertical gaps. The gap specifies the space between adjacent components. With horizontal and vertical gaps of zero, components in adjacent regions will touch each other. As Figure 7.4 shows, each component within a `BorderLayout` will be resized to fill an entire region.

public BorderLayout (int hgap, int vgap)

This version of the constructor allows you to create a `BorderLayout` with a horizontal gap of `hgap` and vertical gap of `vgap`, putting some space between the different components. The units for gaps are pixels. It is possible to have negative gaps if you want components to overlap.

Informational methods

public int getHgap () ★

The `getHgap()` method retrieves the current horizontal gap setting.

public void setHgap (int hgap) ★

The `setHgap()` method changes the current horizontal gap setting to `hgap`. After changing the gaps, you must `validate()` the `Container`.

public int getVgap () ★

The `getVgap()` method retrieves the current vertical gap setting.

public void setVgap (int hgap) ★

The `setVgap()` method changes the current vertical gap setting to `vgap`. After changing the gaps, you must `validate()` the `Container`.

LayoutManager methods

public void addLayoutComponent (String name, Component component) ☆

This version of `addLayoutComponent()` has been deprecated and replaced by the `addLayoutComponent(Component, Object)` method of the `LayoutManager2` interface.

public void removeLayoutComponent (Component component)

The `removeLayoutComponent()` method of `BorderLayout` removes `component` from the container, if it is in one of the five regions. If `component` is not in the container already, nothing happens.

public Dimension preferredLayoutSize (Container target)

The `preferredLayoutSize()` method of `BorderLayout` calculates the preferred dimensions for the components in `target`. To compute the preferred height, a `BorderLayout` adds the height of the `getPreferredSize()` of the north and south components to the maximum `getPreferredSize()` height of the east, west, and center components. The vertical gaps are added in for the north and south components, if present. The top and bottom insets are also added into the height. To compute the preferred width, a `BorderLayout` adds the width of the `getPreferredSize()` of east, west, and center components, along with the horizontal gap for the east and west regions. It compares this value to the preferred widths of the north and south components. The `BorderLayout` takes the maximum of these three and then adds the left and right insets, plus twice the horizontal gap. The result is the preferred width for the container.

public Dimension minimumLayoutSize (Container target)

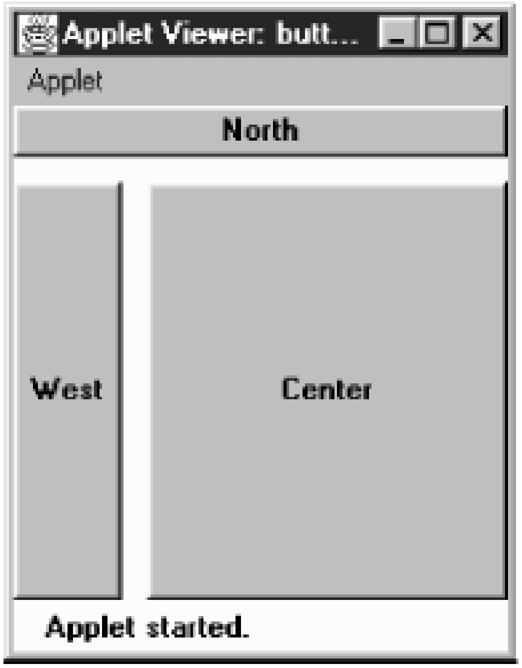
The `minimumLayoutSize()` method of `BorderLayout` calculates the minimum dimensions for the components in `target`. To compute the minimum height, a `BorderLayout` adds the height of the `getMinimumSize()` of the north and south components to the maximum of the minimum heights of the east, west, and center components. The vertical gaps are added in for the north and south components, if present, along with the container's top and bottom insets. To compute the minimum width, a `BorderLayout` adds the width of the `getMinimumSize()` of east, west, and center components, along with the horizontal gap for the east and west regions. The `BorderLayout` takes the maximum of these three and then adds the left and right insets, plus twice the horizontal gap. The result is the minimum width for the container.

public void layoutContainer (Container target)

The `layoutContainer()` method draws `target`'s components on the screen in the appropriate regions. The north region takes up the entire width of the container along the top.

South does the same along the bottom. The heights of north and south will be the heights of the components they contain. The east and west regions are given the widths of the components they contain. For height, east and west are given whatever is left in the container after satisfying north's and south's height requirements. If there is any extra vertical space, the east and west components are resized accordingly. Any space left in the middle of the screen is assigned to the center region. If there is insufficient space for all the components, space is allocated according to the following priority: north, south, west, east, and center. Unlike `FlowLayout`, `BorderLayout` reshapes the internal components of the container to fit within their region. Figure 7.5 shows what happens if the east and south regions are not present and the gaps are nonzero.

Figure 7.5: BorderLayout with missing regions



LayoutManager2 methods

public void addLayoutComponent (Component component, Object name) ★

This `addLayoutComponent()` method puts `component` in the `name` region of the container. In Java 1.1, if `name` is null, `component` is added to the center. If the name is not "North", "South", "East", "West", or "Center", the component is added to the container but won't be displayed. Otherwise, it is displayed in the appropriate region.

There can only be one component in any region, so any component already in the named region is removed. To get multiple components in one region of a `BorderLayout`, group the components in another container, and add the container as a whole to the layout.

If `name` is not a `String`, `addLayoutComponent()` throws the run-time exception

`IllegalArgumentException`.

public abstract Dimension maximumLayoutSize(Container target) ★

The `maximumLayoutSize()` method returns a `Dimension` object with a width and height of `Integer.MAX_VALUE`. In effect, this means that `BorderLayout` does not support the concept of maximum size.

public abstract float getLayoutAlignmentX(Container target) ★

The `getLayoutAlignmentX()` method says that `BorderLayout` containers should be centered horizontally within the area available.

public abstract float getLayoutAlignmentY(Container target) ★

The `getLayoutAlignmentY()` method says that `BorderLayout` containers should centered vertically within the area available.

public abstract void invalidateLayout(Container target) ★

The `invalidateLayout()` method of `BorderLayout` does nothing.

Miscellaneous methods

public String toString ()

The `toString()` method of `BorderLayout` returns a string showing the current horizontal and vertical gap settings. If both gaps are zero, the result will be:

```
java.awt.BorderLayout[hgap=0,vgap=0]
```

7.4 GridLayout

The `GridLayout` layout manager is ideal for laying out objects in rows and columns, where each cell in the layout has the same size. Components are added to the layout from left to right, top to bottom. `setLayout(new GridLayout(2,3))` changes the `LayoutManager` of the current container to a 2 row by 3 column `GridLayout`. Figure 7.6 shows an applet using this layout.

Figure 7.6: Applet using GridLayout

[Graphic: Figure 7-6]

GridLayout Methods

Constructors

public GridLayout () ★

> This constructor creates a `GridLayout` initially configured to have one row, an infinite number of columns, and no gaps. A gap is the space between adjacent components in the horizontal or vertical direction. With a gap of zero, components in adjacent cells will have no space between them.

public GridLayout (int rows, int columns)

This constructor creates a `GridLayout` initially configured to be `rows` x `columns` in size. The default setting for horizontal and vertical gaps is zero pixels. The gap is the space between adjacent components in the horizontal and vertical directions. With a gap of zero, components in adjacent cells will have no space between them.

You can set the number of rows or columns to zero; this means that the layout will grow without bounds in that direction. If both `rows` and `columns` are zero, the run-time exception `IllegalArgumentException` will be thrown.

NOTE:

The rows and columns passed to the `GridLayout` constructor are only recommended values. It is possible that the system will pick other values if the number of objects you add to the layout is sufficiently different from the size you requested; for example, you placed nine objects in a six-element grid.

public GridLayout (int rows, int columns, int hgap, int vgap)

This version of the constructor is called by the previous one. It creates a `GridLayout` with an initial configuration of `rows` x `columns`, with a horizontal gap of `hgap` and vertical gap of `vgap`. The gap is the space between the different components in the different directions, measured in pixels. It is possible to have negative gaps if you want components to overlap.

You can set the number of rows or columns to zero; this means that the layout will grow without bounds in that direction. If both `rows` and `columns` are zero, the run-time exception `IllegalArgumentException` will be thrown.

Informational methods

public int getColumns () ★

The `getColumns()` method retrieves the current column setting, which may differ from the number of columns displayed.

public void setColumns (int columns) ★

The `setColumns()` method changes the current column setting to `columns`. After changing the setting, you must `validate()` the `Container`. If you try to set the number of rows and the number of columns to zero, this method throws the run-time exception

`IllegalArgumentException`.

public int getRows () ★

The `getRows()` method retrieves the current row setting; this may differ from the number of rows displayed.

public void setRows (int rows) ★

The `setRows()` method changes the current row setting to `rows`. After changing the setting, you must `validate()` the `Container`. If you try to set the number of rows and the number of columns to zero, this method throws the run-time exception `IllegalArgumentException`.

public int getHgap () ★

The `getHgap()` method retrieves the current horizontal gap setting.

public void setHgap (int hgap) ★

The `setHgap()` method changes the current horizontal gap setting to `hgap`. After changing the gaps, you must `validate()` the `Container`.

public int getVgap () ★

The `getVgap()` method retrieves the current vertical gap setting.

public void setVgap (int hgap) ★

The `setVgap()` method changes the current vertical gap setting to `vgap`. After changing the gaps, you must `validate()` the `Container`.

LayoutManager methods

public void addLayoutComponent (String name, Component component)

The `addLayoutComponent()` method of `GridLayout` does nothing.

public void removeLayoutComponent (Component component)

The `removeLayoutComponent()` method of `GridLayout` does nothing.

public Dimension preferredLayoutSize (Container target)

The `preferredLayoutSize()` method of `GridLayout` calculates the preferred dimensions for the components in `target`. The preferred size depends on the size of the grid, which may not be the size requested by the constructor; the `GridLayout` treats the constructor's arguments as recommendations and may ignore them if appropriate.

The actual number of rows and columns is based upon the number of components within the `Container`. The `GridLayout` tries to observe the number of rows requested first, calculating the number of columns. If the requested number of rows is nonzero, the number of columns is determined by (# components + rows - 1) / rows. If request is for zero rows, the number of rows to use is determined by a similar formula: (# components + columns - 1) / columns. Table 7.1 demonstrates this calculation. The last entry in this table is of special interest: if you request a 3x3 grid but only place four components in the layout, you get a 2x2 layout as a result. If you do not want to be surprised, size the `GridLayout` based on the number of objects you plan to put into the display.

Table 7.1: GridLayout Row/Column Calculation

Rows	Columns	# Components	Display Rows	Display Columns
0	1	10	10	1
0	2	10	5	2
1	0	10	1	10
2	0	10	2	5
2	3	10	2	5
2	3	20	2	10
3	2	10	3	4
3	3	3	3	1
3	3	4	2	2

Once we know the dimensions of the grid, it's easy to compute the preferred size for the layout. The `GridLayout` takes the maximum height and maximum width of the preferred sizes for all the components in the layout. (Note that the maximum width and maximum height aren't necessarily from the same component.) This becomes the preferred size of each cell within the layout. The preferred size of the layout as a whole is computed using the preferred size of a cell and adding gaps and insets as appropriate.

public Dimension minimumLayoutSize (Container target)

The `minimumLayoutSize()` method of `GridLayout` calculates the minimum dimensions for the components in `target`. First it determines the actual number of rows and columns in the final layout, using the method described previously. The `minimumLayoutSize()` method then determines the widest and tallest `getMinimumSize()` of a component, and this becomes the minimum size of a cell within the layout. The minimum size of the layout as a whole is computed using the minimum size of a cell and adding gaps and insets as appropriate.

public void layoutContainer (Container target)

The `layoutContainer()` method draws `target`'s components on the screen in a series of rows and columns. Each component within a `GridLayout` will be the same size, if it is possible. If there is insufficient space for all the components, the size of each is reduced proportionally.

Miscellaneous methods

public String toString ()

The `toString()` method of `GridLayout` returns a string including the current horizontal and vertical gap settings, along with the rows and columns settings. For a `GridLayout` created with 2 rows and 3 columns, the result would be:

```
java.awt.GridLayout[hgap=0,vgap=0,rows=2,cols=3]
```

PREVIOUS HOME NEXT
BorderLayout BOOK INDEX CardLayout

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

7.5 CardLayout

The CardLayout layout manager is significantly different from the other layouts. Whereas the other layout managers attempt to display all the components within the container at once, a CardLayout displays only one component at a time. (That component could be a Component or another Container.) The result is similar to Netscape Navigator's Property sheets or a tabbed Dialog, without the tabs. You can flip through the cards (components) in the layout in order or jump to a specific card if you know its name. The following call to setLayout() changes the LayoutManager of the current container to CardLayout:

```
lm = new CardLayout();
setLayout (lm);
```

Unlike most other layout managers, CardLayout has a number of instance methods that programs have to call. Therefore, you usually have to retain a reference to the layout manager. In addition, you usually have some other component to control the CardLayout (i.e., select which card to view). Most simply, you could put some buttons in a panel and stick this panel in the north region of a BorderLayout; then make another panel with a CardLayout, and place that in the center. A more complex task would be to build a set of tabs to control the CardLayout.

A CardLayout allows you to assign names to the components it manages. You can use the name to jump to an arbitrary component by calling the manager's show() method. In Java 1.0, naming was optional; you could call add(Component) to put a component in the layout with a null name. A null name meant only that you couldn't flip to the component at will; you could only display the component by calling next() or previous() (or first() or last()), which cycle through all the components in order. In Java 1.1, all components added to a CardLayout must be named.

CardLayout Methods

Constructors

public CardLayout ()

This constructor creates a `CardLayout` using a horizontal and vertical gap of zero pixels. With `CardLayout`, there is no space between components because only one component is visible at a time; think of the gaps as insets.

public CardLayout (int hgap, int vgap)

This version of the constructor allows you to create a `CardLayout` with a horizontal gap of `hgap` and vertical gap of `vgap` to add some space around the outside of the component that is displayed. The units for gaps are pixels. Using negative gaps chops off components at the edges of the container.

Informational methods

public int getHgap () ★

The `getHgap()` method retrieves the current horizontal gap setting.

public void setHgap (int hgap) ★

The `setHgap()` method changes the current horizontal gap setting to `hgap`. After changing the gaps, you must `validate()` the `Container`.

public int getVgap () ★

The `getVgap()` method retrieves the current vertical gap setting.

public void setVgap (int hgap) ★

The `setVgap()` method changes the current vertical gap setting to `vgap`. After changing the gaps, you must `validate()` the `Container`.

LayoutManager methods

public void addLayoutComponent (String name, Component component) ☆

This version of `addLayoutComponent()` has been deprecated and replaced by the `addLayoutComponent(Component, Object)` method of the `LayoutManager2` interface.

public void removeLayoutComponent (Component component)

The `removeLayoutComponent()` method of `CardLayout` removes `component` from the container. If `component` is not in the container already, nothing happens.

public Dimension preferredLayoutSize (Container target)

The `preferredLayoutSize()` method of `CardLayout` retrieves the preferred size for all the components within it. The `preferredLayoutSize()` method then determines the widest and tallest size of all components (not necessarily from the same one), adds the appropriate insets and gaps, and uses that as the preferred size for the layout.

public Dimension minimumLayoutSize (Container target)

The `minimumLayoutSize()` method of `CardLayout` calculates the minimum size for all the components within it. The `minimumLayoutSize()` method then determines the widest and tallest minimum size of all components (not necessarily from the same one), adds the appropriate insets and gaps, and uses that as the minimum size for the layout.

public void layoutContainer (Container target)

The `layoutContainer()` method draws `target`'s visible components one on top of another. Initially, all components are visible. Components do not become invisible until you select one for display, by calling the `first()`, `last()`, `next()`, `previous()`, or `show()` methods. Where possible, `CardLayout` reshapes all components to fit the target container.

LayoutManager2 methods

public void addLayoutComponent (Component component, Object name) ★

This `addLayoutComponent()` method of `CardLayout` puts `component` into an internal table with a key of `name`. The `name` comes from the version of `add()` that has a constraints object as a parameter. The name allows you to refer to the component when you call other card layout methods, like `show()`. If you call the version of `add()` that only takes a `Component` parameter, you cannot call the `show()` method to flip to the specific component.

If `name` is not a `String`, the run-time exception `IllegalArgumentException` is thrown.

public abstract Dimension maximumLayoutSize(Container target) ★

The `maximumLayoutSize()` method returns a `Dimension` object with a width and height of `Integer.MAX_VALUE`. In practice, this means that `CardLayout` doesn't support the concept of maximum size.

public abstract float getLayoutAlignmentX(Container target) ★

The `getLayoutAlignmentX()` method says that `CardLayout` containers should be centered horizontally within the area available.

public abstract float getLayoutAlignmentY(Container target) ★

The `getLayoutAlignmentY()` method says that `CardLayout` containers should be centered vertically within the area available.

public abstract void invalidateLayout(Container target) ★

The `invalidateLayout()` method of `CardLayout` does nothing.

CardLayout methods

This group of methods controls which component the `CardLayout` displays. The `show()` is only usable if you assigned components names when adding them to the container. The others can be used even if the components are unnamed; they cycle through the components in the order in which they were added. All of these methods require the parent `Container` (i.e., the container being managed by this layout manager) as an argument. If the layout manager of the `parent` parameter is anything other than the container using this instance of the `CardLayout`, the method throws the run-time exception `IllegalArgumentException`.

public void first (Container parent)

The `first()` method flips to the initial component in `parent`.

public void next (Container parent)

The `next()` method flips to the following component in `parent`, wrapping back to the beginning if the current component is the last.

public void previous (Container parent)

The `previous()` method flips to the prior component in `parent`, wrapping to the end if the current component is the first.

public void last (Container parent)

The `last()` method flips to the final component in `parent`.

public void show (Container parent, String name)

The `show()` method displays the component in parent that was assigned the given `name` when it was added to the container. If there is no component with `name` contained within `parent`, nothing happens.

Miscellaneous methods

public String toString ()

The `toString()` method of `CardLayout` returns the a string showing the current horizontal and vertical gap settings. The result for a typical `CardLayout` would be:

```
java.awt.CardLayout[hgap=0,vgap=0]
```

CardLayout Example

Figure 7.7 shows a simple `CardLayout`. This layout has three cards that cycle when you make a selection. The first card (A) contains some `Checkbox` items within a `Panel`, the second card (B) contains a single `Button`, and the third (C) contains a `List` and a `Choice` within another `Panel`.

Figure 7.7: Different views of CardLayout

[Graphic: Figure 7-7]

Example 7.1 is the code that generated Figure 7.7.

Example 7.1: The CardExample Class

```
import java.awt.*;
import java.applet.*;
public class CardExample extends Applet {
    CardLayout cl = new CardLayout();
    public void init () {
        String fonts[] = Toolkit.getDefaultToolkit().getFontList();
        setLayout (cl);
        Panel pA = new Panel();
        Panel pC = new Panel ();
        p1.setLayout (new GridLayout (3, 2));
        List l = new List(4, false);
        Choice c = new Choice ();
        for (int i=0;i<fonts.length;i++) {
            pA.add (new Checkbox (fonts[i]));
            l.addItem (fonts[i]);
            c.addItem (fonts[i]);
        }
        pC.add (l);
        pC.add (c);
        add ("One", pA);
        add ("Two", new Button ("Click Here"));
        add ("Three", pC);
    }
    public boolean action (Event e, Object o) {
        cl.next(this);
        return true;
    }
}
```

Each panel within the `CardLayout` has its own layout manager. `Panel` A uses a `GridLayout`; panel C uses its default layout manager, which is a `FlowLayout`. When the user takes any action (i.e., clicking on a checkbox or button, or selecting an item from the `List` or `Choice` components), the system generates a call to `action()`, which calls the `CardLayout`'s `next()` method, thus displaying the next card in the sequence.

PREVIOUS HOME NEXT

GridLayout BOOK INDEX GridBagLayout

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

7.6 GridBagLayout

The `GridBagLayout` is the most complex and flexible of the standard layout managers. Although it sounds like it should be a subclass of `GridLayout`, it's a different animal entirely. With `GridLayout`, elements are arranged in a rectangular grid, and each element in the container is sized identically (where possible). With `GridBagLayout`, elements can have different sizes and can occupy multiple rows or columns. The position and behavior of each element is specified by an instance of the `GridBagConstraints` class. By properly constraining the elements, you can specify the number of rows and columns an element occupies, which element grows when additional screen real estate is available, and various other restrictions. The actual grid size is based upon the number of components within the `GridBagLayout` and the `GridBagConstraints` of those objects. For example, Figure 7.8 shows a `GridBagLayout` with seven components, arranged on a 3x3 grid. The maximum capacity of a screen using `GridBagLayout` in Java 1.0 is 128 x 128 cells; in Java 1.1, the maximum size is 512 x 512 cells.

Figure 7.8: GridBagLayout with seven components on a 3x3 grid

[Graphic: Figure 7-8]

With the other layout managers, adding a component to the container requires only a call to `add()`. In Java 1.0, the `GridBagLayout` also requires you to call `setConstraints()` to tell the layout manager how to position the component. With Java 1.1, you use the new `add()` method that permits you to pass the component and its constraints in a single method call (`add(Component, Object)`). If no components are added with constraints (thus all using the defaults), the `GridBagLayout` places the components in a single row at the center of the screen and sizes them to their `getPreferredSize()`. This is a nice way to place a single object in the center of the screen without stretching it to take up the available space, as `BorderLayout` does. Figure 7.9 compares the default `GridBagLayout` with a `BorderLayout` displaying the same object in the center region.

Figure 7.9: Centering a component: GridBagLayout vs. BorderLayout

[Graphic: Figure 7-9]

When designing a container that will use `GridBagLayout`, it is easiest to plan what you want on graph paper, and then determine how the constraints should be set. The alternative, adding the components to the layout and then tweaking the constraints until you have something you like, could lead to premature baldness. Seriously, a trial-and-error approach to getting the constraints right will certainly be frustrating and will probably fail. Figure 7.10, using the same `GridBagLayout` used in Figure 7.8, indicates how the layout manager counts cells. The partial code used to create the screen follows in Example 7.2.

Example 7.2: Creating a GridBagLayout

```
public void init() {
    Button b;
    GridBagLayout gb = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();
    setLayout(gb);
    try {
/* Row One - Three button */
        b = new Button ("One");
        addComponent (this, b, 0, 0, 1, 1,
                GridBagConstraints.NONE, GridBagConstraints.CENTER);
        b = new Button ("Two");
        addComponent (this, b, 1, 0, 1, 1,
                GridBagConstraints.NONE, GridBagConstraints.CENTER);
        b = new Button ("Three");
        addComponent (this, b, 2, 0, 1, 1,
                GridBagConstraints.NONE, GridBagConstraints.CENTER);
/* Row Two - Two buttons */
        b = new Button ("Four");
        addComponent (this, b, 0, 1, 2, 1,
                GridBagConstraints.NONE, GridBagConstraints.CENTER);
        b = new Button ("Five");
        addComponent (this, b, 2, 1, 1, 2,
                GridBagConstraints.NONE, GridBagConstraints.CENTER);
/* Row Three - Two buttons */
```

```
        b = new Button ("Six");
        addComponent (this, b, 0, 2, 1, 1,
                GridBagConstraints.NONE, GridBagConstraints.CENTER);
        b = new Button ("Seven");
        addComponent (this, b, 1, 2, 1, 1,
                GridBagConstraints.NONE, GridBagConstraints.CENTER);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Figure 7.10: How GridBagLayout counts rows and columns

[Graphic: Figure 7-10]

Most of the work in Example 7.2 is done by the helper method `addComponent()`, which creates a set of constraints, applies them to a component, and adds the component to a container. The code for `addComponent()` appears in GridBagConstraints; its signature is:

```
public static void addComponent (Container container, Component component,
        int gridx, int gridy, int gridwidth, int gridheight, int fill,
        int anchor) throws AWTException ;
```

The top left cell in the layout has location (`0,0`). There's nothing very surprising about buttons one, two, three, six, and seven. They occupy a 1x1 area on the layout's 3x3 grid. Button four occupies a 2x1 area; it is placed at location (0,1), and thus occupies this cell plus the cell at (`1,1`). Likewise, button five occupies a 1x2 area, and takes up the cells at (`2,1`) and (`2,2`). The total size of the layout is determined entirely by the components that are placed in it and their constraints.

GridBagLayout Methods

Variables

There are a handful of instance variables for `GridBagLayout`. They are not initialized until the container whose layout is `GridBagLayout` has been validated.

public int columnWidths[]

> The `columnWidths[]` array contains the widths of the components in the row with the most elements. The values of this array are returned by the `getLayoutDimensions()` method. You can access the array directly, but it is not recommended.

public int rowHeights[]

> The `rowHeights[]` array contains the heights of the components in the column with the most elements. The values of this array are returned by the `getLayoutDimensions()` method. You can access the array directly, but it is not recommended.

public double columnWeights[]

> The `columnWeights[]` array contains the `weightx` values of the components in the row with the most elements. The values of this array are returned by the `getLayoutWeights()` method. You can access the array directly, but it is not recommended.

public double rowWeights[]

> The `row Weights[]` array contains the `weighty` values of the components in the column with the most elements. The values of this array are returned by the `getLayoutWeights()` method. You can access the array directly, but it is not recommended.

Constructors

public GridBagLayout ()

> The constructor for `GridBagLayout` creates an instance of `GridBagLayout` with default `GridBagConstraints` behavior. An internal table is used to keep track of the components added to the layout.

LayoutManager methods

public void addLayoutComponent (String name, Component component)

> The `addLayoutComponent()` method of `GridBagLayout` does nothing. This method is not deprecated, unlike the similarly named methods in the other layout managers that implement `LayoutManager2`.

public void removeLayoutComponent (Component component)

The `removeLayoutComponent()` method of `GridBagLayout` does nothing.

public Dimension preferredLayoutSize (Container target)

The `preferredLayoutSize()` method calculates the preferred dimensions of the components of `target`. Sizing is based on the constraints of the various components. This task is definitely better off left to the computer.

public Dimension minimumLayoutSize (Container target)

The `minimumLayoutSize()` method calculates the minimum dimensions required to position the components of `target`. Sizing is based on the constraints of the various components.

public void layoutContainer (Container target)

The `layoutContainer()` method positions the components within `target` based upon the constraints of each component. If a component's anchor constraints are invalid, `layoutContainer()` throws the run-time exception `IllegalArgumentException`. The process of arranging the components is very complicated and beyond the scope of this book.

LayoutManager2 methods

public void addLayoutComponent (Component component, Object constraints) ★

This `addLayoutComponent()` method of `GridBagLayout` associates the `component` with the given `constraints` object. It calls the `setConstaints()` method.

If `name` is not a `GridBagConstraints`, `addLayoutComponent()` throws the run-time exception `IllegalArgumentException`.

public abstract Dimension maximumLayoutSize(Container target) ★

The `maximumLayoutSize()` method returns a `Dimension` object with a width and height of `Integer.MAX_VALUE`. In practice, this means that `GridBagLayout` doesn't support the concept of maximum size.

public abstract float getLayoutAlignmentX(Container target) ★

The `getLayoutAlignmentX()` method says that `GridBagLayout` containers should be centered horizontally within the area available.

public abstract float getLayoutAlignmentY(Container target) ★

The `getLayoutAlignmentY()` method says that `GridBagLayout` containers should be centered vertically within the area available.

public abstract void invalidateLayout(Container target) ★

The `invalidateLayout()` method of `GridBagLayout` does nothing.

Constraints

public GridBagConstraints getConstraints (Component component)

The `getConstraints()` method returns a clone of the current constraints for `component`. This makes it easier to generate constraints for a component based on another component.

public void setConstraints (Component component, GridBagConstraints constraints)

The `setConstraints()` method changes the `constraints` on `component` to a clone of `constraints`. The system creates a `clone()` of `constraints` so you can change the original constraints without affecting `component`.

Layout

public Point getLayoutOrigin ()

The `getLayoutOrigin()` method returns the origin for the `GridBagLayout`. The origin is the top left point within the container at which the components are drawn. Before the container is validated, `getLayoutOrigin()` returns the `Point` (`0,0`). After validation, `getLayoutOrigin()` returns the actual origin of the layout. The space used by the components within a `GridBagLayout` may not fill the entire container. You can use the results of `getLayoutOrigin()` and `getLayoutDimensions()` to find the layout's actual size and draw a `Rectangle` around the objects.

public int[][] getLayoutDimensions ()

The `getLayoutDimensions()` method returns two one-dimensional arrays as a single two-dimensional array. Index 0 is an array of widths (`columnWidths` instance variable), while index 1 is an array of heights (`rowHeights` instance variable). Until the layout is validated, these will be empty. After validation, the first array contains the widths of the components in the row with the most elements. The second contains the heights of the components in the column with the most elements. For Figure 7.10, the results would be (38, 51, 48) for widths since the first row has three elements and (21, 21, 21) for the heights since the first (and second) column has three elements in it.

public double[][] getLayoutWeights ()

The `getLayoutWeights()` method returns two one-dimensional arrays as a single two-dimensional array. Index 0 is an array of column weights (`columnWeights` instance variable), while index 1 is an array of row weights (`rowWeights` instance variable). Until the layout is validated, these will be empty. After validation, the first dimension contains all the `weightx` values of the components in the row with the most elements. The second dimension contains all the `weighty` values of the components in the column with the most elements. For Figure 7.10, the results would be (0, 0, 0) for `weightx` since the first row has three elements and (0, 0, 0) for `weighty` since the first column has three elements in it.

Miscellaneous methods

public Point location (int x, int y)

The `location()` method returns the `Point` (`0,0`) until the container is validated. After validation, this method returns the grid element under the location (`x`, `y`), where `x` and `y` are in pixels. The results could be used as the `gridx` and `gridy` constraints when adding another component.

public String toString ()

The `toString()` method of `GridBagLayout` returns the name of the class:

```
java.awt.GridBagLayout
```

PREVIOUS
CardLayout

HOME
BOOK INDEX

NEXT
GridBagConstraints

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

7.7 GridBagConstraints

GridBagConstraints are the meat behind the GridBagLayout; they specify how to display components. Unlike other layout managers, which have a built-in idea about what to do with their display, the GridBagLayout is a blank slate. The constraints attached to each component tell the layout manager how to build its display.

Every Component added to a GridBagLayout has a GridBagConstraints object associated with it. When an object is first added to the layout, it is given a default set of constraints (described later in this section). Calling setConstraints() (or add(Component, GridBagConstraints)) applies a new set of constraints to the object. Most people create a helper method to make the setConstraints() calls, passing constraint information as parameters. The helper method used in Example 7.2 follows:

```
public static void addComponent (Container container, Component component,
    int gridx, int gridy, int gridwidth, int gridheight, int fill,
    int anchor) throws AWTException {
    LayoutManager lm = container.getLayout();
    if (!(lm instanceof GridBagLayout)) {
        throw new AWTException ("Invalid layout" + lm);
    } else {
        GridBagConstraints gbc = new GridBagConstraints ();
        gbc.gridx = gridx;
        gbc.gridy = gridy;
        gbc.gridwidth = gridwidth;
        gbc.gridheight = gridheight;
        gbc.fill = fill;
        gbc.anchor = anchor;
        ((GridBagLayout)lm).setConstraints(component, gbc);
        container.add (component);
    }
}
```

In Java 1.1, you can make this method slightly cleaner by adding the component and applying the constraints in the same call to add(). To do so, replace the lines calling setConstraints() and add() with this line:

```
        // Java 1.1 only
        container.add(component, gbc);
```

GridBagConstraints Methods

Constants and variables

public int anchor

> The `anchor` specifies the direction in which the component will drift in the event that it is smaller than the space available for it. `CENTER` is the default. Others available are `NORTH`, `SOUTH`, `EAST`, `WEST`, `NORTHEAST`, `NORTHWEST`, `SOUTHEAST`, and `SOUTHWEST`.

public final static int CENTER
public final static int EAST
public final static int NORTH
public final static int NORTHEAST
public final static int NORTHWEST
public final static int SOUTH
public final static int SOUTHEAST
public final static int SOUTHWEST
public final static int WEST

> Constants used to set the `anchor`.

public int fill

> The value of `fill` controls the component's resize policy. If `fill` is `NONE` (the default), the layout manager tries to give the component its preferred size. If `fill` is `VERTICAL`, it resizes in height if additional space is available. If fill is `HORIZONTAL`, it resizes in width. If fill is `BOTH`, the layout manager takes advantage of all the space available in either direction. Figure 7.11 demonstrates `VERTICAL` (A), `HORIZONTAL` (B), and `NONE` (C) values; Figure 7.8 demonstrated the use of `BOTH`.

public final static int NONE
public final static int BOTH
public final static int HORIZONTAL
public final static int VERTICAL

> Constants used to set `fill`.

Figure 7.11: GridBagLayout with fill values of VERTICAL, HORIZONTAL, and NONE

[Graphic: Figure 7-11]

public int gridx
public int gridy

The `gridx` and `gridy` variables specify the grid position where this component will be placed. (`0,0`) specifies the cell at the origin of the screen. Table 7.2 shows the `gridx` and `gridy` values for the screen in Figure 7.8.

It isn't necessary to set `gridx` and `gridy` to a specific location; if you set these fields to `RELATIVE` (the default), the system calculates the location for you. According to the comments in the source code, if `gridx` is `RELATIVE`, the component appears to the right of the last component added to the layout. If `gridy` is `RELATIVE`, the component appears below the last component added to the layout. However, this is misleadingly simple. `RELATIVE` placement works best if you are adding components along a row or a column. In this case, there are four possibilities to consider:

- `gridx` and `gridy` `RELATIVE`: components are placed in one row.
- `gridx` `RELATIVE`, `gridy` constant: components are placed in one row, each to the right of the previous component.
- `gridx` constant, `gridy` `RELATIVE`: components are placed in one column, each below the previous component.
- Varying `gridx` or `gridy` while setting the other field to `RELATIVE` appears to start a new row, placing the component as the first element in the row.

public int gridwidth
public int gridheight

`gridwidth` and `gridheight` set the number of rows (`gridwidth`) and columns (`gridheight`) a particular component occupies. If `gridwidth` or `gridheight` is set to `REMAINDER`, the component will be the last element of the row or column occupying any space that's remaining. Table 7.2 shows the `gridwidth` and `gridheight` values for the screen in Figure 7.8. For the components in the last column, the `gridwidth` values could be `REMAINDER`. Likewise, `gridheight` could be set to `REMAINDER` for the components in the last row.

gridwidth and gridheight may also have the value RELATIVE, which forces the component to be the next to last component in the row or column. Looking back to Figure 7.8: if button six has a gridwidth of RELATIVE, button seven won't appear because button five is the last item in the row, and six is already next to last. If button five has a gridheight of RELATIVE, the layout manager will reserve space below it, so the button can be the next to last item in the column.

public final static int RELATIVE

Constant used for gridx and gridy to request relative placement, and by gridheight and gridwidth to specify the next to last component in a column or row. The behavior of RELATIVE placement can be very counter intuitive; in most cases, you will be better off specifying gridx, gridy, gridheight, and gridwidth explicitly.

public final static int REMAINDER

Constant used for gridwidth and gridheight, to specify that a component should fill the rest of the row or column.

Table 7.2: Demonstrating gridx/gridy/gridwidth/gridheight

Component	**gridx**	**gridy**	**gridwidth**	**gridheight**
One	0	0	1	1
Two	1	0	1	1
Three	2	0	1	1
Four	0	1	2	1
Five	2	1	1	2
Six	0	2	1	1
Seven	1	2	1	3

public Insets insets

The insets field specifies the external padding in pixels around the component (i.e., between the component and the edge of the cell, or cells, allotted to it). An Insets object can specify different padding for the top, bottom, left, and right sides of the component.

public int ipadx
public int ipady

ipadx and ipady specify the internal padding within the component. ipadx specifies the extra space to the right and left of the component (so the minimum width increases by `2*ipadx` pixels). `ipady` specifies the extra space above and below the component (so the minimum height increases by `2*ipady` pixels).

The difference between insets (external padding) and the `ipadx`, `ipady` variables (internal padding) is confusing. The insets don't add space to the component itself; they are external to the component. `ipadx` and `ipady` change the component's minimum size, so they do add space to the component itself.

public double weightx
public double weighty

The `weightx` and `weighty` variables describe how to distribute any additional space within the container. They allow you to control how components grow (or shrink) when the user resizes the container. If `weightx` is 0, the component won't get any additional space available in its row. If one or more components in a row have `weightx` values greater than 0, any extra space is distributed proportionally between them. For example, if one component has a `weightx` value of 1 and the others are all 0, that one component will get all the additional space. If four components in a row each have `weightx` values of 1 and the other components have `weightx` values of 0, the four components each get one quarter of the additional space. `weighty` behaves similarly. Because `weightx` and `weighty` control the distribution of extra space in any row or column, setting either for one component may affect the position of other components.

Constructors

public GridBagConstraints ()

The constructor creates a `GridBagConstraints` object in which all the fields have their default values. These defaults are shown in the Table 7.3.

Table 7.3: GridBagConstraints Defaults.

Variable	**Value**	**Description**
`anchor`	`CENTER`	If the component is smaller than the space available, it will be centered within its region.
`fill`	`NONE`	The component should not resize itself if extra space is available within its region.
`gridx`	`RELATIVE`	The component associated with this constraint will be positioned relative to the last item added. If all components have `gridx` and `gridy` `RELATIVE`, they will be placed in a single row.
`gridy`	`RELATIVE`	The component associated with this constraint will be positioned relative to the last item added.
`gridwidth`	`1`	The component will occupy a single cell within the layout.

`gridheight`	`1`	The component will occupy a single cell within the layout.
`insets`	`0x0x0x0`	No extra space is added around the edges of the component.
`ipadx`	`0`	There is no internal padding for the component.
`ipady`	`0`	There is no internal padding for the component.
`weightx`	`0`	The component will not get any extra space, if it is available.
`weighty`	`0`	The component will not get any extra space, if it is available.

Miscellaneous methods

public Object clone ()

> The `clone()` method creates a clone of the `GridBagConstraints` so the same `GridBagConstraints` object can be associated with multiple components.

PREVIOUS HOME NEXT
GridBagLayout BOOK INDEX Combining Layouts

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

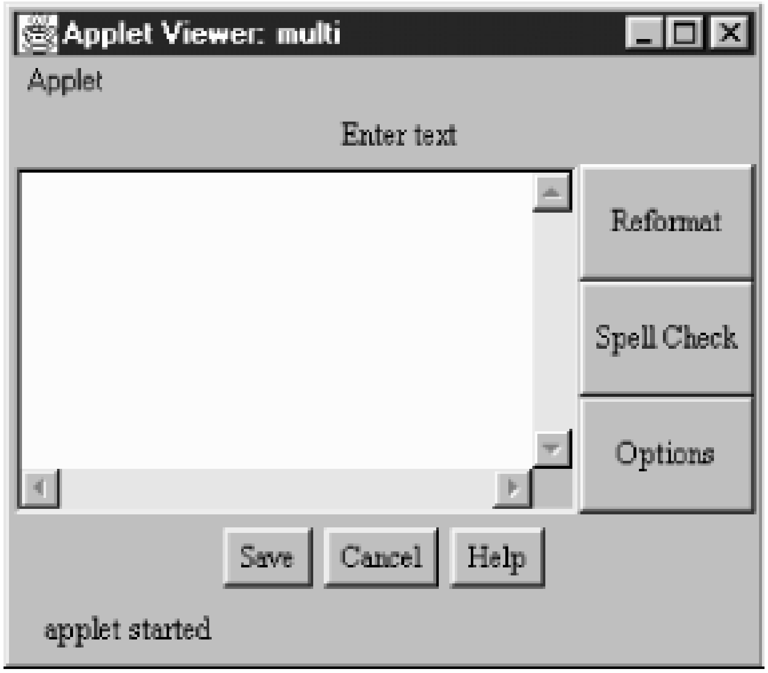
7.8 Combining Layouts

If you can't create the display you want with any of the standard layout managers, or you are unable to figure out `GridBagLayout`, you may want to try combining several different layouts. This technique can often help you build the display you want. Figure 7.12 shows a display that uses three panels and three different layouts.

Here's the source code to generate the display in Figure 7.12:

```
import java.awt.*;
public class multi extends java.applet.Applet {
    public void init() {
        Panel s = new Panel();
        Panel e = new Panel();
        setLayout (new BorderLayout ());
        add ("North", new Label ("Enter text", Label.CENTER));
        add ("Center", new TextArea ());
        e.setLayout (new GridLayout (0,1));
        e.add (new Button ("Reformat"));
        e.add (new Button ("Spell Check"));
        e.add (new Button ("Options"));
        add ("East", e);
        s.setLayout (new FlowLayout ());
        s.add (new Button ("Save"));
        s.add (new Button ("Cancel"));
        s.add (new Button ("Help"));
        add ("South", s);
    }
}
```

Figure 7.12: Multipanel screen using several layouts



The display in Figure 7.12 is created by adding four sections to a single `BorderLayout`. The north region contains a panel with a single `Label` in it. The panel uses its default `LayoutManager`, which is a `FlowLayout`. Why bother with this panel? Why not just add a label at the north position in the `BorderLayout`? Our strategy gives the label the position and size we want: the label is centered and displayed at its preferred size. If we had added the label directly to the `BorderLayout`, it would have been left justified and resized to fill the region.

The `TextArea` has no special requirements, so we added it directly to the center of the `BorderLayout`.

The three buttons on the right of the screen were arranged in a panel with a `GridLayout`; then this panel was placed in the east region of the `BorderLayout`.

To create the buttons at the bottom of the screen, we used another `Panel` with a `FlowLayout`. It centers the three buttons and displays them at their preferred size, with a gap between them.

With a little work, we could have created this display using a single `Panel` with a `GridBagLayout`. The result would have been more efficient; placing panels within panels has performance implications. Each container in the display has its own peer object, which uses up system resources. Furthermore, in the 1.0 version of AWT, nesting containers complicates event handling. However, using a `GridBagLayout` would have required much more work: figuring out the right `GridBagConstraints` for each component would be time consuming and result in code that is harder to understand. Sometimes, it's best to settle for the easy solution: a hybrid layout composed of several simple panels, rather than a single very complex panel.

In Java 1.1, you can make this program even more efficient in its resource usage by using a lightweight component instead of panels. This is particularly easy because the panels in the multipanel screen exist strictly to help with layout and not for partitioning event handling. Therefore, you can define a `LightweightPanel` that extends `Container`, with no methods. Use this class instead of `Panel`. The `LightweightPanel` allows you to lay out areas without creating unnecessary peers. Here's all the code for the `LightweightPanel`:

```
// Java 1.1 only
import java.awt.*;
public class LightweightPanel extends Container {
}
```

7.9 Disabling the LayoutManager

To create a container with no layout manager, use `null` as the argument to `setLayout()`. If you do this, you must size and position every component individually. In most cases, disabling the `LayoutManager` is a bad idea because what might look great on one platform could look really bad on another, due to differences in fonts, native components, and other display characteristics. Figure 7.13 displays a container with a disabled `LayoutManager`; both buttons were positioned by specifying their size and location explicitly.

Figure 7.13: Applet with disabled layout manager

[Graphic: Figure 7-13]

Here's the code that produces Figure 7.13:

```
import java.awt.Button;
import java.applet.Applet;
```

```
public class noLayout extends Applet {
    public void init () {
        setLayout (null);
        Button x = new Button ("Hello");
        add (x);
        x.reshape (50, 60, 50, 70);
        Button y = new Button ("World");
        add (y);
        y.reshape (100, 120, 50, 70);
    }
}
```

PREVIOUS HOME NEXT
Combining Layouts BOOK INDEX Designing Your Own LayoutManager

7.10 Designing Your Own LayoutManager

What if you can't find a `LayoutManager` that fits your requirements, or you find yourself repeatedly building the same multipanel display? In cases like these, you can build your own layout manager. It's really not that difficult; you only need to implement the five methods of the `LayoutManager` interface, plus a constructor and any additional methods your design requires. In this section, we'll review the `LayoutManager` interface and then construct a custom `LayoutManager` called `CornerLayout`.

LayoutManager Methods

A custom `LayoutManager` must implement the following five methods (ten methods if you implement `LayoutManager2`). For many layout managers, several of these methods can be stubs that don't do anything.

public void addLayoutComponent (String name, Component component)

> The `addLayoutComponent()` method is called by the `add(name, component)` method of `Container`. If your new `LayoutManager` does not have named component areas or does not pass generic positioning information via `name`, this method will be a stub with no code; you can let the container keep track of the components for you. Otherwise, this method must keep track of the component added, along with the information in name.
>
> How would you implement this method? For layouts that have named component areas (like `BorderLayout`), you could use a private instance variable to hold the component for each area. For layouts like `CardLayout`, which lets you refer to individual components by name, you might want to store the components and their names in an internal `Hashtable`.

public void removeLayoutComponent (Component component)

> This method is called by the `remove()` and `removeAll()` methods of `Container`. If you are storing information in internal instance variables or tables, you can remove the information about the given `Component` from the tables at this point. If you're not keeping track of the components yourself, this method can be a stub that does nothing.

public Dimension preferredLayoutSize (Container target)

> This method is called by `preferredSize()` to calculate the desired size of `target`.[1] Obviously, the preferred size of the container depends on the layout strategy that you implement. To compute the preferred size,

you usually need to call the `preferredSize()` method of every component in the container.

> [1] This is still true in Java 1.1; the new method, `getPreferredSize()`, just calls the deprecated method, `preferredSize()`.

Computing the preferred size can be messy. However, some layout strategies let you take a shortcut. If your layout policy is "I'm going to cram all the components into the space given to me, whether they fit or not," you can compute the preferred size of your layout simply by calling `target.size()` or (in Java 1.1) `target.getSize()`.

public Dimension minimumLayoutSize (Container target)

This method is called by `minimumSize()` to calculate the minimum size of `target`. The minimum size of the container depends on the layout strategy that you implement. To compute the minimum size, you usually need to call the `minimumSize()` method of every component in the container.

As with `preferredLayoutSize()`, you can sometimes save a lot of work by returning `target.size()`.

public void layoutContainer (Container target)

This method is called when target is first displayed and whenever it is resized. It is responsible for arranging the components within the container. Depending upon the type of `LayoutManager` you are creating, you will either loop through all the components in the container with the `getComponent()` method or use the named components that you saved in the `addLayoutComponent()` method. To position and size the components, call their `reshape()` or `setBounds()` methods.

A New LayoutManager: CornerLayout

`CornerLayout` is a simple but useful layout manager that is similar in many respects to `BorderLayout`. Like `BorderLayout`, it positions components in five named regions: "Northeast", "Northwest", "Southeast", "Southwest", and "Center". These regions correspond to the four corners of the container, plus the center. The "Center" region has three modes. `NORMAL`, the default mode, places the "Center" component in the center of the container, with its corners at the inner corner of the other four regions. `FULL_WIDTH` lets the center region occupy the full width of the container. `FULL_HEIGHT` lets the center region occupy the full height of the container. You cannot specify both `FULL_HEIGHT` and `FULL_WIDTH`; if you did, the "Center" component would overlap the corner components and take over the container. Figure 7.14 shows a `CornerLayout` in each of these modes.

Not all regions are required. If a complete side is missing, the required space for the container decreases. Ordinarily, the other components would grow to fill this vacated space. However, if the container is sized to its preferred size, so are the components. Figure 7.15 shows this behavior.

Figure 7.14: CornerLayout

[Graphic: Figure 7-14]

Figure 7.15: CornerLayout with missing regions

[Graphic: Figure 7-15]

Example 7.3 is the code for the `CornerLayout`. It shows the Java 1.0 version of the layout manager. At the end of this section, I show the simple change needed to adapt this manager to Java 1.1.

Example 7.3: The CornerLayout LayoutManager

```
import java.awt.*;
/**
 * An 'educational' layout. CornerLayout will layout a container
 * using members named "Northeast", "Northwest", "Southeast",
 * "Southwest", and "Center".
 *
 * The "Northeast", "Northwest", "Southeast" and "Southwest" components
 * get sized relative to the adjacent corner's components and
 * the constraints of the container's size. The "Center" component will
 * get any space left over.
 */
public class CornerLayout implements LayoutManager {
    int hgap;
    int vgap;
    int mode;
    public final static int NORMAL = 0;
    public final static int FULL_WIDTH = 1;
    public final static int FULL_HEIGHT = 2;
    Component northwest;
    Component southwest;
    Component northeast;
    Component southeast;
    Component center;
```

The CornerLayout class starts by defining instance variables to hold the gaps and mode and the components for each corner of the screen. It also defines three constants that control the behavior of the center region: NORMAL, FULL_WIDTH, and FULL_HEIGHT.

```
    /**
     * Constructs a new CornerLayout.
     */
    public CornerLayout() {
        this (0, 0, CornerLayout.NORMAL);
    }
    public CornerLayout(int mode) {
        this (0, 0, mode);
    }
    public CornerLayout(int hgap, int vgap) {
        this (hgap, vgap, CornerLayout.NORMAL);
    }
    public CornerLayout(int hgap, int vgap, int mode) {
        this.hgap = hgap;
        this.vgap = vgap;
        this.mode = mode;
    }
```

The constructors for CornerLayout are simple. The default (no arguments) constructor creates a CornerLayout with no gaps; the "Center" region is NORMAL mode. The last constructor, which is called by the other three, stores the gaps and the mode in instance variables.

```
    public void addLayoutComponent (String name, Component comp) {
        if ("Center".equals(name)) {
            center = comp;
        } else if ("Northwest".equals(name)) {
            northwest = comp;
        } else if ("Southeast".equals(name)) {
            southeast = comp;
        } else if ("Northeast".equals(name)) {
            northeast = comp;
        } else if ("Southwest".equals(name)) {
            southwest = comp;
        }
    }
```

addLayoutComponent() figures out which region a component has been assigned to, and saves the component in the corresponding instance variable. If the name of the component isn't "Northeast", "Northwest", Southeast", "Southwest", or "Center", the component is ignored.

```
    public void removeLayoutComponent (Component comp) {
        if (comp == center) {
            center = null;
        } else if (comp == northwest) {
            northwest = null;
        } else if (comp == southeast) {
            southeast = null;
        } else if (comp == northeast) {
            northeast = null;
        } else if (comp == southwest) {
            southwest = null;
        }
    }
```

`removeLayoutComponent()` searches for a given component in each region; if it finds the component, `removeLayoutComponent()` discards it by setting the instance variable to null.

```
    public Dimension minimumLayoutSize (Container target) {
        Dimension dim = new Dimension(0, 0);
        Dimension northeastDim = new Dimension (0,0);
        Dimension northwestDim = new Dimension (0,0);
        Dimension southeastDim = new Dimension (0,0);
        Dimension southwestDim = new Dimension (0,0);
        Dimension centerDim    = new Dimension (0,0);
        if ((northeast != null) && northeast.isVisible ()) {
            northeastDim = northeast.minimumSize ();
        }
        if ((southwest != null) && southwest.isVisible ()) {
            southwestDim = southwest.minimumSize ();
        }
        if ((center != null) && center.isVisible ()) {
            centerDim = center.minimumSize ();
        }
        if ((northwest != null) && northwest.isVisible ()) {
            northwestDim = northwest.minimumSize ();
        }
        if ((southeast != null) && southeast.isVisible ()) {
            southeastDim = southeast.minimumSize ();
        }
        dim.width = Math.max (northwestDim.width, southwestDim.width) +
                        hgap + centerDim.width + hgap +
                        Math.max (northeastDim.width, southeastDim.width);
        dim.height = Math.max (northwestDim.height, northeastDim.height) +
                        + vgap + centerDim.height + vgap +
                        Math.max (southeastDim.height, southwestDim.height);
        Insets insets = target.insets();
        dim.width += insets.left + insets.right;
        dim.height += insets.top + insets.bottom;
        return dim;
    }
```

`minimumLayoutSize()` computes the minimum size of the layout by finding the minimum sizes of all components. To compute the minimum width, `minimumLayoutSize()` adds the width of the center, plus the greater of the widths of the western regions (northwest and southwest), plus the greater of the widths of the eastern regions (northeast and southeast), then adds the appropriate gaps and insets. The minimum height is computed similarly; the method takes the greater of the minimum heights of the northern regions, the greater of the minimum heights of the southern regions, and adds them to the minimum height of the center region, together with the appropriate gaps and insets.

```
    public Dimension preferredLayoutSize (Container target) {
        Dimension dim = new Dimension(0, 0);
        Dimension northeastDim = new Dimension (0,0);
        Dimension northwestDim = new Dimension (0,0);
        Dimension southeastDim = new Dimension (0,0);
        Dimension southwestDim = new Dimension (0,0);
        Dimension centerDim    = new Dimension (0,0);
        if ((northeast != null) && northeast.isVisible ()) {
            northeastDim = northeast.preferredSize ();
        }
        if ((southwest != null) && southwest.isVisible ()) {
            southwestDim = southwest.preferredSize ();
        }
        if ((center != null) && center.isVisible ()) {
            centerDim = center.preferredSize ();
        }
        if ((northwest != null) && northwest.isVisible ()) {
            northwestDim = northwest.preferredSize ();
        }
        if ((southeast != null) && southeast.isVisible ()) {
            southeastDim = southeast.preferredSize ();
        }
        dim.width = Math.max (northwestDim.width, southwestDim.width) +
                        hgap + centerDim.width + hgap +
                        Math.max (northeastDim.width, southeastDim.width);
        dim.height = Math.max (northwestDim.height, northeastDim.height) +
                        + vgap + centerDim.height + vgap +
                        Math.max (southeastDim.height, southwestDim.height);
        Insets insets = target.insets();
        dim.width += insets.left + insets.right;
        dim.height += insets.top + insets.bottom;
        return dim;
    }
```

`preferredLayoutSize()` computes the preferred size of the layout. The method is almost identical to `minimumLayoutSize()`, except that it uses the preferred dimensions of each component.

```
    public void layoutContainer (Container target) {
        Insets insets = target.insets();
        int top = insets.top;
        int bottom = target.size ().height - insets.bottom;
        int left = insets.left;
        int right = target.size ().width - insets.right;
        Dimension northeastDim = new Dimension (0,0);
        Dimension northwestDim = new Dimension (0,0);
        Dimension southeastDim = new Dimension (0,0);
        Dimension southwestDim = new Dimension (0,0);
        Dimension centerDim    = new Dimension (0,0);
        Point topLeftCorner, topRightCorner, bottomLeftCorner,
                        bottomRightCorner;
        if ((northeast != null) && northeast.isVisible ()) {
            northeastDim = northeast.preferredSize ();
        }
        if ((southwest != null) && southwest.isVisible ()) {
            southwestDim = southwest.preferredSize ();
        }
        if ((center != null) && center.isVisible ()) {
            centerDim = center.preferredSize ();
        }
        if ((northwest != null) && northwest.isVisible ()) {
            northwestDim = northwest.preferredSize ();
        }
        if ((southeast != null) && southeast.isVisible ()) {
            southeastDim = southeast.preferredSize ();
        }
        topLeftCorner = new Point (left +
                          Math.max (northwestDim.width, southwestDim.width),
                          top +
                          Math.max (northwestDim.height, northeastDim.height));
        topRightCorner = new Point (right -
                          Math.max (northeastDim.width, southeastDim.width),
                          top +
                          Math.max (northwestDim.height, northeastDim.height));
        bottomLeftCorner = new Point (left +
                          Math.max (northwestDim.width, southwestDim.width),
                          bottom -
                          Math.max (southwestDim.height, southeastDim.height));
        bottomRightCorner = new Point (right  -
                          Math.max (northeastDim.width, southeastDim.width),
                          bottom -
                          Math.max (southwestDim.height, southeastDim.height));
        if ((northwest != null) && northwest.isVisible ()) {
            northwest.reshape (left, top,
                                left + topLeftCorner.x,
                                top + topLeftCorner.y);
        }
        if ((southwest != null) && southwest.isVisible ()) {
            southwest.reshape (left, bottomLeftCorner.y,
                                bottomLeftCorner.x - left,
                                bottom - bottomLeftCorner.y);
       }
```

```
        if ((southeast != null) && southeast.isVisible ()) {
            southeast.reshape (bottomRightCorner.x,
                        bottomRightCorner.y,
                        right - bottomRightCorner.x,
                        bottom - bottomRightCorner.y);
        }
        if ((northeast != null) && northeast.isVisible ()) {
            northeast.reshape (topRightCorner.x, top,
                                right - topRightCorner.x,
                                topRightCorner.y);
        }
        if ((center != null) && center.isVisible ()) {
            int x = topLeftCorner.x + hgap;
            int y = topLeftCorner.y + vgap;
            int width = bottomRightCorner.x - topLeftCorner.x - hgap * 2;
            int height = bottomRightCorner.y - topLeftCorner.y - vgap * 2;
            if (mode == CornerLayout.FULL_WIDTH) {
                x = left;
                width = right - left;
            } else if (mode == CornerLayout.FULL_HEIGHT) {
                y = top;
                height = bottom - top;
            }
            center.reshape (x, y, width, height);
        }
    }
```

`layoutContainer()` does the real work: it positions and sizes the components in our layout. It starts by computing the region of the target container that we have to work with, which is essentially the size of the container minus the insets. The boundaries of the working area are stored in the variables `top`, `bottom`, `left`, and `right`. Next, we get the preferred sizes of all visible components and use them to compute the corners of the "Center" region; these are stored in the variables `topLeftCorner`, `topRightCorner`, `bottomLeftCorner`, and `bottomRightCorner`.

Once we've computed the location of the "Center" region, we can start placing the components in their respective corners. To do so, we simply check whether the component is visible; if it is, we call its `reshape()` method. After dealing with the corner components, we place the "Center" component, taking into account any gaps (`hgap` and `vgap`) and the layout's mode. If the mode is `NORMAL`, the center component occupies the region between the inner corners of the other components. If the mode is `FULL_HEIGHT`, it occupies the full height of the screen. If it is `FULL_WIDTH`, it occupies the full width of the screen.

```
    public String toString() {
        Sting str;
        switch (mode) {
            case FULL_HEIGHT: str = "tall"; break;
            case FULL_WIDTH: str = "wide"; break;
            default: str = "normal"; break;
        }
        return getClass().getName () + "[hgap=" + hgap + ",vgap=" + vgap +
            ",mode="+str+"]";
    }
```

```
}
```

`toString()` simply returns a string describing the layout.

Strictly speaking, there's no reason to update the `CornerLayout` for Java 1.1. Nothing about Java 1.1 says that new layout managers have to implement the `LayoutManager2` interface. However, implementing `LayoutManager2` isn't a bad idea, particularly since `CornerLayout` works with constraints; like `BorderLayout`, it has named regions. To extend `CornerLayout` so that it implements `LayoutManager2`, add the following code; we'll create a new `CornerLayout2`:

```
// Java 1.1 only
import java.awt.*;
public class CornerLayout2 extends CornerLayout implements LayoutManager2 {
    public void addLayoutComponent(Component comp, Object constraints) {
        if ((constraints == null) || (constraints instanceof String)) {
            addLayoutComponent((String)constraints, comp);
        } else {
            throw new IllegalArgumentException(
                  "cannot add to layout: constraint must be a string (or null)");
        }
    }
    public Dimension maximumLayoutSize(Container target) {
        return new Dimension(Integer.MAX_VALUE, Integer.MAX_VALUE);
    }
    public float getLayoutAlignmentX(Container parent) {
        return Component.CENTER_ALIGNMENT;
    }
    public float getLayoutAlignmentY(Container parent) {
        return Component.CENTER_ALIGNMENT;
    }
    public void invalidateLayout(Container target) {
    }
}
```

PREVIOUS
Disabling the LayoutManager

HOME
BOOK INDEX

NEXT
The sun.awt Layout Collection

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

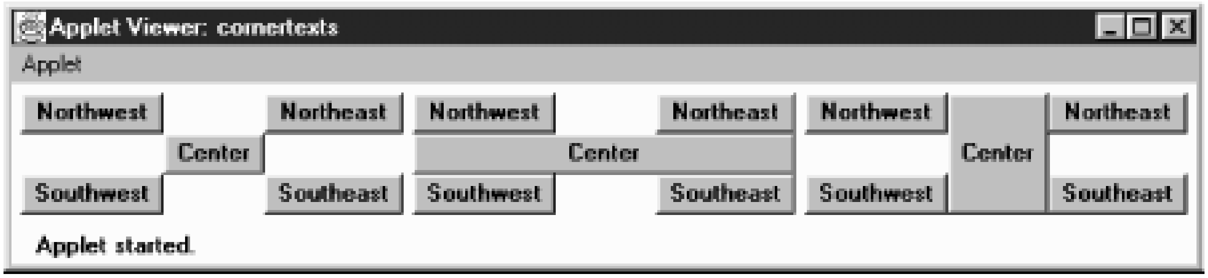
7.11 The sun.awt Layout Collection

The `sun.awt` package defines four additional layouts. The first two, `HorizBagLayout` and `VerticalBagLayout`, are available only when used with Sun's JDK or Internet Explorer, since they are not provided with Netscape Navigator and may not be available from other vendors. Therefore, these layout managers should be used selectively within applets. The third layout manager, `VariableGridLayout`, is available with Netscape Navigator 2.0 or 3.0 and Internet Explorer. Usage of this layout manager is safer within applets but is still at your own risk. The final layout manager is introduced in Java 1.1, `OrientableFlowLayout`. Only time will tell where that one will be available. Any of these layout managers could be moved into a future version of `java.awt` if there is enough interest.

HorizBagLayout

In a `HorizBagLayout`, the components are all arranged in a single row, from left to right. The height of each component is the height of the container; the width of each component is its preferred width. Figure 7.16 shows `HorizBagLayout` in use.

Figure 7.16: HorizBagLayout



Applet started.

Constructors

public HorizBagLayout ()

This constructor creates a `HorizBagLayout` with a horizontal gap of zero pixels. The gap is the space between the different components in the horizontal direction.

public HorizBagLayout (int hgap)

This constructor creates a `HorizBagLayout` using a horizontal gap of `hgap` pixels.

LayoutManager methods

public void addLayoutComponent (String name, Component component)

The `addLayoutComponent()` method of `HorizBagLayout` does nothing.

public void removeLayoutComponent (Component component)

The `removeLayoutComponent()` method of `HorizBagLayout` does nothing.

public Dimension preferredLayoutSize (Container target)

The `preferredLayoutSize()` method of `HorizBagLayout` sums up the preferred widths of all the components in `target`, along with the `hgap` and right and left insets to get the width of the `target`. The height returned will be the preferred height of the tallest component.

public Dimension minimumLayoutSize (Container target)

The `minimumLayoutSize()` method of `HorizBagLayout` sums up the minimum widths of all the components in `target`, along with the `hgap` and right and left insets to get the width of the `target`. The height returned will be the minimum height of the tallest component.

public void layoutContainer (Container target)

The `layoutContainer()` method draws `target`'s components on the screen in one row. The height of each component is the height of the container. Each component's width is its preferred width, if enough space is available.

Miscellaneous methods

public String toString ()

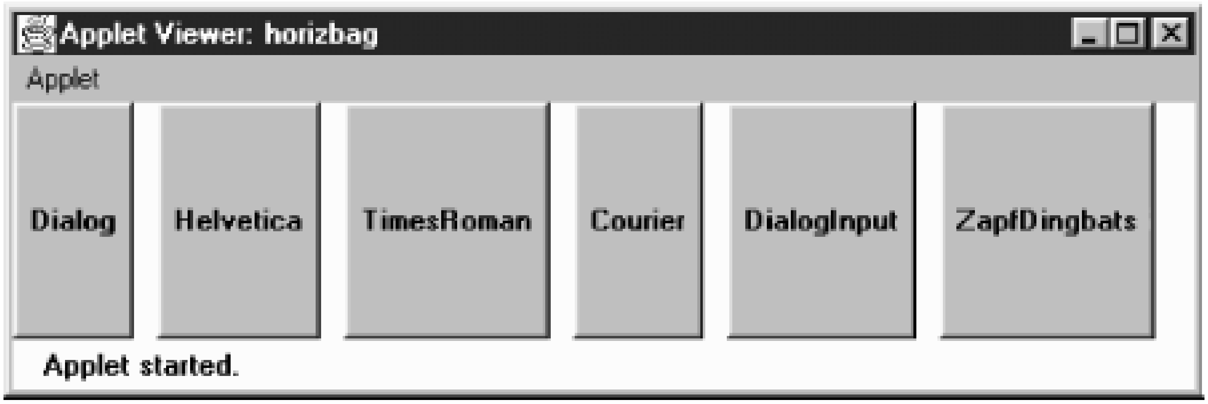
The `toString()` method of `HorizBagLayout` returns a string with the current horizontal gap setting--for example:

```
sun.awt.HorizBagLayout[hgap=0]
```

VerticalBagLayout

The `VerticalBagLayout` places all the components in a single column. The width of each component is the width of the container; each component is given its preferred height. Figure 7.17 shows `VerticalBagLayout` in use.

Figure 7.17: VerticalBagLayout



Constructors

public VerticalBagLayout ()

> This constructor creates a `VerticalBagLayout` with a vertical gap of zero pixels. The gap is the space between components in the vertical direction. With a gap of 0, adjacent components will touch each other.

public VerticalBagLayout (int vgap)

> This constructor creates a `VerticalBagLayout` with a vertical gap of `vgap` pixels.

LayoutManager methods

public void addLayoutComponent (String name, Component component)

> The `addLayoutComponent()` method of `VerticalBagLayout` does nothing.

public void removeLayoutComponent (Component component)

The `removeLayoutComponent()` method of `VerticalBagLayout` does nothing.

public Dimension preferredLayoutSize (Container target)

To get the preferred height of the layout, the `preferredLayoutSize()` method sums up the preferred height of all the components in `target` along with the `vgap` and top and bottom insets. For the preferred width, `preferredLayoutSize()` returns the preferred width of the widest component.

public Dimension minimumLayoutSize (Container target)

To get the minimum height of the layout, the `minimumLayoutSize()` method sums up the minimum height of all the components in `target` along with the `vgap` and top and bottom insets. For the minimum width, `minimumLayoutSize()` returns the minimum width of the widest component.

public void layoutContainer (Container target)

The `layoutContainer()` method draws `target`'s components on the screen in one column. The width of each component is the width of the container. Each component's height is its `preferredSize()` height, if available.

Miscellaneous methods

public String toString ()

The `toString()` method of `VerticalBagLayout` returns a string with the current vertical gap setting. For example:

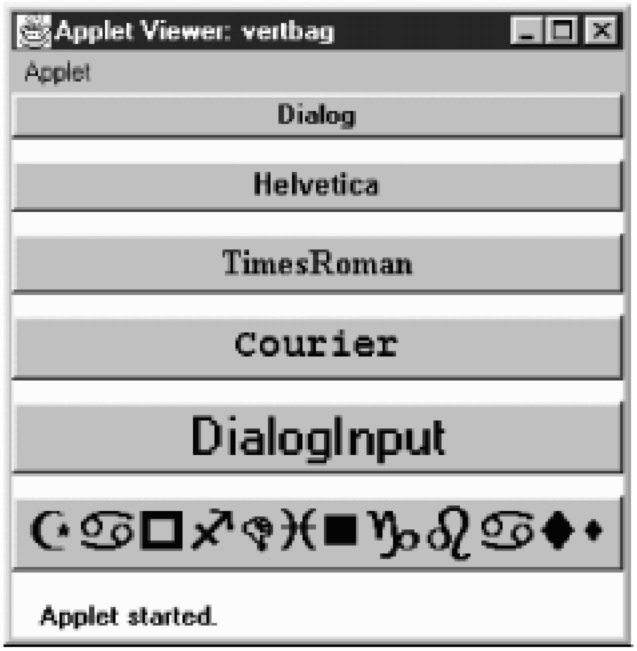
```
sun.awt.VerticalBagLayout[vgap=0]
```

VariableGridLayout

The `VariableGridLayout` builds upon the `GridLayout`. It arranges components on a grid of rows and columns. However, instead of giving all components the same size, the `VariableGridLayout` allows you to size rows and columns fractionally. Another difference between `VariableGridLayout` and `GridBagLayout` is that a `VariableGridLayout` has a fixed size. If you ask for a 3x3 grid, you will get exactly that. The layout manager throws the `ArrayIndexOutOfBoundsException` run-time exception if you try to add too many components.

Figure 7.18 shows a `VariableGridLayout` in which row one takes up 50 percent of the screen, and rows two and three take up 25 percent of the screen each. Column one takes up 50 percent of the screen; columns two and three take 25 percent each.

Figure 7.18: VariableGridLayout in Netscape Navigator



Here is the code that creates Figure 7.18:

```
import java.awt.*;
java.applet.Applet;
import sun.awt.VariableGridLayout;
public class vargrid extends Applet {
    public void init () {
        VariableGridLayout vgl;
        setLayout (vgl = new VariableGridLayout (3,3));
        vgl.setRowFraction (0, 1.0/2.0);
        vgl.setRowFraction (1, 1.0/4.0);
        vgl.setRowFraction (2, 1.0/4.0);
        vgl.setColFraction (0, 1.0/2.0);
        vgl.setColFraction (1, 1.0/4.0);
        vgl.setColFraction (2, 1.0/4.0);
        add (new Button ("One"));
        add (new Button ("Two"));
        add (new Button ("Three"));
        add (new Button ("Four"));
        add (new Button ("Five"));
        add (new Button ("Six"));
        add (new Button ("Seven"));
        add (new Button ("Eight"));
        add (new Button ("Nine"));
    }
}
```

Constructors

public VariableGridLayout (int rows, int columns)

> This constructor creates a `VariableGridLayout` with the specified number of `rows` and `columns`. You cannot specify zero for one dimension. If either `rows` or `columns` is zero, the constructor throws the `NullPointerException` run-time exception. This constructor uses the default values for horizontal and vertical gaps (zero pixels), which means that components in adjacent cells will touch each other.

public VariableGridLayout (int rows, int columns, int hgap, int vgap)

> This version of the constructor is called by the previous one. It creates a `VariableGridLayout` with the specified number of `rows` and `columns`, a horizontal gap of `hgap`, and a vertical gap of `vgap`. The gaps specify in pixels the space between adjacent components in the horizontal and vertical directions. It is possible to have negative gaps if you want components to overlap. You cannot specify zero for the number of rows or columns. If either `rows` or `columns` is zero, the constructor throws the run-time exception `NullPointerException`.

Support methods

The distinguishing feature of a `VariableGridLayout` is that you can tell a particular row or column to take up a certain fraction of the display. By default, the horizontal space available is split evenly among the grid's columns; vertical space is split evenly among the rows. This group of methods lets you find out how much space is allotted to each row or column and lets you change that allocation. The sum of the fractional amounts for each direction should add up to one. If greater than one, part of the display will be drawn offscreen. If less than one, additional screen real estate will be unused.

public void setRowFraction (int rowNumber, double fraction)

> This method sets the percentage of space available for row `rowNumber` to `fraction`.

public void setColFraction (int colNumber, double fraction)

> This method sets the percentage of space available for column `colNumber` to `fraction`.

public double getRowFraction (int rowNumber)

This method returns the current fractional setting for row `rowNumber`.

public double getColFraction (int colNumber)

This method returns the current fractional setting for column `colNumber`.

LayoutManager methods

The only method from `GridLayout` that is overridden is the `layoutContainer()` method.

public void layoutContainer (Container target)

The `layoutContainer()` method draws `target`'s components on the screen in a series of rows and columns. The size of each component within a `VariableGridLayout` is determined by the `RowFraction` and `ColFraction` settings for its row and column.

Miscellaneous methods

public String toString ()

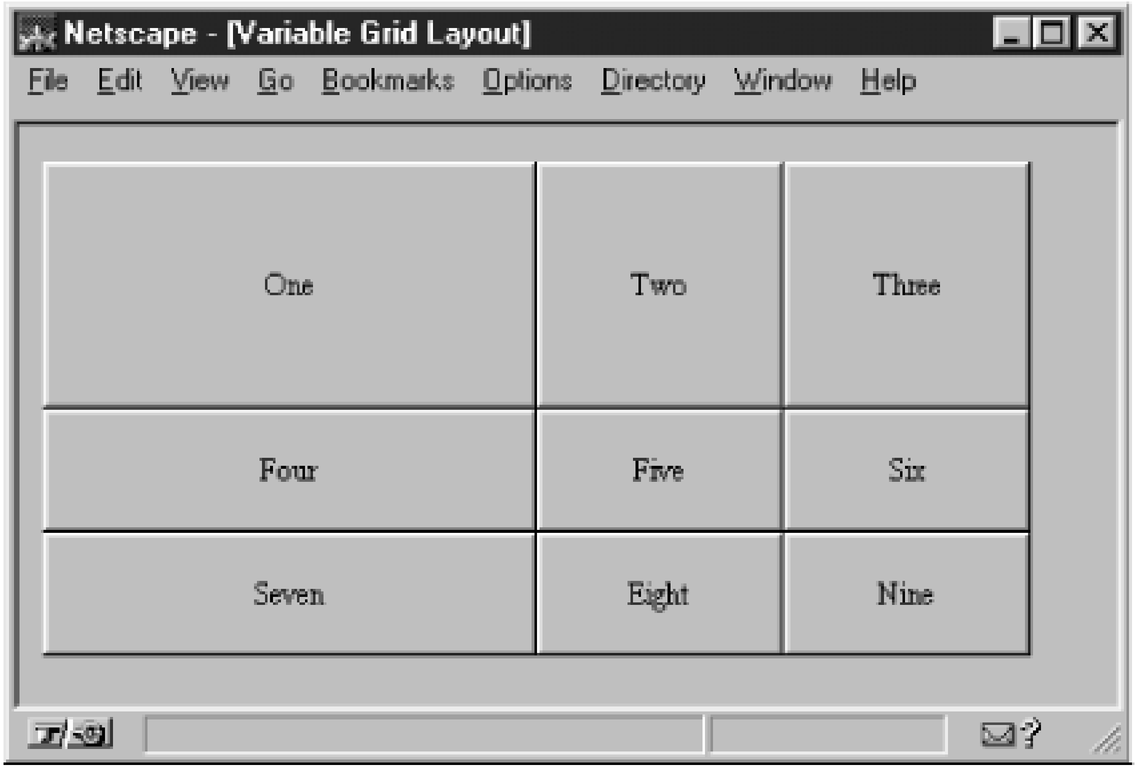
The `toString()` method of `VariableGridLayout` returns a string with the current horizontal and vertical gap settings, the number of rows and columns, and the row and column fractional amounts. For example, the string produced by Figure 7.19 would be:

```
sun.awt.VariableGridLayout[hgap=0,vgap=0,rows=3,cols=3,
    rowFracs=[3]<0.50><0.25><0.25>,colFracs=[3]<0.50><0.25><0.25>]
```

OrientableFlowLayout

The `OrientableFlowLayout` is available for those who want something like a `FlowLayout` that lets you arrange components from top to bottom. Figure 7.19 shows `OrientableFlowLayout` in use.

Figure 7.19: OrientableFlowLayout


Netscape - [Variable Grid Layout]
File Edit View Go Bookmarks Options Directory Window Help
One
Two
Three
Four
Five
Six
Seven
Eight
Nine

Constants

Since `OrientableFlowLayout` subclasses `FlowLayout`, the `FlowLayout` constants of `LEFT`, `RIGHT`, and `CENTER` are still available.

public static final int HORIZONTAL ★

> The `HORIZONTAL` constant tells the layout manager to arrange components from left to right, like the `FlowLayout` manager.

public static final int VERTICAL ★

> The `VERTICAL` constant tells the layout manager to arrange components from top to bottom.

public static final int TOP ★

> The `TOP` constant tells the layout manager to align the first component at the top of the screen (top justification).

public static final int BOTTOM ★

> The `BOTTOM` constant tells the layout manager to align the first component at the bottom of the screen (bottom justification).

Constructors

public OrientableFlowLayout () ★

This constructor creates a `OrientableFlowLayout` that acts like the default `FlowLayout`. The objects flow from left to right and have an `hgap` and `vgap` of 5.

public OrientableFlowLayout (int direction) ★

This constructor creates a `OrientableFlowLayout` in the given `direction`. Valid values are `OrientableFlowLayout.HORIZONTAL` or `OrientableFlowLayout.VERTICAL`.

public OrientableFlowLayout (int direction, int horizAlignment, int vertAlignment) ★

This constructor creates a `OrientableFlowLayout` in the given direction. Valid values are `OrientableFlowLayout.HORIZONTAL` or `OrientableFlowLayout.VERTICAL`. `horizAlignment` provides the horizontal alignment setting. `vertAlignment` provides a vertical alignment setting; it may be `OrientableFlowLayout.TOP`, `FlowLayout.CENTER`, or `OrientableFlowLayout.BOTTOM`. If `direction` is `HORIZONTAL`, the vertical alignment is ignored. If `direction` is `VERTICAL`, the horizontal alignment is ignored.

public OrientableFlowLayout (int direction, int horizAlignment, int vertAlignment, int horizHgap, int horizVgap, int vertHgap, int vertVgap) ★

The final constructor adds separate horizontal and vertical gaps to the settings of `OrientableFlowLayout`. The `horizHgap` and `horizVgap` parameters are the gaps when horizontally aligned. The `vertHgap` and `vertVgap` parameters are the gaps when vertically aligned.

LayoutManager methods

public Dimension preferredLayoutSize (Container target) ★

The `preferredLayoutSize()` method of `OrientableFlowLayout` calculates the preferred dimensions for the `target` container. The `OrientableFlowLayout` computes the preferred size by placing all the components in one row or column, depending upon the current orientation, and adding their individual preferred sizes along with gaps and insets.

public Dimension minimumLayoutSize (Container target) ★

The `minimumLayoutSize()` method of `OrientableFlowLayout` calculates the minimum dimensions for the container by adding up the sizes of the components. The `OrientableFlowLayout` computes the minimum size by placing all the components in one row or column, depending upon the current orientation, and adding their individual minimum sizes along with gaps and insets.

public void layoutContainer (Container target) ★

The `layoutContainer()` method draws `target`'s `Component`s on the screen, starting with the first row or column of the display, and going from left to right across the screen, or from top to bottom, based on the current orientation. When it reaches the margin of the container, it skips to the next row or column and continues drawing additional components.

Miscellaneous methods

public void orientHorizontally () ★

The `orientHorizontally()` method allows you to change the orientation of the `LayoutManager` to horizontal. The container must be validated before you see the effect of the change.

public void orientVertically () ★

The `orientVertically()` method allows you to change the orientation of the `LayoutManager` to vertical. The container must be validated before you see the effect of the change.

public String toString () ★

The `toString()` method of `OrientableFlowLayout` returns a string with the current orientation setting, along with the entire `FlowLayout.toString()` results. For example:

```
sun.awt.OrientableFlowLayout[orientation=vertical,
sun.awt.OrientableFlowLayout[hgap=5,vgap=5,align=center]]
```

7.12 Other Layouts Available on the Net

Many custom layout managers are available on the Internet. Many of these duplicate the layout behavior of other environments. For example, the `FractionalLayout` is based on Smalltalk's positioning mechanism; it is located at http://www.mcs.net/~elunt/Java/FractionalLayoutDescription.html. The `RelativeLayout` allows you to position components relative to others, similar to an X Window form; you can find it at http://www-elec.enst.fr/java/RelativeLayout.java. If you like the way Tcl/Tk arranges widgets, try the `PackerLayout`; it is available at http://www.geom.umn.edu/~daeron/apps/ui/pack/gui.html. If none of these suit you, you can find a collection of links to custom layout managers at http://www.softbear.com/people/larry/javalm.htm. Gamelan (http://www.gamelan.com/) is always a good source for Java classes; try searching for `LayoutManager`.

8.2 TextField

`TextField` is the `TextComponent` for single-line input. Some constructors permit you to set the width of the `TextField` on the screen, but the current `LayoutManager` may change it. The text in the `TextField` is left justified, and the justification is not customizable. To change the font and size of text within the `TextField`, call `setFont()` as shown in Chapter 3, *Fonts and Colors*.

The width of the field does not limit the number of characters that the user can type into the field. It merely suggests how wide the field should be. To limit the number of characters, it is necessary to override the `keyDown()` method for the `Component`. Extending TextField contains an example showing how to do this.

TextField Methods

Constructors

public TextField ()

> This constructor creates an empty `TextField`. The width of the `TextField` is zero columns, but it will be made wide enough to display just about one character, depending on the current font and size.

public TextField (int columns)

> This constructor creates an empty `TextField`. The `TextField` width is `columns`. The `TextField` will try to be wide enough to display `columns` characters in the current font and size. As I mentioned previously, the layout manager may change the size.

public TextField (String text)

> This constructor creates a `TextField` with `text` as its content. In Java 1.0 systems, the `TextField` is 0 columns wide (the `getColumns()` result), but the system will size it to fit the length of text. With Java 1.1, `getColumns()` actually returns `text.length`.

public TextField (String text, int columns)

> This constructor creates a `TextField` with `text` as its content and a width of `columns`.

The following example uses all four constructors; the results are shown in Figure 8.2. With the third constructor, you see that the `TextField` is not quite wide enough for our text. The system uses an average width per character to try to determine how wide the field should be. If you want to be on the safe side, specify the field's length explicitly, and add a few extra characters to ensure that there is enough room on the screen for the entire text.

```
import java.awt.TextField;
public class texts extends java.applet.Applet {
   public void init () {
        add (new TextField ());                      // A
        add (new TextField (15));                    // B
        add (new TextField ("Empty String"));        // C
        add (new TextField ("Empty String", 20)); // D
   }
}
```

Figure 8.2: Using the TextField constructors

[Graphic: Figure 8-2]

Sizing

public int getColumns ()

> The `getColumns()` method returns the number of columns set with the constructor or a later call to `setColumns()`. This could be different from the displayed width of the `TextField`, depending upon the current `LayoutManager`.

public void setColumns (int columns) ★

> The `setColumns()` method changes the preferred number of columns for the `TextField` to display to `columns`. Because the current `LayoutManager` will do what it wants, the new setting may be completely ignored. If `columns` < 0, `setColumns()` throws the run-time exception `IllegalArgumentException`.

public Dimension getPreferredSize (int columns) ★
public Dimension preferredSize (int columns) ☆

The `getPreferredSize()` method returns the `Dimension` (width and height) for the preferred size of a `TextField` with a width of `columns`. The `columns` specified may be different from the number of columns designated in the constructor.

`preferredSize()` is the Java 1.0 name for this method.

public Dimension getPreferredSize () ★
public Dimension preferredSize () ☆

The `getPreferredSize()` method returns the `Dimension` (width and height) for the preferred size of the `TextField`. Without the `columns` parameter, this `getPreferredSize()` uses the constructor's number of columns (or the value from a subsequent call to `setColumns()`) to calculate the `TextField`'s preferred size.

`preferredSize()` is the Java 1.0 name for this method.

public Dimension getMinimumSize (int columns) ★
public Dimension minimumSize (int columns) ☆

The `getMinimumSize`() method returns the minimum `Dimension` (width and height) for the size of a `TextField` with a width of `columns`. The `columns` specified may be different from the columns designated in the constructor.

`minimumSize()` is the Java 1.0 name for this method.

public Dimension getMinimumSize () ★
public Dimension minimumSize ()

The `getMinimumSize()` method returns the minimum `Dimension` (width and height) for the size of the `TextField`. Without the columns parameter, this `getMinimumSize()` uses the constructor's number of columns (or the value from a subsequent call to `setColumns()`) to calculate the `TextField`'s minimum size.

`minimumSize()` is the Java 1.0 name for this method.

Echoing character

It is possible to change the character echoed back to the user when he or she types. This is extremely useful for implementing password entry fields.

public char getEchoChar ()

The `getEchoChar()` method returns the currently echoed character. If the `TextField` is echoing normally, `getEchoChar()` returns zero.

public void setEchoChar (char c) ★
public void setEchoCharacter (char c) ☆

The `setEchoChar()` method changes the character that is displayed to the user to `c` for every character in the `TextField`. It is possible to change the echo character on the fly so that existing characters will be replaced. A `c` of zero, `(char)0`, effectively turns off any change and makes the `TextField` behave normally.

`setEchoCharacter()` is the Java 1.0 name for this method.

public boolean echoCharIsSet ()

The `echoCharIsSet()` method returns `true` if the echo character is set to a nonzero value. If the `TextField` is displaying input normally, this method returns `false`.

Miscellaneous methods

public synchronized void addNotify ()

The `addNotify()` method creates the `TextField` peer. If you override this method, first call `super.addNotify()`, then add your customizations for the new class. Then you will be able to do everything you need with the information about the newly created peer.

protected String paramString ()

When you call the `toString()` method of `TextField`, the default `toString()` method of `Component` is called. This in turn calls `paramString()`, which builds up the string to display. The `TextField` level can add only one item. If the echo character is nonzero, the current echo character is added (the method `getEchoChar()`). Using `` new TextField (`Empty String`, 20) ``, the results displayed could be:

```
java.awt.TextField[0,0,0x0,invalid,text="Empty String",editable,selection=0-0]
```

TextField Events

With the 1.0 event model, `TextField` components can generate `KEY_PRESS` and `KEY_ACTION` (which calls `keyDown()`), `KEY_RELEASE` and `KEY_ACTION_RELEASE` (which calls `keyUp()`), and `ACTION_EVENT` (which calls `action()`).

With the 1.1 event model, you register an `ActionListener` with the method `addActionListener()`. Then when the user presses Return within the `TextField` the `ActionListener.actionPerformed()` method is called through the protected `TextField.processActionEvent()` method. Key, mouse, and focus listeners are registered through the three `Component` methods of `addKeyListener()`, `addMouseListener()`, and `addFocusListener()`, respectively. Action

public boolean action (Event e, Object o)

> The `action()` method for a `TextField` is called when the input focus is in the `TextField` and the user presses the Return key. `e` is the `Event` instance for the specific event, while `o` is a `String` representing the current contents (the `getText()` method).

Keyboard

public boolean keyDown (Event e, int key)

> The `keyDown()` method is called whenever the user presses a key. `keyDown()` may be called many times in succession if the key remains pressed. `e` is the `Event` instance for the specific event, while `key` is the integer representation of the character pressed. The identifier for the event (`e.id`) for `keyDown()` could be either `Event.KEY_PRESS` for a regular key or `Event.KEY_ACTION` for an action-oriented key (i.e., an arrow or function key). Some of the things you can do through this method are validate input, convert each character to uppercase, and limit the number or type of characters entered. The technique is simple: you just need to remember that the user's keystroke is actually displayed by the `TextField` peer, which receives the event after the `TextField` itself. Therefore, a `TextField` subclass can modify the character displayed by modifying the `key` field (`e.key`) of the `Event` and returning `false`, which passes the `Event` on down the chain; remember that returning `false` indicates that the `Event` has not been completely processed. The following method uses this technique to convert all input to uppercase.

```
public boolean keyDown (Event e, int key) {
    e.key = Character.toUppercase (char(key));
    return false;
}
```

> If `keyDown()` returns `true`, it indicates that the `Event` has been completely processed. In this case, the `Event` never propagates to the peer, and the keystroke is never displayed.

public boolean keyUp (Event e, int key)

> The `keyUp()` method is called whenever the user releases a key. `e` is the `Event` instance for the specific event, while `key` is the integer representation of the character pressed. The identifier for the event (`e.id`) for `keyUp()` could be either `Event.KEY_RELEASE` for a regular key or `Event.KEY_ACTION_RELEASE` for an action-oriented key (i.e., an arrow or function key). Among other things, `keyUp()` may be used to determine how long the key has been pressed.

Mouse

Ordinarily, the `TextField` component does not trigger any mouse events.

> **NOTE:**

Mouse events are not generated for `TextField` with JDK 1.0.2. Your run-time environment may behave differently. See Appendix C for more information about platform dependencies.

Focus

The `TextField` component does not reliably generate focus events.

> **NOTE:**

The `GOT_FOCUS` and `LOST_FOCUS` events can be generated by `TextField`s, but these events are not reliable across platforms. With Java 1.0, they are generated on most UNIX platforms but not on Windows NT/95 platforms. They are generated on all platforms under Java 1.1. See Appendix C for more information about platform dependencies.

public boolean gotFocus (Event e, Object o)

> The `gotFocus()` method is triggered when the `TextField` gets the input focus. `e` is the `Event` instance for the specific event, while `o` is a `String` representation of the current contents (`getText()`).

public boolean lostFocus (Event e, Object o)

> The `lostFocus()` method is triggered when the input focus leaves the `TextField`. `e` is the `Event` instance for the specific event, while `o` is a `String` representation of the current contents (`getText()`).

Listeners and 1.1 event handling

With the 1.1 event model, you register event listeners that are told when an event occurs. You can register text event listeners by calling the method `TextComponent.addTextListener()`.

public void addActionListener(ActionListener listener) ★

> The `addActionListener()` method registers `listener` as an object interested in receiving notifications when an `ActionEvent` passes through the `EventQueue` with this `TextField` as its target. The `listener.actionPerformed()` method is called when these events occur. Multiple listeners can be registered. The following code demonstrates how to use an `ActionListener` to reverse the text in the `TextField`.

```
// Java 1.1 only
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
class MyAL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println ("The current text is: " +
            e.getActionCommand());
        if (e.getSource() instanceof TextField) {
            TextField tf = (TextField)e.getSource();
            StringBuffer sb = new StringBuffer (e.getActionCommand());
            tf.setText (sb.reverse().toString());
        }
    }
}
public class text11 extends Applet {
    public void init () {
        TextField tf = new TextField ("Help Text", 20);
        add (tf);
        tf.addActionListener (new MyAL());
   }
}
```

public void removeActionListener(ActionListener listener) ★

The `removeActionListener()` method removes `listener` as a interested listener. If `listener` is not registered, nothing happens.

protected void processEvent(AWTEvent e) ★

The `processEvent()` method receives all `AWTEvent`s with this `TextField` as its target. `processEvent()` then passes them along to any listeners for processing. When you subclass `TextField`, overriding `processEvent()` allows you to process all events yourself, before sending them to any listeners. In a way, overriding `processEvent()` is like overriding `handleEvent()` using the 1.0 event model.

If you override `processEvent()`, remember to call `super.processEvent(e)` last to ensure that regular event processing can occur. If you want to process your own events, it's a good idea to call `enableEvents()` (inherited from `Component`) to ensure that events are delivered even in the absence of registered listeners.

protected void processActionEvent(ActionEvent e) ★

The `processActionEvent()` method receives all `ActionEvent`s with this `TextField` as its target. `processActionEvent()` then passes them along to any listeners for processing. When you subclass `TextField`, overriding the method `processActionEvent()` allows you to process all action events yourself, before sending them to any listeners. In a way, overriding `processActionEvent()` is like overriding `action()` using the 1.0 event model.

If you override the `processActionEvent()` method, remember to call `super.processActionEvent(e)` last to ensure that regular event processing can occur. If you want to process your own events, it's a good idea to call `enableEvents()` (inherited from `Component`) to ensure that events are delivered even in the absence of registered listeners.

The following applet is equivalent to the previous example, except that it overrides `processActionEvent()` to receive events, eliminating the need for an ActionListener. The constructor calls enableEvents() to make sure that events are delivered, even if no listeners are registered.

```
// Java 1.1 only
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
class MyTextField extends TextField {
    public MyTextField (String s, int len) {
        super (s, len);
        enableEvents (AWTEvent.ACTION_EVENT_MASK);
    }
    protected void processActionEvent(ActionEvent e) {
        System.out.println ("The current text is: " +
            e.getActionCommand());
        TextField tf = (TextField)e.getSource();
        StringBuffer sb = new StringBuffer (e.getActionCommand());
        tf.setText (sb.reverse().toString());
        super.processActionEvent(e)
    }
}
public class text12 extends Applet {
   public void init () {
        TextField tf = new MyTextField ("Help Text", 20);
        add (tf);
   }
}
```

PREVIOUS | HOME | NEXT
Text Component | BOOK INDEX | TextArea

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

8.3 TextArea

TextArea is the TextComponent for multiline input. Some constructors permit you to set the rows and columns of the TextArea on the screen. However, the LayoutManager may change your settings. As with TextField, the only way to limit the number of characters that a user can enter is to override the keyDown() method. The text in a TextArea appears left justified, and the justification is not customizable.

In Java 1.1, you can control the appearance of a TextArea scrollbar; earlier versions gave you no control over the scrollbars. When visible, the vertical scrollbar is on the right of the TextArea, and the horizontal scrollbar is on the bottom. You can remove either scrollbar with the help of several new TextArea constants; you can't move them to another side. When the horizontal scrollbar is not present, the text wraps automatically when the user reaches the right side of the TextArea. Prior to Java 1.1, there was no way to enable word wrap.

TextArea Variables

Constants

The constants for TextArea are new to Java 1.1; they allow you to control the visibility and word wrap policy of a TextArea scrollbar. There is no way to listen for the events when a user scrolls a TextArea.

public static final int SCROLLBARS_BOTH ★

> The SCROLLBARS_BOTH mode is the default for TextArea. It shows both scrollbars all the time and does no word wrap.

public static final int SCROLLBARS_HORIZONTAL_ONLY ★

The `SCROLLBARS_HORIZONTAL_ONLY` mode displays a scrollbar along the bottom of the `TextArea`. When this scrollbar is present, word wrap is disabled.

public static final int SCROLLBARS_NONE ★

The `SCROLLBARS_NONE` mode displays no scrollbars around the `TextArea` and enables word wrap. If the text is too long, the `TextArea` displays the lines surrounding the cursor. You can use the cursor to move up and down within the `TextArea`, but you cannot use a scrollbar to navigate. Because this mode has no horizontal scrollbar, word wrap is enabled.

public static final int SCROLLBARS_VERTICAL_ONLY ★

The `SCROLLBARS_VERTICAL_ONLY` mode displays a scrollbar along the right edge of the `TextArea`. If the text is too long to display, you can scroll within the area. Because this mode has no horizontal scrollbar, word wrap is enabled.

TextArea Methods

Constructors

public TextArea ()

This constructor creates an empty `TextArea` with both scrollbars. The `TextArea` is 0 rows high and 0 columns wide. Depending upon the platform, the `TextArea` could be really small (and useless) or rather large. It is a good idea to use one of the other constructors to control the size of the `TextArea`.

public TextArea (int rows, int columns)

This constructor creates an empty `TextArea` with both scrollbars. The `TextArea` is `rows` high and `columns` wide.

public TextArea (String text)

This constructor creates a `TextArea` with an initial content of `text` and both scrollbars. The `TextArea` is 0 rows high and 0 columns wide. Depending upon the platform, the `TextArea` could be really small (and useless) or rather large. It is a good idea to use one of the other constructors to control the size of the `TextArea`.

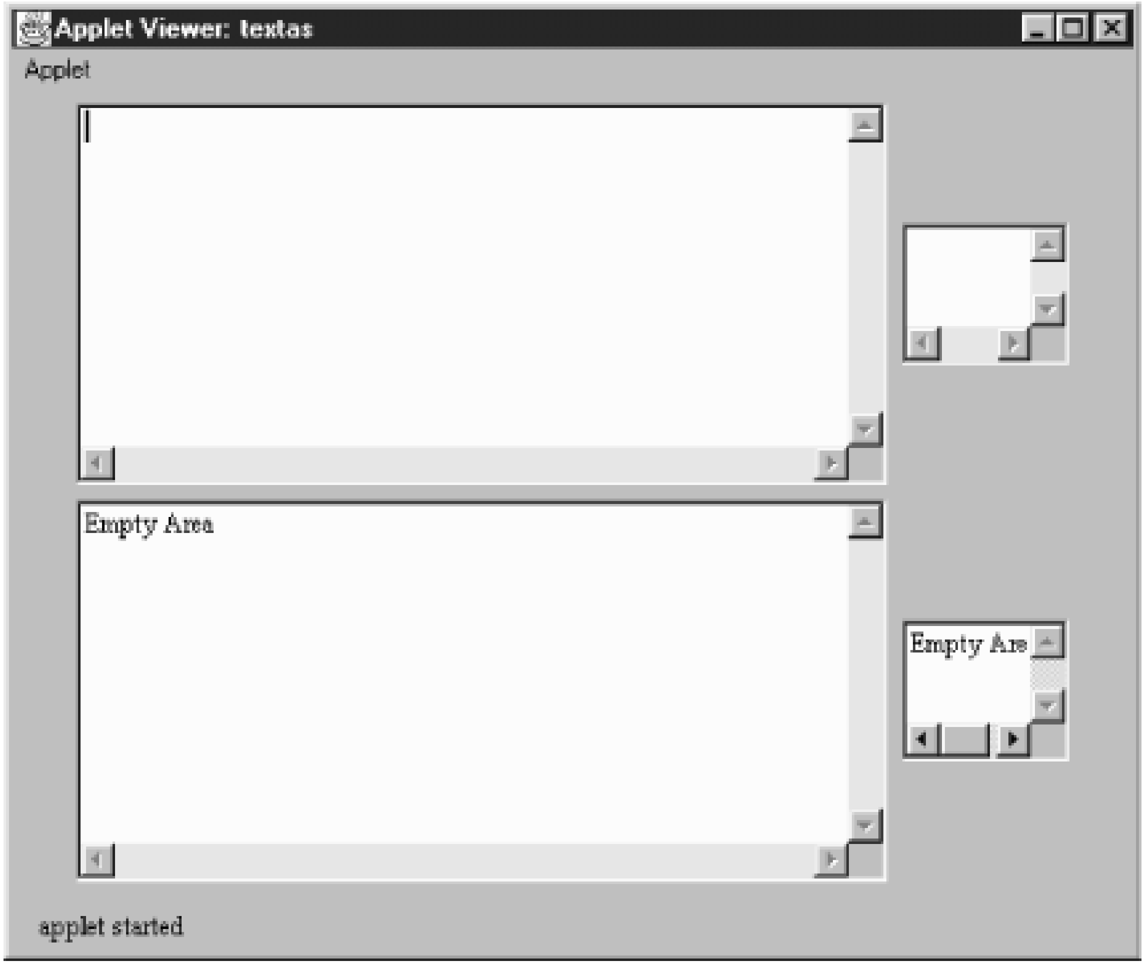
public TextArea (String text, int rows, int columns)

> This constructor creates a `TextArea` with an initial content of `text`. The `TextArea` is `rows` high and `columns` wide and has both scrollbars.

The following example uses the first four constructors. The results are shown in Figure 8.3. With the sizeless constructors, notice that Windows 95 creates a rather large `TextArea`. UNIX systems create a much smaller area. Depending upon the `LayoutManager`, the `TextArea`s could be resized automatically.

```
import java.awt.TextArea;
public class textas extends java.applet.Applet {
    public void init () {
        add (new TextArea ());                      // A
        add (new TextArea (3, 10));                 // B
        add (new TextArea ("Empty Area"));          // C
        add (new TextArea ("Empty Area", 3, 10));  // D
    }
}
```

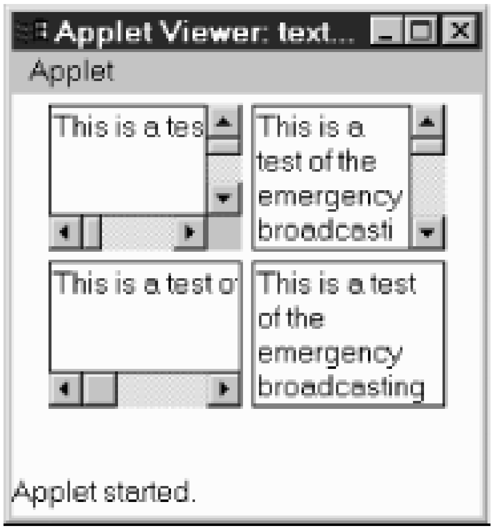
Figure 8.3: TextArea constructor



public TextArea (String text, int rows, int columns, int scrollbarPolicy) ★

> The final constructor creates a `TextArea` with an initial content of `text`. The `TextArea` is `rows` high and `columns` wide. The initial scrollbar display policy is designated by the `scrollbarPolicy` parameter and is one of the `TextArea` constants in the previous example. This constructor is the only way provided to change the scrollbar visibility; there is no `setScrollbarVisibility()` method. Figure 8.4 displays the different settings.

Figure 8.4: TextArea policies



Applet started.

Setting text

The text-setting methods are usually called in response to an external event. When you handle the insertion position, you must translate it from the visual row and column to a one-dimensional position. It is easier to position the insertion point based upon the beginning, end, or current selection (`getSelectionStart()` and `getSelectionEnd()`).

public void insert (String string, int position) ★
public void insertText (String string, int position) ☆

> The `insert()` method inserts `string` at `position` into the `TextArea`. If `position` is beyond the end of the `TextArea`, `string` is appended to the end of the `TextArea`.
>
> `insertText()` is the Java 1.0 name for this method.

public void append (String string) ★
public void appendText (String string) ☆

The `append()` method inserts `string` at the end of the `TextArea`.

`appendText()` is the Java 1.0 name for this method.

public void replaceRange (String string, int startPosition, int endPosition) ★
public void replaceText (String string, int startPosition, int endPosition) ☆

The `replaceRange()` method replaces the text in the current `TextArea` from `startPosition` to `endPosition` with `string`. If `endPosition` is before `startPosition`, it may or may not work as expected. (For instance, on a Windows 95 platform, it works fine when the `TextArea` is displayed on the screen. However, when the `TextArea` is not showing, unexpected results happen. Other platforms may vary.) If `startPosition` is 0 and `endPosition` is the length of the contents, this method functions the same as `TextComponent.setText()`.

`replaceText()` is the Java 1.0 name for this method.

Sizing

public int getRows ()

The `getRows()` method returns the number of rows set by the constructor or a subsequent call to `setRows()`. This could be different from the displayed height of the `TextArea`.

public void setRows (int rows) ★

The `setRows()` method changes the preferred number of rows to display for the `TextField` to `rows`. Because the current `LayoutManager` will do what it wants, the new setting may be ignored. If rows < 0, `setRows()` throws the run-time exception `IllegalArgumentException`.

public int getColumns ()

The `getColumns()` method returns the number of columns set by the constructor or a subsequent call to `setColumns()`. This could be different from the displayed width of the `TextArea`.

public void setColumns (int columns) ★

The `setColumns()` method changes the preferred number of columns to display for the `TextArea` to `columns`. Because the current `LayoutManager` will do what it wants, the new setting may be ignored. If `columns` < 0, `setColumns()` throws the run-time exception `IllegalArgumentException`.

public Dimension getPreferredSize (int rows, int columns) ★
public Dimension preferredSize (int rows, int columns) ☆

The `getPreferredSize()` method returns the `Dimension` (width and height) for the preferred size of the `TextArea` with a preferred height of `rows` and width of `columns`. The `rows` and `columns` specified may be different from the current settings.

`preferredSize()` is the Java 1.0 name for this method.

public Dimension getPreferredSize (int rows, int columns) ★
public Dimension preferredSize () ☆

The `getPreferredSize()` method returns the `Dimension` (width and height) for the preferred size of the `TextArea`. Without the rows and columns parameters, this `getPreferredSize()` uses the constructor's number of rows and columns to calculate the `TextArea`'s preferred size.

`preferredSize()` is the Java 1.0 name for this method.

public Dimension getMinimumSize (int rows, int columns) ★
public Dimension minimumSize (int rows, int columns) ☆

The `getMinimumSize()` method returns the minimum `Dimension` (width and height) for the size of the `TextArea` with a height of `rows` and width of `columns`. The `rows` and `columns` specified may be different from the current settings.

`minimumSize()` is the Java 1.0 name for this method.

public Dimension getMinimumSize () ★
public Dimension minimumSize () ☆

The `getMinimumSize()` method returns the minimum `Dimension` (width and height) for the size of the `TextArea`. Without the rows and columns parameters, this `getMinimumSize()` uses the current settings for rows and columns to calculate the `TextArea`'s minimum size.

`minimumSize()` is the Java 1.0 name for this method.

Miscellaneous methods

public synchronized void addNotify ()

The addNotify() method creates the TextArea peer. If you override this method, call super.addNotify() first, then add your customizations for the new class. You will then be able to do everything you need with the information about the newly created peer.

public int getScrollbarVisibility() ★

The getScrollbarVisibility() method retrieves the scrollbar visibility setting, which is set by the constructor. There is no setScollbarVisibility() method to change the setting. The return value is one of the TextArea constants: SCROLLBARS_BOTH, SCROLLBARS_HORIZONTAL_ONLY, SCROLLBARS_NONE, or SCROLLBARS_VERTICAL_ONLY.

protected String paramString ()

When you call the toString() method of TextArea, the default toString() method of Component is called. This in turn calls paramString(), which builds up the string to display. The TextArea level adds the number of rows and columns for the TextArea, and Java 1.1 adds the scrollbar visibility policy. Using new TextArea(`Empty Area`, 3, 10), the results displayed could be:

```
java.awt.TextArea[text0,0,0,0x0,invalid,text="Empty Area",
editable,selection=0-0, rows=3,columns=10, scrollbarVisibility=both]
```

TextArea Events

With the 1.0 event model, the TextArea component can generate KEY_PRESS and KEY_ACTION (which calls keyDown()) along with KEY_RELEASE and KEY_ACTION_RELEASE (which called keyUp()). There is no ACTION_EVENT generated for TextArea.

NOTE:

The GOT_FOCUS and LOST_FOCUS events can be generated by this component but not reliably across platforms. Currently, they are generated on most UNIX platforms but not on Microsoft Windows NT/95 under Java 1.0. These events are generated under Java 1.1.

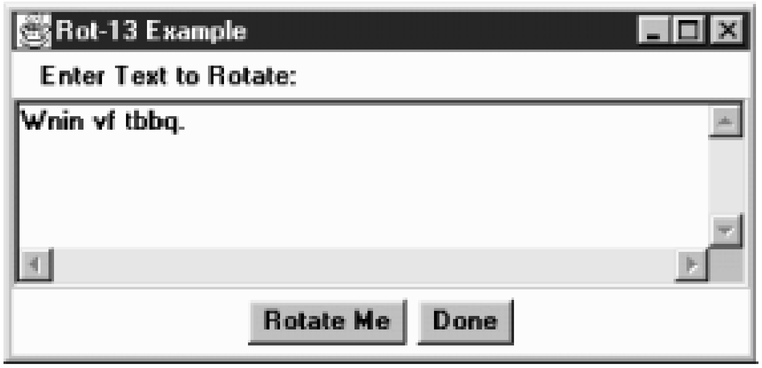
Similarly, the mouse events are not generated with JDK 1.0.2. See Appendix C for more information about platform dependencies.

With the Java 1.1 event model, there are no listeners specific to TextArea. You can register key, mouse, and focus listeners through the Component methods of addKeyListener(), addMouseListener(), and addFocusListener(), respectively. To register listeners for text events, call TextComponent.addTextListener(). Action

The TextArea component has no way to trigger the action event, since carriage return is a valid character. You would need to put something like a Button on the screen to cause an action for a TextArea. The following Rot13 program demonstrates this technique. The user enters text in the TextArea and selects the Rotate Me button to rotate the text. If the user selects Rotate Me again, it rotates again, back to the original position. Without the button, there would be no way to trigger the event. Figure 8.5 shows this example in action.

```
import java.awt.*;
public class Rot13 extends Frame {
    TextArea ta;
    Component rotate, done;
    public Rot13 () {
        super ("Rot-13 Example");
        add ("North", new Label ("Enter Text to Rotate:"));
        ta = (TextArea)(add ("Center", new TextArea (5, 40)));
        Panel p = new Panel ();
        rotate = p.add (new Button ("Rotate Me"));
        done = p.add (new Button ("Done"));
        add ("South", p);
    }
    public static void main (String args[]) {
        Rot13 rot = new Rot13();
        rot.pack();
        rot.show();
    }
    public boolean handleEvent (Event e) {
        if (e.id == Event.WINDOW_DESTROY) {
            hide();
            dispose();
            System.exit (0);
            return true;
        }
        return super.handleEvent (e);
    }
    public boolean action (Event e, Object o) {
        if (e.target == rotate) {
            ta.setText (rot13Text (ta.getText()));
            return true;
        } else if (e.target == done) {
            hide();
            dispose();
            System.exit (0);
        }
        return false;
    }
    String rot13Text (String s) {
        int len = s.length();
        StringBuffer returnString = new StringBuffer (len);
        char c;
        for (int i=0;i<len;i++) {
            c = s.charAt (i);
            if (((c >= 'A') && (c <= 'M')) ||
                ((c >= 'a') && (c <= 'm')))
                c += 13;
            else if (((c >= 'N') && (c <= 'Z')) ||
                ((c >= 'n') && (c <= 'z')))
                c -= 13;
            returnString.append (c);
        }
        return returnString.toString();
    }
}
```

Figure 8.5: TextArea with activator button



Rot-13 Example

Enter Text to Rotate:

Wnin vf tbbq.

Keyboard

Ordinarily, the `TextArea` component generates all the key events.

public boolean keyDown (Event e, int key)

The keyDown() method is called whenever the user presses a key. keyDown() may be called many times in succession if the key remains pressed. e is the Event instance for the specific event, while key is the integer representation of the character pressed. The identifier for the event (e.id) for keyDown() could be either Event.KEY_PRESS for a regular key or Event.KEY_ACTION for an action-oriented key (i.e., an arrow or function key). Some of the things you can do through this method are validate input, convert each character to uppercase, and limit the number or type of characters entered. The technique is simple: you just need to remember that the user's keystroke is actually displayed by the TextArea peer, which receives the event after the TextArea itself. Therefore, a TextArea subclass can modify the character displayed by modifying the key field (e.key) of the Event and returning false, which passes the Event on down the chain; remember that returning false indicates that the Event has not been completely processed. The following method uses this technique to convert all alphabetic characters to the opposite case:

```
public boolean keyDown (Event e, int key) {
    if (Character.isUpperCase ((char)key)) {
        e.key = Character.toLowerCase ((char)key);
    } else if (Character.isLowerCase ((char)key)) {
        e.key = Character.toUpperCase ((char)key);
    }
    return false;
}
```

If keyDown() returns true, it indicates that the Event has been completely processed. In this case, the Event never propagates to the peer, and the keystroke is never displayed.

public boolean keyUp (Event e, int key)

The keyUp() method is called whenever the user releases a key. e is the Event instance for the specific event, while key is the integer representation of the character pressed. The identifier for the event (e.id) for keyUp() could be either Event.KEY_RELEASE for a regular key, or Event.KEY_ACTION_RELEASE for an action-oriented key (i.e., an arrow or function key).

Mouse

Ordinarily, the TextArea component does not trigger any mouse events.

NOTE:

Mouse events are not generated for TextArea with JDK 1.0.2. See Appendix C for more information about platform dependencies.

Focus

The `TextArea` component does not reliably generate focus events.

> **NOTE:**

The `GOT_FOCUS` and `LOST_FOCUS` events can be generated by this component but not reliably across platforms. With the JDK, they are generated on most UNIX platforms but not on Microsoft Windows NT/95 under JDK 1.0. These events are generated with JDK 1.1. See Appendix C for more information about platform dependencies.

public boolean gotFocus (Event e, Object o)

> The `gotFocus()` method is triggered when the `TextArea` gets the input focus. `e` is the `Event` instance for the specific event, while `o` is a `String` representation of the current contents (`getText()`).

public boolean lostFocus (Event e, Object o)

> The `lostFocus()` method is triggered when the input focus leaves the `TextArea`. `e` is the `Event` instance for the specific event, while `o` is a `String` representation of the current contents (`getText()`).

Listeners and 1.1 event handling

There are no listeners specific to the `TextArea` class. You can register Key, mouse, and focus listeners through the `Component` methods of `addKeyListener()`, `addMouseListener()`, and `addFocusListener()`, respectively. Also, you register listeners for text events by calling `TextComponent.addTextListener()`.

8.4 Extending TextField

To extend what you learned so far, Example 8.1 creates a sub-class of `TextField` that limits the number of characters a user can type into it. Other than the six constructors, all the work is in the `keyDown()` method. The entire class follows.

Example 8.1: The SizedTextField Class Limits the Number of Characters a User can Type

```
import java.awt.*;
public class SizedTextField extends TextField {
    private int size;  // size = 0 is unlimited
    public SizedTextField () {
        super ("");
        this.size = 0;
    }
    public SizedTextField (int columns) {
        super (columns);
        this.size = 0;
    }
    public SizedTextField (int columns, int size) {
        super (columns);
        this.size = Math.max (0, size);
    }
    public SizedTextField (String text) {
        super (text);
        this.size = 0;
    }
    public SizedTextField (String text, int columns) {
        super (text, columns);
        this.size = 0;
    }
    public SizedTextField (String text, int columns, int size) {
        super (text, columns);
        this.size = Math.max (0, size);
    }
    public boolean keyDown (Event e, int key) {
        if ((e.id == Event.KEY_PRESS) && (this.size > 0) &&
            (((TextField)(e.target)).getText ().length () >= this.size)) {
            // Check for backspace / delete / tab--let these pass through
            if ((key == 127) || (key == 8) || (key == 9)) {
                return false;
            }
            return true;
        }
        return false;
    }
    protected String paramString () {
        String str = super.paramString ();
        if (size != 0) {
            str += ",size=" + size;
        }
        return str;
    }
}
```

Most of the `SizedTextField` class consists of constructors; you really don't need to provide an equivalent to all the superclass's constructors, but it's not a bad idea. The `keyDown()` method looks at what the user types before it reaches the screen and acts accordingly. It checks the length of the `TextField` and compares it to the maximum length. It then does another check to see if the user typed a Backspace, Delete, or Tab, all of which we want to allow: if the field has gotten too long, we want to allow the user to shorten it. We also want to allow tab under all circumstances, so that focus traversal works properly. The rest of the logic is simple:

- If the user typed Backspace, Delete, or Tab, return `false` to propagate the event.

- If the field is too long, return `true` to prevent the event from reaching the peer. This effectively ignores the character.

PREVIOUS | HOME | NEXT
TextArea | BOOK INDEX | Pick Me

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

9.2 Lists

Like the `Choice` component, the `List` provides a way to present your user with a fixed sequence of choices to select. However, with `List`, several items can be displayed at a time on the screen. A `List` can also allow multiple selection, so that more than one choice can be selected.

Normally, a scrollbar is associated with the `List` to enable the user to move to the items that do not fit on the screen. On some platforms, the `List` may not display the scrollbar if there is enough room to display all choices. A `List` can be resized by the `LayoutManager` according to the space available. Figure 9.2 shows two lists, one of which has no items to display.

List Methods

Constructors

public List ()

> This constructor creates an empty `List` with four visible lines. You must rely on the current `LayoutManager` to resize the `List` or override the `preferredSize()` (version 1.0) or `getPreferredSize()` (version 1.1) method to affect the size of the displayed `List`. A `List` created with this constructor is in single-selection mode, so the user can select only one item at a time.

public List (int rows)

> This constructor creates a `List` that has `rows` visible lines. This is just a request; the `LayoutManager` is free to adjust the height of the `List` to some other amount based upon available space. A `List` created with this constructor is in single-selection mode, so the user will be able to select only one item at a time.

public List (int rows, boolean multipleSelections)

The final constructor for `List` creates a `List` that has `rows` visible lines. This is just a request; the `LayoutManager` is free to adjust the height of the `List` to some other amount based upon available space. If `multipleSelections` is `true`, this `List` permits multiple items to be selected. If `false`, this is a single-selection list.

Figure 9.2: Two lists; the list on the right is empty

[Graphic: Figure 9-2]

Content control

public int getItemCount () ★
public int countItems () ☆

The `getItemCount()` method returns the length of the list. The length of the list is the number of items in the list, not the number of visible rows.

`countItems()` is the Java 1.0 name for this method.

public String getItem (int index)

The `getItem()` method returns the `String` representation for the item at position `index`. The `String` is the parameter passed to the `addItem()` or `add()` method.

public String[] getItems () ★

The `getItems()` method returns a `String` array that contains all the elements in the `List`. This method does not care if an item is selected or not.

public synchronized void add (String item) ★
public synchronized void addItem (String item) ☆

The `add()` method adds `item` as the last entry in the `List`. If `item` already exists in the list, this method adds it again.

`addItem()` is the Java 1.0 name for this method.

public synchronized void add (String item, int index) ★
public synchronized void addItem (String item, int index) ☆

This version of the `add()` method has an additional parameter, `index`, which specifies where to add `item` to the `List`. If `index < 0` or `index >= getItemCount()`, `item` is added to the end of the `List`. The position count is zero based, so if `index` is 0, it will be added as the first item.

`addItem()` is the Java 1.0 name for this method.

public synchronized void replaceItem (String newItem, int index)

The `replaceItem()` method replaces the contents at position `index` with `newItem`. If the item at `index` has been selected, `newItem` will not be selected.

public synchronized void removeAll () ★
public synchronized void clear () ☆

The `removeAll()` method clears out all the items in the list.

`clear()` is the Java 1.0 name for this method.

NOTE:

Early versions (Java1.0) of the `clear()` method did not work reliably across platforms. You were better off calling the method `listVar.delItems(0, listVar.countItems()-1)`, where `listVar` is your `List` instance.

public synchronized void remove (String item) ★

The `remove()` method removes `item` from the list of available choices. If `item` appears in the `List` several times, only the first instance is removed. If `item` is `null`, `remove()` throws the run-time exception `NullPointerException`. If `item` is not found in the `List`, `remove()` throws the `IllegalArgumentException` run-time exception.

public synchronized void remove (int position) ★
public synchronized void delItem (int position) ☆

> The `remove()` method removes the entry at `position` from the `List`. If `position` is invalid--either `position` < 0 or `position` >= `getItemCount()`--`remove()` throws the `ArrayIndexOutOfBoundsException` run-time exception with a message indicating that `position` was invalid.
>
> `delItem()` is the Java 1.0 name for this method.

public synchronized void delItems (int start, int end) ☆

> The `delItems()` method removes entries from position `start` to position `end` from the `List`. If either parameter is invalid--either `start` < 0 or `end` >= `getItemCount()`--`delItems()` throws the `ArrayIndexOutOfBoundsException` run-time exception with a message indicating which position was invalid. If `start` is greater than `end`, nothing happens.

Selection and positioning

public synchronized int getSelectedIndex ()

> The `getSelectedIndex()` method returns the position of the selected item. If nothing is selected in the `List`, `getSelectedIndex()` returns -1. The value -1 is also returned if the `List` is in multiselect mode and multiple items are selected. For multiselection lists, use `getSelectedIndexes()` instead.

public synchronized int[] getSelectedIndexes ()

> The `getSelectedIndexes()` method returns an integer array of the selected items. If nothing is selected, the array will be empty.

public synchronized String getSelectedItem ()

> The `getSelectedItem()` method returns the label of the selected item. The label is the string used in the `add()` or `addItem()` call. If nothing is selected in the `List`, `getSelectedItem()` returns `null`. The return value is also `null` if `List` is in multiselect mode and multiple items are selected. For multiselection lists, use `getSelectedItems()` instead.

public synchronized String[] getSelectedItems ()

The `getSelectedItems()` method returns a `String` array of the selected items. If nothing is selected, the array is empty.

public synchronized Object[] getSelectedObjects ()

The `getSelectedObjects()` method returns the results of the method `getSelectedItems()` as an `Object` array instead of a `String` array, to conform to the `ItemSelectable` interface. If nothing is selected, the returned array is empty.

public synchronized void select (int index)

The `select()` method selects the item at position `index`, which is zero based. If the `List` is in single-selection mode, any other selected item is deselected. If the `List` is in multiple-selection mode, calling this method has no effect on the other selections. The item at position `index` is made visible.

NOTE:

A negative index seems to select everything within the `List`. This seems more like an irregularity than a feature to rely upon.

public synchronized void deselect (int index)

The `deselect()` method deselects the item at position `index`, which is zero based. `deselect()` does not reposition the visible elements.

public boolean isIndexSelected (int index) ★
public boolean isSelected (int index) ☆

The `isIndexSelected()` method checks whether `index` is currently selected. If it is, `isIndexSelected()` returns `true`; otherwise, it returns `false`.

`isSelected()` is the Java 1.0 name for this method.

public boolean isMultipleMode () ★
public boolean allowsMultipleSelections () ☆

The `isMultipleMode()` method returns the current state of the `List`. If the `List` is in multiselection mode, `isMultipleMode()` returns `true`; otherwise, it returns `false`.

`allowsMultipleSelections()` is the Java 1.0 name for this method.

public void setMultipleMode (boolean value) ★
public void setMultipleSelections (boolean value) ☆

The `setMultipleMode()` method allows you to change the current state of a `List` from one selection mode to the other. The currently selected items change when this happens. If `value` is `true` and the `List` is going from single- to multiple-selection mode, the selected item gets deselected. If `value` is `false` and the `List` is going from multiple to single, the last item physically selected remains selected (the last item clicked on in the list, not the item with the highest index). If there was no selected item, the first item in the list becomes selected, or the last item that was deselected becomes selected. If staying within the same mode, `setMultipleMode()` has no effect on the selected items.

`setMultipleSelections()` is the Java 1.0 name for this method.

public void makeVisible (int index)

The `makeVisible()` method ensures that the item at position `index` is displayed on the screen. This is useful if you want to make sure a certain entry is displayed when another action happens on the screen.

public int getVisibleIndex ()

The `getVisibleIndex()` method returns the last index from a call to the method `makeVisible()`. If `makeVisible()` was never called, -1 is returned.

Sizing

public int getRows ()

The `getRows()` method returns the number of rows passed to the constructor of the `List`. It does not return the number of visible rows. To get a rough idea of the number of visible rows, compare the `getSize()` of the component with the results of `getPreferredSize(getRows())`.

public Dimension getPreferredSize (int rows) ★
public Dimension preferredSize (int rows) ☆

The `getPreferredSize()` method returns the preferable `Dimension` (width and height) for the size of a `List` with a height of `rows`. The `rows` specified may be different from the rows designated in the constructor.

`preferredSize()` is the Java 1.0 name for this method.

public Dimension getPreferredSize () ★
public Dimension preferredSize () ☆

The `getPreferredSize()` method returns the `Dimension` (width and height) for the preferred size of the `List`. Without the rows parameter, this version of `getPreferredSize()` uses the constructor's number of rows to calculate the `List`'s preferred size.

`preferredSize()` is the Java 1.0 name for this method.

public Dimension getMiminumSize (int rows) ★
public Dimension minimumSize (int rows) ☆

The `getMinimumSize()` method returns the minimum `Dimension` (width and height) for the size of a `List` with a height of `rows`. The `rows` specified may be different from the rows designated in the constructor. For a `List`, `getMinimumSize()` and `getPreferredSize()` should return the same dimensions.

`minimumSize()` is the Java 1.0 name for this method.

public Dimension getMiminumSize () ★
public Dimension minimumSize () ☆

The `getMinimumSize()` method returns the minimum `Dimension` (width and height) for the size of the `List`. Without the rows parameter, this `getMinimumSize()` uses the constructor's number of rows to calculate the `List`'s minimum size.

`minimumSize()` is the Java 1.0 name for this method.

Miscellaneous methods

public synchronized void addNotify ()

The `addNotify()` method creates the `List` peer. If you override this method, call `super.addNotify()` first, then add your customizations for the new class. You will then be able to do everything you need with the information about the newly created peer.

public synchronized void removeNotify ()

The `removeNotify()` method destroys the peer of the `List` and removes it from the screen. Prior to the `List` peer's destruction, the last selected entry is saved. If you override this method for a specific `List`, issue the particular commands that you need for your new object, then call `super.removeNotify()` last.

protected String paramString ()

When you call the `toString()` method of `List`, the default `toString()` method of `Component` is called. This in turn calls `paramString()`, which builds up the string to display. At the `List` level, the currently selected item (`getSelectedItem()`) is appended to the output. Using Figure 9.2 as an example, the results would be the following:

```
java.awt.List[0,34,107x54,selected=null]
```

List Events

The primary event for a `List` occurs when the user selects an item in the list. With the 1.0 event model, double-clicking a selection causes an `ACTION_EVENT` and triggers the `action()` method, while single-clicking causes a `LIST_SELECT` or `LIST_DESELECT` event. Once the `List` has the input focus, it is possible to change the selection by using the arrow or keyboard keys. The arrow keys scroll through the list of choices, triggering the `KEY_ACTION`, `LIST_SELECT`, `LIST_DESELECT`, and `KEY_ACTION_RELEASE` events, and thus the `keyDown()`, `handleEvent()`, and `keyUp()` methods (no specific method gets called for `LIST_SELECT` and `LIST_DESELECT`). `action()` is called only when the user double-clicks on an item with the mouse. If the mouse is used to scroll through the list, no mouse events are triggered; `ACTION_EVENT` is generated only when the user double-clicks on an item.

With the 1.1 event model, you register an `ItemListener` with `addItemListener()` or an `ActionListener` with the `addActionListener()` method. When the user selects the `List`, either the `ItemListener.itemStateChanged()` method or the `ActionListener.actionPerformed()` method is called through the protected `List.processItemEvent()` method or `List.processActionEvent()` method. Key, mouse, and focus listeners are registered through the three `Component` methods of `addKeyListener()`, `addMouseListener()`, and `addFocusListener()`, respectively. Action

public boolean action (Event e, Object o)

The `action()` method for a `List` is called when the user double-clicks on any item in the `List`. `e` is the `Event` instance for the specific event, while `o` is the label for the item selected, from the `add()` or `addItem()` call. If `List` is in multiple-selection mode, you might not wish to catch this event because it's not clear whether the user wanted to choose the item just selected or all of the items selected. You can solve this problem by putting a multi-selecting list next to a `Button` that the user presses when the selection process is finished. Capture the event generated by the `Button`. The following example shows how to set up and handle a list in this manner, with the display shown in Figure 9.3. In this example, I just print out the selections to prove that I captured them.

```
import java.awt.*;
import java.applet.*;
public class list3 extends Applet {
   List l;
   public void init () {
        String fonts[];
        fonts = Toolkit.getDefaultToolkit().getFontList();
        l = new List(4, true);
        for (int i = 0; i < fonts.length; i++) {
            l.addItem (fonts[i]);
        }
        setLayout (new BorderLayout (10, 10));
        add ("North", new Label ("Pick Font Set"));
        add ("Center", l);
        add ("South", new Button ("Submit"));
        resize (preferredSize());
        validate();
   }
   public boolean action (Event e, Object o) {
        if (e.target instanceof Button) {
             String chosen[] = l.getSelectedItems();
             for (int i=0;i<chosen.length;i++)
                System.out.println (chosen[i]);
        }
        return false;
   }
}
```

Figure 9.3: Multiselect List

[Graphic: Figure 9-3]

Keyboard

Ordinarily, `List` generates all the `KEY` events once it has the input focus. But the way it handles keyboard input differs slightly depending upon the selection mode of the list. Furthermore, each platform offers slightly different behavior, so code that depends on keyboard events in `List` is not portable. One strategy is to take advantage of the keyboard events when they are available but allow for another way of managing the list in case they are not.

public boolean keyDown (Event e, int key)

> The `keyDown()` method is called whenever the user presses a key while the `List` has the input focus. `e` is the `Event` instance for the specific event, while `key` is the integer representation of the character pressed. The identifier for the event (`e.id`) for `keyDown()` could be either `KEY_PRESS` for a regular key or `KEY_ACTION` for an action-oriented key (i.e., arrow or function key). If you check the current selection in this method through `getSelectedItem()` or `getSelectedIndex()`, you will actually be told the previously selected item because the `List`'s selection has not changed yet. `keyDown()` is not called when the user selects items with the mouse.

public boolean keyUp (Event e, int key)

> The `keyUp()` method is called whenever the user releases a key while the `List` has the input focus. `e` is the `Event` instance for the specific event, while `key` is the integer representation of the character pressed. The identifier for the event (`e.id`) for `keyUp()` could be either `KEY_RELEASE` for a regular key or `KEY_ACTION_RELEASE` for an action-oriented key (i.e., arrow or function key).

Mouse

Ordinarily, the `List` component does not trigger any mouse events. Double-clicking the mouse over any element in the list generates an `ACTION_EVENT`. Single-clicking could result in either a

LIST_SELECT or LIST_DESELECT, depending on the mode of the List and the current state of the item chosen. When the user changes the selection with the mouse, the ACTION_EVENT is posted only when an item is double-clicked. List

There is a special pair of events for lists: LIST_SELECT and LIST_DESELECT. No special method is called when these events are triggered. However, you can catch them in the handleEvent() method. If the List is in single-selection mode, a LIST_SELECT event is generated whenever the user selects one of the items in the List. In multiple-selection mode, you will get a LIST_SELECT event when an element gets selected and a LIST_DESELECT event when it is deselected. The following code shows how to use this event type.

```
public boolean handleEvent (Event e) {
    if (e.id == Event.LIST_SELECT) {
        System.out.println ("Selected item: " + e.arg);
        return true;
    } else {
        return super.handleEvent (e);
    }
}
```

Focus

Normally, the List component does not reliably trigger any focus events. Listeners and 1.1 event handling

With the 1.1 event model, you register listeners, and they are told when the event happens.

public void addItemListener(ItemListener listener) ★

> The addItemListener() method registers listener as an object interested in being notified when an ItemEvent passes through the EventQueue with this List as its target. The listener.itemStateChanged() method is called when these events occur. Multiple listeners can be registered.

public void removeItemListener(ItemListener listener) ★

> The removeItemListener() method removes listener as an interested listener. If listener is not registered, nothing happens.

public void addActionListener(ActionListener listener) ★

The `addActionListener()` method registers `listener` as an object interested in being notified when an `ActionEvent` passes through the `EventQueue` with this `List` as its target. The `listener.actionPerformed()` method is called when these events occur. Multiple listeners can be registered.

public void removeActionListener(ActionListener listener) ★

The `removeActionListener()` method removes `listener` as a interested listener. If `listener` is not registered, nothing happens.

protected void processEvent(AWTEvent e) ★

The `processEvent()` method receives all `AWTEvent`s with this `List` as its target. `processEvent()` then passes them along to any listeners for processing. When you subclass `List`, overriding `processEvent()` allows you to process all events yourself, before sending them to any listeners. In a way, overriding the method `processEvent()` is like overriding `handleEvent()` using the 1.0 event model.

If you override `processEvent()`, remember to call `super.processEvent(e)` last to ensure that regular event processing can occur. If you want to process your own events, it's a good idea to call `enableEvents()` (inherited from `Component`) to ensure that events are delivered even in the absence of registered listeners.

protected void processItemEvent(ItemEvent e) ★

The `processItemEvent()` method receives all `ItemEvent`s with this `List` as its target. `processItemEvent()` then passes them along to any listeners for processing. When you subclass `List`, overriding `processItemEvent()` allows you to process all events yourself, before sending them to any listeners. In a way, overriding `processItemEvent()` is like overriding `handleEvent()` to deal with `LIST_SELECT` and `LIST_DESELECT` using the 1.0 event model.

If you override `processItemEvent()`, remember to call the method `super.processItemEvent(e)` last to ensure that regular event processing can occur. If you want to process your own events, it's a good idea to call `enableEvents()` (inherited from `Component`) to ensure that events are delivered even in the absence of registered listeners.

protected void processActionEvent(ActionEvent e) ★

The `processActionEvent()` method receives all `ActionEvent`s with this `List` as its target. `processActionEvent()` then passes them along to any listeners for processing. When you subclass `List`, overriding `processActionEvent()` allows you to process all action events yourself, before sending them to any listeners. In a way, overriding `processActionEvent()` is like overriding `action()` using the 1.0 event model.

If you override `processActionEvent()`, remember to call the method `super.processActionEvent(e)` last to ensure that regular event processing can occur. If you want to process your own events, it's a good idea to call `enableEvents()` (inherited from `Component`) to ensure that events are delivered even in the absence of registered listeners.

9.3 Checkbox

The Checkbox is a general purpose way to record a true or false state. When several checkboxes are associated in a CheckboxGroup (CheckboxGroup), only one can be selected at a time; selecting each Checkbox causes the previous selection to become deselected. The CheckboxGroup is Java's way of offering the interface element known as radio buttons or a radio box. When you create a Checkbox, you decide whether to place it into a CheckboxGroup by setting the proper argument in its constructor.

Every Checkbox has both a label and a state, although the label could be empty. You can change the label based on the state of the Checkbox. Figure 9.4 shows what several Checkbox components might look like. The two on the left are independent, while the five on the right are in a CheckboxGroup. Note that the appearance of a Checkbox varies quite a bit from platform to platform. However, the appearance of a CheckboxGroup is always different from the appearance of an ungrouped Checkbox, and the appearance of a checked Checkbox is different from an unchecked Checkbox.

Figure 9.4: Two separate checkboxes and a CheckboxGroup

[Graphic: Figure 9-4]

Checkbox Methods

Constructors

public Checkbox ()

This constructor for `Checkbox` creates a new instance with no label or grouping. The initial state of the item is `false`. A checkbox doesn't necessarily need a label; however, a checkbox without a label might be confusing, unless it is being used as a column in a table or a spreadsheet.

public Checkbox (String label)

The second constructor creates a new `Checkbox` with a label of `label` and no grouping. The initial state of the item is `false`. If you want a simple yes/no choice and plan to make no the default, use this constructor. If the `Checkbox` will be in a group or you want its initial value to be `true`, use the next constructor.

public Checkbox (String label, boolean state) ★

This constructor allows you to specify the `Checkbox`'s initial state. With it you create a `Checkbox` with a label of `label` and an initial state of `state`.

public Checkbox (String label, boolean state, CheckboxGroup group) ★
public Checkbox (String label, CheckboxGroup group, boolean state)

The final constructor for `Checkbox` is the most flexible. With this constructor you create a `Checkbox` with a label of `label`, a `CheckboxGroup` of `group`, and an initial state of `state`. If `group` is `null`, the `Checkbox` is independent.

In Java 1.0, you created an independent `Checkbox` with an initial value of `true` by using `null` as the group:

```
Checkbox cb = new Checkbox ("Help", null, true)
```

The shape of the `Checkbox` reflects whether it's in a `CheckboxGroup` or independent. On Microsoft Windows, grouped checkboxes are represented as circles. On a UNIX system, they are diamonds. On both systems, independent checkboxes are squares.

Label

public String getLabel ()

The `getLabel()` method retrieves the current label on the `Checkbox` and returns it as a `String` object.

public synchronized void setLabel (String label)

The `setLabel()` method changes the label of the `Checkbox` to `label`. If the new label is a different size than the old one, you have to `validate()` the container after the change to ensure the entire label will be seen.

State

A state of `true` means the `Checkbox` is selected. A state of `false` means that the `Checkbox` is not selected.

public boolean getState ()

The `getState()` method retrieves the current state of the `Checkbox` and returns it as a boolean.

public void setState (boolean state)

The `setState()` method changes the state of the `Checkbox` to `state`. If the `Checkbox` is in a `CheckboxGroup` and `state` is true, the other items in the group become false.

ItemSelectable method

public Objects[] getSelectedObjects () ★

The `getSelectedObjects()` method returns the `Checkbox` label as a one-element `Object` array if it is currently selected, or `null` if the `Checkbox` is not selected. Because this method is part of the `ItemSelectable` interface, you can use it to look at the selected items in a `Choice`, `List`, or `Checkbox`.

CheckboxGroup

This section lists methods that you issue to `Checkbox` to affect its relationship to a `CheckboxGroup`. Methods provided by the `CheckboxGroup` itself can be found later in this chapter.

public CheckboxGroup getCheckboxGroup ()

The `getCheckboxGroup()` method returns the current `CheckboxGroup` for the

Checkbox. If the Checkbox is not in a group, this method returns null.

public void setCheckboxGroup (CheckboxGroup group)

The setCheckboxGroup() method allows you to insert a Checkbox into a different CheckboxGroup. To make the Checkbox independent, pass a group argument of null. The method sets every Checkbox in the original CheckboxGroup to false (cb.getCheckboxGroup().setCurrent(null)), then the Checkbox is added to the new group without changing any values in the new group.

Checkbox components take on a different shape when they are in a CheckboxGroup. If the checkbox was originally not in a CheckboxGroup, the shape of the checkbox does not change automatically when you put it in one with setCheckboxGroup(). (This also holds when you remove a Checkbox from a CheckboxGroup and make it independent or vice versa.) In order for the Checkbox to look right once added to group, you need to destroy and create (removeNotify() and addNotify(), respectively) the Checkbox peer to correct the shape. Also, it is possible to get multiple true Checkbox components in group this way, since the new CheckboxGroup's current selection does not get adjusted. To avoid this problem, make sure it is grouped properly the first time, or be sure to clear the selections with a call to getCheckboxGroup().setCurrent(null).

Miscellaneous methods

public synchronized void addNotify ()

The addNotify() method will create the Checkbox peer in the appropriate shape. If you override this method, call super.addNotify() first, then add your customizations for the new class. You will then be able to do everything you need with the information about the newly created peer.

protected String paramString ()

When you call the toString() method of Checkbox, the default toString() method of Component is called. This in turn calls paramString() which builds up the string to display. At the Checkbox level, the label (if non-null) and the state of the item are appended. Assuming the Dialog Checkbox in Figure 9.4 was selected, the results would be:

```
java.awt.Checkbox[85,34,344x32,label=Dialog,state=true]
```

Checkbox Events

The primary event for a Checkbox occurs when the user selects it. With the 1.0 event model, this generates an ACTION_EVENT and triggers the action() method. Once the Checkbox has the input focus, the various keyboard events can be generated, but they do not serve any useful purpose because the Checkbox doesn't change. The sole key of value for a Checkbox is the spacebar. This may generate the ACTION_EVENT after KEY_PRESS and KEY_RELEASE; thus the sequence of method calls would be keyDown(), keyUp(), and then action().

With the version 1.1 event model, you register an ItemListener with the method addItemListener(). Then when the user selects the Checkbox, the method ItemListener.itemStateChanged() is called through the protected Checkbox.processItemEvent() method. Key, mouse, and focus listeners are registered through the Component methods of addKeyListener(), addMouseListener(), and addFocusListener(), respectively. Action

public boolean action (Event e, Object o)

> The action() method for a Checkbox is called when the user selects it. e is the Event instance for the specific event, while o is the opposite of the old state of the toggle. If the Checkbox was true when it was selected, o will be false. Likewise, if it was false, o will be true. This incantation sounds unnecessarily complex, and for a single Checkbox, it is: o is just the new state of the Checkbox. The following code uses action() with a single Checkbox.
>
> ```
> public boolean action (Event e, Object o) {
>     if (e.target instanceof Checkbox) {
>         System.out.println ("Checkbox is now " + o);
>     }
>     return false;
> }
> ```
>
> On the other hand, if the Checkbox is in a CheckboxGroup, o is still the opposite of the old state of the toggle, which may or may not be the new state of the Checkbox. If the Checkbox is initially false, o will be true, and the Checkbox's new state will be true. However, if the Checkbox is initially true, selecting the Checkbox doesn't change anything because one Checkbox in the group must always be true. In this case, o is false (the opposite of the old state), though the Checkbox's state remains true.
>
> Therefore, if you're working with a CheckboxGroup and need to do something once when the selection changes, perform your action only when o is true. To find out which Checkbox was actually chosen, you need to call the getLabel() method for the target of event e. (It would be nice if o gave us the label of the Checkbox that was selected, but it doesn't.) An example of this follows:

```
public boolean action (Event e, Object o) {
    if (e.target instanceof Checkbox) {
        System.out.println (((Checkbox)(e.target)).getLabel() +
            " was selected.");
        if (new Boolean (o.toString()).booleanValue()) {
            System.out.println ("New option chosen");
        } else {
            System.out.println ("Use re-selected option");
        }
    }
    return false;
}
```

One other unfortunate twist of `CheckboxGroup`: it would be nice if there was some easy way to find out about checkboxes that change state without selection--for example, if you could find out which `Checkbox` was deselected when a new `Checkbox` was selected. Unfortunately, you can't, except by keeping track of the state of all your checkboxes at all times. When a `Checkbox` state becomes `false` because another `Checkbox` was selected, no additional event is generated, in either Java 1.0 or 1.1. Keyboard

`Checkbox`es are able to capture keyboard-related events once the `Checkbox` has the input focus, which happens when it is selected. If you can find a use for this, you can use `keyDown()` and `keyUp()`. For most interface designs I can think of, `action()` is sufficient. A possible use for keyboard events is to jump to other `Checkbox` options in a `CheckboxGroup`, but I think that is more apt to confuse users than help.

public boolean keyDown (Event e, int key)

> The `keyDown()` method is called whenever the user presses a key while the `Checkbox` has the input focus. `e` is the `Event` instance for the specific event, while `key` is the integer representation of the character pressed. The identifier for the event (`e.id`) for `keyDown()` could be either `KEY_PRESS` for a regular key or `KEY_ACTION` for an action-oriented key (i.e., arrow or function key). There is no visible indication that the user has pressed a key over the checkbox.

public boolean keyUp (Event e, int key)

> The `keyUp()` method is called whenever the user releases a key while the `Checkbox` has the input focus. `e` is the `Event` instance for the specific event, while `key` is the integer representation of the character pressed. The identifier for the event (`e.id`) for `keyUp()` could be either `KEY_RELEASE` for a regular key or `KEY_ACTION_RELEASE` for an action-oriented key (i.e., arrow or function key). `keyUp()` may be used to determine how long `key` has been

pressed.

Mouse

Ordinarily, the `Checkbox` component does not reliably trigger any mouse events. Focus

Ordinarily, the `Checkbox` component does not reliably trigger any focus events. Listeners and 1.1 event handling

With the 1.1 event model, you register listeners, and they are told when the event happens.

public void addItemListener(ItemListener listener) ★

The `addItemListener()` method registers `listener` as an object interested in being notified when an `ItemEvent` passes through the `EventQueue` with this `Checkbox` as its target. Then, the `listener.itemStateChanged()` method will be called. Multiple listeners can be registered.

public void removeItemListener(ItemListener listener) ★

The `removeItemListener()` method removes `listener` as a interested listener. If listener is not registered, nothing happens.

protected void processEvent(AWTEvent e) ★

The `processEvent()` method receives every `AWTEvent` with this `Checkbox` as its target. `processEvent()` then passes it along to any listeners for processing. When you subclass `Checkbox`, overriding `processEvent()` allows you to process all events yourself, before sending them to any listeners. In a way, overriding `processEvent()` is like overriding `handleEvent()` using the 1.0 event model.

If you override `processEvent()`, remember to call `super.processEvent(e)` last to ensure that regular event processing can occur. If you want to process your own events, it's a good idea to call `enableEvents()` (inherited from `Component`) to ensure that events are delivered even in the absence of registered listeners.

protected void processItemEvent(ItemEvent e) ★

The `processItemEvent()` method receives every `ItemEvent` with this `Checkbox` as its target. `processItemEvent()` then passes it along to any listeners for processing. When you subclass `Checkbox`, overriding `processItemEvent()` allows you to process all events yourself, before sending them to any listeners. In a way, overriding `processItemEvent()` is like overriding `action()` using the 1.0 event model.

If you override `processItemEvent()`, remember to call the method `super.processItemEvent(e)` last to ensure that regular event processing can occur. If you want to process your own events, it's a good idea to call `enableEvents()` (inherited from `Component`) to ensure that events are delivered even in the absence of registered listeners.

PREVIOUS | HOME | NEXT
Lists | BOOK INDEX | CheckboxGroup

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

9.4 CheckboxGroup

The `CheckboxGroup` lets multiple checkboxes work together to provide a mutually exclusion choice (at most one `Checkbox` can be selected at a time). Because the `CheckboxGroup` is neither a `Component` nor a `Container`, you should normally put all the `Checkbox` components associated with a `CheckboxGroup` in their own `Panel` (or other `Container`). The `LayoutManager` of the `Panel` should be `GridLayout (0, 1)` if you want them in one column. Figure 9.5 shows both a good way and bad way of positioning a set of `Checkbox` items in a `CheckboxGroup`. The image on the left is preferred because the user can sense that the items are grouped; the image on the right suggests three levels of different checkboxes and can therefore surprise the user when checkboxes are deselected.

Figure 9.5: Straightforward and confusing layouts of Checkbox components

[Graphic: Figure 9-5]

CheckboxGroup Methods

Constructors

public CheckboxGroup ()

> This constructor creates an instance of `CheckboxGroup`.

Miscellaneous methods

public int getSelectedCheckbox () ★
public Checkbox getCurrent () ☆

> The `getSelectedCheckbox()` method returns the `Checkbox` within the `CheckboxGroup` whose value is `true`. If no item is selected, `null` is returned.
>
> `getCurrent()` is the Java 1.0 name for this method.

public synchronized void setSelectedCheckbox (Checkbox checkbox) ★
public synchronized void setCurrent (Checkbox checkbox) ☆

> The `setSelectedCheckbox()` method makes `checkbox` the currently selected `Checkbox` within the `CheckboxGroup`. If `checkbox`is `null`, the method deselects all the items in the `CheckboxGroup`. If `checkbox` is not within the `CheckboxGroup`, nothing happens.
>
> `setCurrent()` is the Java 1.0 name for this method.

public String toString ()

> The `toString()` method of `CheckboxGroup` creates a `String` representation of the current choice (as returned by `getSelectedCheckbox()`). Using the "straightforward" layout in Figure 9.5 as an example, the results would be:

```
java.awt.CheckboxGroup[current=java.awt.Checkbox[0,31,85x21,
    label=Helvetica,state=true]]
```

> If there is no currently selected item, the results within the square brackets would be `current=null`.

10.3 MenuShortcut

`MenuShortcut` is a class used to represent a keyboard shortcut for a `MenuItem`. When these events occur, an action event is generated that triggers the menu component. When a shortcut is associated with a `MenuItem`, the `MenuItem` automatically displays a visual clue, which indicates that a keyboard accelerator is available.

MenuShortcut Methods

Constructors

public MenuShortcut (int key) ★

> The first `MenuShortcut` constructor creates a `MenuShortcut` with `key` as its designated hot key. The `key` parameter can be any of the virtual key codes from the `KeyEvent` class (e.g., VK_A, VK_B, etc.). These constants are listed in Table 4.4. To use the shortcut, the user must combine the given `key` with a platform-specific modifier key. On Windows and Motif platforms, the modifier is the Control key; on the Macintosh, it is the Command key. For example, if the shortcut key is F1 (`VK_F1`) and you're using Windows, you would press Ctrl+F1 to execute the shortcut. To find out the platform's modifier key, call the `Toolkit.getMenuShortcutKeyMask()` method.

public MenuShortcut(int key, boolean useShiftModifier) ★

> This `MenuShortcut` constructor creates a `MenuShortcut` with `key` as its designated hot key. If `useShiftModifier` is `true`, the Shift key must be depressed for this shortcut to trigger the action event (in addition to the shortcut key). The `key` parameter represents the integer value of a `KEY_PRESS` event, so in addition to ASCII values, possible values include the various `Event` keyboard constants (listed in Table 4.2) like `Event.F1`, `Event.HOME`, and

Event.PAUSE. For example, if key is the ASCII value for A and useShiftModifier is true, the shortcut key is Shift+Ctrl+A on a Windows/Motif platform.

Miscellaneous methods

public int getKey () ★

The getKey() method retrieves the virtual key code for the key that triggered this MenuShortcut. The virtual key codes are the VK constants defined by the KeyEvent class (see Table 4.4).

public boolean usesShiftModifier() ★

The usesShiftModifier() method returns true if this MenuShortcut requires the Shift key be pressed, false otherwise.

public boolean equals(MenuShortcut s) ★

The equals() method overrides Object's equals() method to define equality for menu shortcuts. Two MenuShortcut objects are equal if their key and useShiftModifier values are equal.

protected String paramString () ★

The paramString() method of MenuShortcut helps build up a string describing the shortcut; it appends the shortcut key and a shift modifier indicator to the string under construction. Oddly, this method is not currently used, nor can you call it; MenuShortcut has its own toString() method that does the job itself.

public String toString() ★

The toString() method of MenuShortcut builds a String to display the contents of the MenuShortcut.

PREVIOUS | HOME | NEXT
MenuContainer | BOOK INDEX | MenuItem

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

10.4 MenuItem

A `MenuItem` is the basic item that goes on a `Menu`. Menus themselves are menu items, allowing submenus to be nested inside of menus. `MenuItem` is a subclass of `MenuComponent`.

MenuItem Methods

Constructors

public MenuItem () ★

> The first `MenuItem` constructor creates a `MenuItem` with an empty label and no keyboard shortcut. To set the label at later time, use `setLabel()`.

public MenuItem (String label)

> This `MenuItem` constructor creates a `MenuItem` with a label of `label` and no keyboard shortcut. A label of "-" represents a separator.

public MenuItem (String label, MenuShortcut shortcut) ★

> The final `MenuItem` constructor creates a `MenuItem` with a label of `label` and a `MenuShortcut` of `shortcut`. Pressing the shortcut key is the same as selecting the menu item.

Menu labels

Each `MenuItem` has a label. This is the text that is displayed on the menu.

NOTE:

Prior to Java 1.1, there was no portable way to associate a hot key with a `MenuItem`. However, in Java 1.0, if you precede a character with an & on a Windows platform, it will appear underlined, and that key will act as the menu's mnemonic key (a different type of shortcut from `MenuShortcut`). Unfortunately, on a Motif platform, the user will see the &. Because the & is part of the label, even if it is not displayed, you must include it explicitly whenever you compare the label to a string.

public String getLabel ()

The `getLabel()` method retrieves the label associated with the `MenuItem`.

public void setLabel (String label)

The `setLabel()` method changes the label of the `MenuItem` to `label`.

Shortcuts

public MenuShortcut getMenuShortcut () ★

The `getMenuShortcut()` method retrieves the shortcut associated with this `MenuItem`.

public void setShortcut (MenuShortcut shortcut) ★

The `setShortcut()` method allows you to change the shortcut associated with a `MenuItem` to `shortcut` after the `MenuItem` has been created.

public void deleteMenuShortcut () ★

The `deleteMenuShortcut()` method removes any associated `MenuShortcut` from the `MenuItem`. If there was no shortcut, nothing happens.

Enabling

public boolean isEnabled ()

The `isEnabled()` method checks to see if the `MenuItem` is currently enabled. An enabled `MenuItem` can be selected by the user. A disabled `MenuItem`, by convention, appears grayed out on the `Menu`. Initially, each `MenuItem` is enabled.

public synchronized void setEnabled(boolean b) ★
public void enable (boolean condition) ☆

> The setEnabled() method either enables or disables the MenuItem based on the value of condition. If condition is true, the MenuItem is enabled. If condition is false, it is disabled. When enabled, the user can select it, generating ACTION_EVENT or notifying the ActionListener. When disabled, the peer does not generate an ACTION_EVENT if the user tries to select the MenuItem. A disabled MenuItem is usually grayed out to signify its state. The way that disabling is signified is platform specific.
>
> enable() is the Java 1.0 name for this method.

public synchronized void enable () ☆

> The enable() method enables the MenuItem. In Java 1.1, it is better to use setEnabled().

public synchronized void disable () ☆

> The disable() method disables the component so that the user cannot select it. In Java 1.1, it is better to use setEnabled().

Miscellaneous methods

public synchronized void addNotify ()

> The addNotify() method creates the MenuItem peer.

public String paramString ()

> The paramString() method of MenuItem should be protected like other paramString() methods. However, it is public so you have access to it. When you call the toString() method of a MenuItem, the default toString() method of MenuComponent is called. This in turn calls paramString() which builds up the string to display. At the MenuItem level, the current label of the object and the shortcut (if present) is appended to the output. If the constructor for the MenuItem was new MenuItem(`File`), the results of toString() would be:

```
java.awt.MenuItem[label=File]
```

MenuItem Events

Event handling

With 1.0 event handing, a `MenuItem` generates an `ACTION_EVENT` when it is selected. The argument to `action()` will be the label of the `MenuItem`. But the target of the `ACTION_EVENT` is the `Frame` containing the menu. You cannot subclass `MenuItem` and catch the `Event` within it with `action()`, but you can with `postEvent()`. No other events are generated for `MenuItem` instances.

public boolean action (Event e, Object o)--overridden by user, called by system

The `action()` method for a `MenuItem` signifies that the user selected it. `e` is the `Event` instance for the specific event, while `o` is the label of the `MenuItem`.

Listeners and 1.1 event handling

With the 1.1 event model, you register listeners, and they are told when the event happens.

public String getActionCommand() ★

The `getActionCommand()` method retrieves the command associated with this `MenuItem`. By default, it is the label. However, the default can be changed by using the `setActionCommand()` method (described next). The command acts like the second parameter to the `action()` method in the 1.0 event model.

public void setActionCommand(String command) ★

The `setActionCommand()` method changes the command associated with a `MenuItem`. When an `ActionEvent` happens, the `command` is part of the event. By default, this would be the label of the `MenuItem`. However, you can change the action command by calling this method. Using action commands is a good idea, particularly if you expect your code to run in a multilingual environment.

public void addActionListener(ItemListener listener) ★

The `addActionListener()` method registers `listener` as an object interested in being notified when an `ActionEvent` passes through the `EventQueue` with this `MenuItem` as its target. The `listener.actionPerformed()` method is called whenever these events occur. Multiple listeners can be registered.

public void removeActionListener(ItemListener listener) ★

The `removeActionListener()` method removes `listener` as an interested listener. If `listener` is not registered, nothing happens.

protected final void enableEvents(long eventsToEnable) ★

Using the `enableEvents()` method is usually not necessary. When you register an action listener, the `MenuItem` listens for action events. However, if you wish to listen for events when listeners are not registered, you must enable the events explicitly by calling this method. The settings for the `eventsToEnable` parameter are found in the `AWTEvent` class; you can use any of the `EVENT_MASK` constants like `COMPONENT_EVENT_MASK`, `MOUSE_EVENT_MASK`, and `WINDOW_EVENT_MASK` ORed together for the events you care about. For instance, to listen for action events, call:

```
enableEvents (AWTEvent.ACTION_EVENT_MASK);
```

protected final void disableEvents(long eventsToDisable) ★

Using the `disableEvents()` method is usually not necessary. When you remove an action listener, the `MenuItem` stops listening for action events if there are no more listeners. However, if you need to, you can disable events explicitly by calling `disableEvents()`. The settings for the `eventsToDisable` parameter are found in the `AWTEvent` class; you can use any of the `EVENT_MASK` constants such as `FOCUS_EVENT_MASK`, `MOUSE_MOTION_EVENT_MASK`, and `ACTION_EVENT_MASK` ORed together for the events you no longer care about.

protected void processEvent(AWTEvent e) ★

The `processEvent()` method receives all `AWTEvent`s with this `MenuItem` as its target. `processEvent()` then passes them along to any listeners for processing. When you subclass `MenuItem`, overriding `processEvent()` allows you to process all events yourself, before sending them to any listeners. In a way, overriding `processEvent()` is like overriding `postEvent()` using the 1.0 event model.

If you override `processEvent()`, remember to call `super.processEvent(e)` last to ensure that regular event processing can occur. If you want to process your own events, it's a good idea to call `enableEvents()` to ensure that events are delivered even in the absence of registered listeners.

protected void processActionEvent(ItemEvent e) ★

The `processActionEvent()` method receives all `ActionEvent`s with this `MenuItem` as its target. `processActionEvent()` then passes them along to any listeners for processing.

When you subclass `MenuItem`, overriding `processActionEvent()` allows you to process all action events yourself, before sending them to any listeners. In a way, overriding `processActionEvent()` is like overriding `action()` using the 1.0 event model.

If you override `processActionEvent()`, remember to call the method `super.processActionEvent(e)` last to ensure that regular event processing can occur. If you want to process your own events, it's a good idea to call `enableEvents()` to ensure that events are delivered even in the absence of registered listeners.

10.5 Menu

`Menu`s are the pull-down objects that appear on the `MenuBar` of a `Frame` or within other menus. They contain `MenuItem`s or `CheckboxMenuItem`s for the user to select. The `Menu` class subclasses `MenuItem` (so it can appear on a `Menu`, too) and implements `MenuContainer`. Tear-off menus are menus that can be dragged, placed elsewhere on the screen, and remain on the screen when the input focus moves to something else. Java supports tear-off menus if the underlying platform does. Motif (UNIX) supports tear-off menus; Microsoft Windows platforms do not.

Menu Methods

Constructors

public Menu () ★

> The first constructor for `Menu` creates a menu that has no label and cannot be torn off. To set the label at a later time, use `setLabel()`.

public Menu (String label)

> This constructor for `Menu` creates a `Menu` with `label` displayed on it. The `Menu` cannot be torn off.

public Menu (String label, boolean tearOff)

> This constructor for `Menu` creates a `Menu` with `label` displayed on it. The handling of `tearOff` is platform dependent.
>
> Figure 10.3 shows a tear-off menu for Windows NT/95 and Motif. Since Windows does not

support tear-off menus, the Windows menu looks and acts like a regular menu.

Figure 10.3: Tear-off menu

[Graphic: Figure 10-3]

Items

public int getItemCount() ★
public int countItems () ☆

The `getItemCount()` method returns the number of items within the `Menu`. Only top-level items are counted: if an item is a submenu, this method doesn't include the items on it.

`countItems()` is the Java 1.0 name for this method.

public MenuItem getItem (int index)

The `getItem()` method returns the `MenuItem` at position `index`. If `index` is invalid, `getItem()` throws the `ArrayIndexOutOfBoundsException` run-time exception.

public synchronized MenuItem add (MenuItem item)

The `add()` method puts `item` on the menu. The label assigned to `item` when it was created is displayed on the menu. If `item` is already in another menu, it is removed from that menu. If `item` is a `Menu`, it creates a submenu. (Remember that `Menu` subclasses `MenuItem`.)

public void add (String label)

This version of `add()` creates a `MenuItem` with `label` as the text and adds that to the menu. If `label` is the `String` "-", a separator bar is added to the `Menu`.

public synchronized void insert(MenuItem item, int index) ★

The `insert()` method puts `item` on the menu at position `index`. The label assigned to `item` when it was created is displayed on the menu. Positions are zero based, and if `index` < 0, `insert()` throws the `IllegalArgumentException` run-time exception.

public synchronized void insert(String label, int index) ★

This version of `insert()` method creates a `MenuItem` with `label` as the text and adds that to the menu at position `index`. If `label` is the String "--", a separator bar is added to the `Menu`. Positions are zero based, and if `index` < 0, this method throws the `IllegalArgumentException` run-time exception.

public void addSeparator ()

The `addSeparator()` method creates a separator `MenuItem` and adds that to the menu. Separator menu items are strictly cosmetic and do not generate events when selected.

public void insertSeparator(int index) ★

The `insertSeparator()` method creates a separator `MenuItem` and adds that to the menu at position `index`. Separator menu items are strictly cosmetic and do not generate events when selected. Positions are zero based. If `index` < 0, `insertSeparator()` throws the `IllegalArgumentException` run-time exception.

public synchronized void remove (int index)

The `remove()` method removes the `MenuItem` at position `index` from the `Menu`. If `index` is invalid, `remove()` throws the `ArrayIndexOutOfBoundsException` run-time exception. `index` is zero based, so it can range from 0 to `getItemCount()-1`.

public synchronized void remove (MenuComponent component)

This version of `remove()` removes the menu item `component` from the `Menu`. If component is not in the `Menu`, nothing happens.

public synchronized void removeAll()

The `removeAll()` removes all `MenuItems` from the `Menu`.

Peers

public synchronized void addNotify ()

> The addNotify() method creates the Menu peer with all the MenuItems on it.

public synchronized void removeNotify ()

> The removeNotify() method destroys the peer of the MenuComponent and removes it from the screen. The peers of the items on the menu are also destroyed.

Miscellaneous methods

public boolean isTearOff ()

> The isTearOff() method returns true if this Menu is a tear-off menu, and false otherwise. Once a menu is created, there is no way to change the tear-off setting. This method can return true even on platforms that do not support tear-off menus.

public String paramString () ★

> The paramString() method of Menu should be protected like other paramString() methods. However, it is public so you have access to it. When you call the toString() method of a Menu, the default toString() method of MenuComponent is called. This in turn calls paramString(), which builds up the string to display. At the Menu level, the setting for TearOff (from constructor) and whether or not it is the help menu (from MenuBar.setHelpMenu()) for the menu bar are added. If the constructor for the Menu was new Menu (`File`), the results of toString() would be:

```
java.awt.Menu [menu0,label=File,tearOff=false,isHelpMenu=false]
```

Menu Events

A Menu does not generate any event when it is selected. An event is generated when a MenuItem on the menu is selected, as long as it is not another Menu. You can capture all the events that happen on a Menu by overriding postEvent().

PREVIOUS
MenuItem
HOME
BOOK INDEX
NEXT
CheckboxMenuItem

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

10.6 CheckboxMenuItem

The `CheckboxMenuItem` is a subclass of `MenuItem` that can be toggled. It is similar to a `Checkbox` but appears on a `Menu`. The appearance depends upon the platform. There may or may not be a visual indicator next to the choice. However, when the `MenuItem` is selected (`true`), a checkmark or some similar graphic will be displayed next to the label.

There is no way to put `CheckboxMenuItem` components into a `CheckboxGroup` to form a radio menu group.

An example of a `CheckboxMenuItem` is the Show Java Console menu item in Netscape Navigator.

CheckboxMenuItem Methods

Constructors

public CheckboxMenuItem (String label)

> The first `CheckboxMenuItem` constructor creates a `CheckboxMenuItem` with no label displayed next to the check toggle. The initial value of the `CheckboxMenuItem` is `false`. To set the label at a later time, use `setLabel()`.

public CheckboxMenuItem (String label)

> The next `CheckboxMenuItem` constructor creates a `CheckboxMenuItem` with `label` displayed next to the check toggle. The initial value of the `CheckboxMenuItem` is `false`.

public CheckboxMenuItem (String label, boolean state)

> The final `CheckboxMenuItem` constructor creates a `CheckboxMenuItem` with `label`

displayed next to the check toggle. The initial value of the `CheckboxMenuItem` is `state`.

Selection

public boolean getState ()

The `getState()` method retrieves the current state of the `CheckboxMenuItem`.

public void setState (boolean condition)

The `setState()` method changes the current state of the `CheckboxMenuItem` to `condition`. When `true`, the `CheckboxMenuItem` will have the toggle checked.

public Object[] getSelectedObjects () ★

The `getSelectedItems()` method returns the currently selected item as an `Object` array. This method, which is required by the `ItemSelectable` interface, allows you to use the same methods to retrieve the selected items of any `Checkbox`, `Choice`, or `List`. The array has at most one element, which contains the label of the selected item; if no item is selected, `getSelectedItems()` returns `null`.

Miscellaneous methods

public synchronized void addNotify ()

The `addNotify()` method creates the `CheckboxMenuItem` peer.

public String paramString ()

The `paramString()` method of `CheckboxMenuItem` should be protected like other `paramString()` methods. However, it is public, so you have access to it. When you call the `toString()` method of a `CheckboxMenuItem`, the default `toString()` method of `MenuComponent` is called. This in turn calls `paramString()` which builds up the string to display. At the `CheckboxMenuItem` level, the current state of the object is appended to the output. If the constructor for the `CheckboxMenuItem` was ``new CheckboxMenuItem(`File`)`` the results would be:

```
java.awt.CheckboxMenuItem[label=File,state=false]
```

CheckboxMenuItem Events

Event handling

A `CheckboxMenuItem` generates an `ACTION_EVENT` when it is selected. The argument to `action()` is the label of the `CheckboxMenuItem`, like the method provided by `MenuItem`, not the state of the `CheckboxMenuItem` as used in `Checkbox`. The target of the `ACTION_EVENT` is the `Frame` containing the menu. You cannot subclass `CheckboxMenuItem` and handle the `Event` within the subclass unless you override `postEvent()`. Listeners and 1.1 event handling

With the Java 1.1 event model, you register listeners, which are told when the event happens.

public void addItemListener(ItemListener listener) ★

The `addItemListener()` method registers `listener` as an object that is interested in being notified when an `ItemEvent` passes through the `EventQueue` with this `CheckboxMenuItem` as its target. When these item events occur, the `listener.itemStateChanged()` method is called. Multiple listeners can be registered.

public void removeItemListener(ItemListener listener) ★

The `removeItemListener()` method removes `listener` as a interested listener. If `listener` is not registered, nothing happens.

protected void processEvent(AWTEvent e) ★

The `processEvent()` method receives every `AWTEvent` with this `CheckboxMenuItem` as its target. `processEvent()` then passes it along to any listeners for processing. When you subclass `CheckboxMenuItem`, overriding `processEvent()` allows you to process all events yourself, before sending them to any listeners. In a way, overriding `processEvent()` is like overriding `postEvent()` using the 1.0 event model.

If you override `processEvent()`, remember to call `super.processEvent(e)` last to ensure that regular event processing can occur. If you want to process your own events, it's a good idea to call `enableEvents()` to ensure that events are delivered, even in the absence of registered listeners.

protected void processItemEvent(ItemEvent e) ★

The `processItemEvent()` method receives every `ItemEvent` with this `CheckboxMenuItem` as its target. `processItemEvent()` then passes it along to any listeners for processing. When you subclass `CheckboxMenuItem`, overriding `processItemEvent()` allows you to process all item events yourself, before sending them to any listeners. In a way, overriding `processItemEvent()` is like overriding `action()` using the 1.0 event model.

If you override `processItemEvent()`, remember to call the method `super.processItemEvent(e)` last to ensure that regular event processing can occur. If you want to process your own events, it's a good idea to call `enableEvents()` to ensure that events are delivered even in the absence of registered listeners.

10.7 MenuBar

The `MenuBar` is the component you add to the `Frame` that is displayed on the top line of the `Frame`; the `MenuBar` contains menus. A `Frame` can display only one `MenuBar` at a time. However, you can change the `MenuBar` based on the state of the program so that different menus can appear at different points. The `MenuBar` class extends `MenuComponent` and implements the `MenuContainer` interface.

A `MenuBar` can be used only as a child component of a `Frame`. An applet cannot have a `MenuBar` attached to it, unless you implement the whole thing yourself. Normally, you cannot modify the `MenuBar` of the applet holder (the browser), unless it is Java based. In other words, you cannot affect the menus of Netscape Navigator, but you can customize *appletviewer* and HotJava, as shown in the following code with the result shown in Figure 10.4. The `getTopLevelParent()` method was introduced in Window with `Window`.

```
import java.awt.*;
public class ChangeMenu extends java.applet.Applet {
    public void init ()  {
        Frame f = ComponentUtilities.getTopLevelParent(this);
        if (f != null) {
            MenuBar mb = f.getMenuBar();
            Menu m = new Menu ("Cool");
            mb.add (m);
        }
    }
}
```

Figure 10.4: Customizing appletviewer's MenuBar

[Graphic: Figure 10-4]

NOTE:

When you add a MenuBar to a Frame, it takes up space that is part of the drawing area. You need to get the top insets to find out how much space is occupied by the MenuBar and be careful not to draw under it. If you do, the MenuBar will cover what you draw.

MenuBar Methods

Constructors

public MenuBar()

The MenuBar constructor creates an empty MenuBar. To add menus to the MenuBar, use the add()method.

Menus

public int getMenuCount () ★
public int countMenus () ☆

The getMenuCount() method returns the number of top-level menus within the MenuBar.

countMenus() is the Java 1.0 name for this method.

public Menu getMenu (int index)

The getMenu() method returns the Menu at position index. If index is invalid, getMenu() throws the run-time exception ArrayIndexOutOfBoundsException.

public synchronized Menu add (Menu m)

The `add()` method puts choice `m` on the `MenuBar`. The label used to create `m` is displayed on the `MenuBar`. If `m` is already in another `MenuBar`, it is removed from it. The order of items added determines the order displayed on the `MenuBar`, with one exception: if a menu is designated as a help menu by `setHelpMenu()`, it is placed at the right end of the menu bar. Only a `Menu` can be added to a `MenuBar`; you can't add a `MenuItem`. In other words, a `MenuItem` has to lie under at least one menu.

public synchronized void remove (int index)

The `remove()` method removes the `Menu` at position `index` from the `MenuBar`. If `index` is invalid, `remove()` throws the `ArrayIndexOutOfBoundsException` run-time exception. `index` is zero based.

public synchronized void remove (MenuComponent component)

This version of `remove()` removes the menu `component` from the `MenuBar`. If `component` is not in `MenuBar`, nothing happens. The system calls this method when you add a new `Menu` to make sure it does not exist on another `MenuBar`.

Shortcuts

public MenuItem getShortcutMenuItem (MenuShortcut shortcut) ★

The `getShortcutMenuItem()` method retrieves the `MenuItem` associated with the `MenuShortcut shortcut`. If `MenuShortcut` does not exist for this `Menu`, the method returns `null`. `getShortcutMenuItem()` walks through the all submenus recursively to try to find `shortcut`.

public synchronized Enumeration shortcuts() ★

The `shortcuts()` method retrieves an `Enumeration` of all the `MenuShortcut` objects associated with this `MenuBar`.

public void deleteShortcut (MenuShortcut shortcut) ★

The `deleteShortcut()` method removes `MenuShortcut` from the associated `MenuItem` in the `MenuBar`. If the shortcut is not associated with any menu item, nothing happens.

Help menus

It is the convention on many platforms to display help menus as the last menu on the `MenuBar`. The

MenuBar class lets you designate one of the menus as this special menu. The physical position of a help menu depends on the platform, but those giving special treatment to help menus place them on the right. A Menu designated as a help menu doesn't have to bear the label "Help"; the label is up to you.

public Menu getHelpMenu ()

> The getHelpMenu() method returns the Menu that has been designated as the help menu with setHelpMenu(). If the menu bar doesn't have a help menu, getHelpMenu() returns null.

public synchronized void setHelpMenu (Menu m)

> The setHelpMenu() method sets the menu bar's help menu to m. This makes m the rightmost menu on the MenuBar, possibly right justified. If m is not already on the MenuBar, nothing happens.

Peers

public synchronized void addNotify ()

> The addNotify() method creates the MenuBar peer with all the menus on it, and in turn their menu items.

public synchronized void removeNotify ()

> The removeNotify() method destroys the peer of the MenuBar and removes it from the screen. The peers of the items on the MenuBar are also destroyed.

MenuBar Events

A MenuBar does not generate any events.

10.8 Putting It All Together

Now that you know about all the different menu classes, it is time to show an example. Example 10.1 contains the code to put up a functional `MenuBar` attached to a `Frame`, using the 1.0 event model. Figure 10.2 (earlier in the chapter) displays the resulting screen. The key parts to examine are how the menus are put together in the `MenuTest` constructor and how their actions are handled within `action()`. I implement one real action in the example: the one that terminates the application when the user chooses Quit. Any other action just displays the label of the item and (if it was a `CheckBoxMenuItem`) the item's state, to give you an idea of how you can use the information returned in the event.

Example 10.1: MenuTest 1.0 Source Code

```
import java.awt.*;
public class MenuTest extends Frame {
    MenuTest () {
        super ("MenuTest");
        MenuItem mi;
        Menu file = new Menu ("File", true);
        file.add ("Open");
        file.add (mi = new MenuItem ("Close"));
        mi.disable();
        Menu extras = new Menu ("Extras", false);
        extras.add (new CheckboxMenuItem ("What"));
        extras.add ("Yo");
        extras.add ("Yo");
        file.add (extras);
        file.addSeparator();
        file.add ("Quit");
        Menu help = new Menu("Help");
        help.add ("About");
        MenuBar mb = new MenuBar();
        mb.add (file);
        mb.add (help);
        mb.setHelpMenu (help);
        setMenuBar (mb);
        resize (200, 200);
    }
    public boolean handleEvent (Event e) {
        if (e.id == Event.WINDOW_DESTROY) {
            System.exit(0);
        }
        return super.handleEvent (e);
    }
    public boolean action (Event e, Object o) {
        if (e.target instanceof MenuItem) {
            if ("Quit".equals (o)) {
                dispose();
                System.exit(1);
            } else {
                System.out.println ("User selected " + o);
                if (e.target instanceof CheckboxMenuItem) {
                    CheckboxMenuItem cb = (CheckboxMenuItem)e.target;
                    System.out.println ("The value is: " + cb.getState());
                }
            }
            return true;
        }
        return false;
    }
    public static void main (String []args) {
        MenuTest f = new MenuTest ();
        f.show();
    }
}
```

The `MenuTest` constructor builds all the menus, creates a menu bar, adds the menus to the menu bar, and adds the menu bar to the `Frame`. To show what is possible, I've included a submenu, a separator bar, a disabled item, and a help menu.

The `handleEvent()` method exists to take care of `WINDOW_DESTROY` events, which are generated if the user uses a native command to exit from the window.

The `action()` method does the work; it received the action events generated whenever the user selects a menu. We ignore most of them, but a real application would need to do more work figuring out the user's selection. As it is, `action()` is fairly simple. If the user selected a menu item, we check to see whether the item's label was "Quit"; if it was, we exit. If the user selected anything else, we print the selection and return `true` to indicate that we handled the event.

Using Java 1.1 Events

Example 10.2 uses the Java 1.1 event model but is otherwise very similar to Example 10.1. Take a close look at the differences and similarities. Although the code that builds the GUI is basically the same in both examples, the event handling is completely different. The helper class `MyMenuItem` is necessary to simplify event handling. In Java 1.1, every menu item can be an event source, so you have to register a listener for each item. Rather than calling `addActionListener()` explicitly for each item, we create a subclass of `MenuItem` that registers a listener automatically. The listener is specified in the constructor to `MyMenuItem`; in this example, the object that creates the menus (`MenuTest12`) always registers itself as the listener. An alternative would be to override `processActionEvent()` in `MyMenuItem`, but then we'd also need to write a subclass for `CheckboxMenuItem`.

Having said all that, the code is relatively simple. `MenuTest12` implements `ActionListener` so it can receive action events from the menus. As I noted previously, it registers itself as the listener for every menu item when it builds the interface. The `actionPerformed()` method is called whenever the user selects a menu item; the logic of this method is virtually the same as it was in Example 10.1. Notice, though, that we use `getActionCommand()` to read the label of the menu item. (Note also that `getActionCommand()` doesn't necessarily return the label; you can change the command associated with the menu item by calling `setActionCommand()`.) Similarly, we call the event's `getSource()` method to get the menu item that actually generated the event; we need this to figure out whether the user selected a `CheckboxMenuItem` (which implements `ItemSelectable`).

We override `processWindowEvent()` so that we can receive `WINDOW_CLOSING` events without registering a listener. Window closings occur when the user uses the native display manager to close the application. If one of these events arrives, we shut down cleanly. To make sure that we receive window events even if there are no listeners, the `MenuTest12` constructor calls `enableEvents(WINDOW_EVENT_MASK)`.

Example 10.2: MenuTest12 Source Code, Using Java 1.1 Event Handling

```
// Java 1.1 only
import java.awt.*;
import java.awt.event.*;
public class MenuTest12 extends Frame implements ActionListener {
    class MyMenuItem extends MenuItem {
        public MyMenuItem (String s, ActionListener al) {
            super (s);
            addActionListener (al);
        }
    }
    public MenuTest12 () {
        super ("MenuTest");
        MenuItem mi;
        Menu file = new Menu ("File", true);
        file.add (new MyMenuItem ("Open", this));
        mi = file.add (new MyMenuItem ("Close", this));
        mi.setEnabled (false);
        Menu extras = new Menu ("Extras", false);
        mi = extras.add (new CheckboxMenuItem ("What"));
        mi.addActionListener(this);
        mi = extras.add (new MyMenuItem ("Yo", this));
        mi.setActionCommand ("Yo1");
        mi = extras.add (new MyMenuItem ("Yo", this));
        mi.setActionCommand ("Yo2");
        file.add (extras);
        file.addSeparator();
        file.add (new MyMenuItem ("Quit", this));
        Menu help = new Menu("Help");
        help.add (new MyMenuItem ("About", this));
        MenuBar mb = new MenuBar();
        mb.add (file);
        mb.add (help);
        mb.setHelpMenu (help);
        setMenuBar (mb);
        setSize (200, 200);
        enableEvents (AWTEvent.WINDOW_EVENT_MASK);
    }
// Cannot override processActionEvent since method of MenuItem
// Would have to subclass both MenuItem and CheckboxMenuItem
    public void actionPerformed(ActionEvent e) {
        if (e.getActionCommand().equals("Quit")) {
            System.exit(0);
        }
        System.out.println ("User selected " + e.getActionCommand());
        if (e.getSource() instanceof ItemSelectable) {
            ItemSelectable is = (ItemSelectable)e.getSource();
            System.out.println ("The value is: " +
                (is.getSelectedObjects().length != 0)));
        }
    }
    protected void processWindowEvent(WindowEvent e) {
        if (e.getID() == WindowEvent.WINDOW_CLOSING) {
            // Notify others we are closing
            super.processWindowEvent(e);
            System.exit(0);
        } else {
            super.processWindowEvent(e);
        }
    }
    public static void main (String []args) {
        MenuTest12 f = new MenuTest12 ();
        f.show();
    }
}
```

I took the opportunity when writing the 1.1 code to make one additional improvement to the program. By using action commands, you can easily differentiate between the two Yo menu items. Just call `setActionCommand()` to assign a different command to each item. (I used "Yo1" and "Yo2".) You could also differentiate between the items by saving a reference to each menu item, calling `getSource()` in the event handler, and comparing the result to the saved references. However, if the `ActionListener` is another class, it would need access to those references. Using action commands is simpler and results in a cleaner event handler.

The intent of the `setActionCommand()` and `getActionCommand()` methods is more for internationalization support. For example, you could use `setActionCommand()` to associate the command Quit with a menu item, then set the item's label to the appropriate text for the user's locality.

PREVIOUS	HOME	NEXT
MenuBar	BOOK INDEX	PopupMenu

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

10.9 PopupMenu

The `PopupMenu` class is new to Java 1.1; it allows you to associate context-sensitive menus with Java components. To associate a pop-up menu with a component, create the menu, and add it to the component using the `add(PopupMenu)` method, which all components inherit from the `Component` class.

In principle, any GUI object can have a pop-up menu. In practice, there are a few exceptions. If the component's peer has its own pop-up menu (i.e., a pop-up menu provided by the run-time platform), that pop-up menu effectively overrides the pop-up menu provided by Java. For example, under Windows NT/95, a `TextArea` has a pop-up menu provided by the Windows NT/95 platforms. Java can't override this menu; although you can add a pop-up menu to a `TextArea`, you can't display that menu under Windows NT/95 with the usual mouse sequence.

PopupMenu Methods

Constructors

public PopupMenu() ★

> The first `PopupMenu` constructor creates an untitled `PopupMenu`. Once created, the menu can be populated with menu items like any other menu.

public PopupMenu(String label) ★

> This constructor creates a `PopupMenu` with a title of `label`. The title appears only on platforms that support titles for context menus. Once created, the menu can be populated with menu items like any other menu.

Miscellaneous methods

public void show(Component origin, int x, int y) ★

Call the show() method to display the PopupMenu. x and y specify the location at which the pop-up menu should appear; origin specifies the Component whose coordinate system is used to locate x and y. In most cases, you'll want the menu to appear at the point where the user clicked the mouse; to do this, set origin to the Component that received the mouse event, and set x and y to the location of the mouse click. It is easy to extract this information from an old-style (1.0) Event or a Java 1.1 MouseEvent. In Java 1.1, the platform-independent way to say "give me the mouse events that are supposed to trigger pop-up menus" is to call MouseEvent.isPopupTrigger(). If this method returns true, you should show the pop-up menu if one is associated with the event source. (Note that the mouse event could also be used for some other purpose.)

If the PopupMenu is not associated with a Component, show() throws the run-time exception NullPointerException. If origin is not the MenuContainer for the PopupMenu and origin is not within the Container that the pop-up menu belongs to, show() throws the run-time exception IllegalArgumentException. Finally, if the Container of origin does not exist or is not showing, show() throws a run-time exception.

public synchronized void addNotify () ★

The addNotify() method creates the PopupMenu peer with all the MenuItems on it.

Example 10.3 is a simple applet that raises a pop-up menu if the user clicks the appropriate mouse button anywhere within the applet. Although the program could use the 1.0 event model, under the 1.0 model, it is impossible to tell which mouse event is appropriate to display the pop-up menu.

Example 10.3: Using a PopupMenu

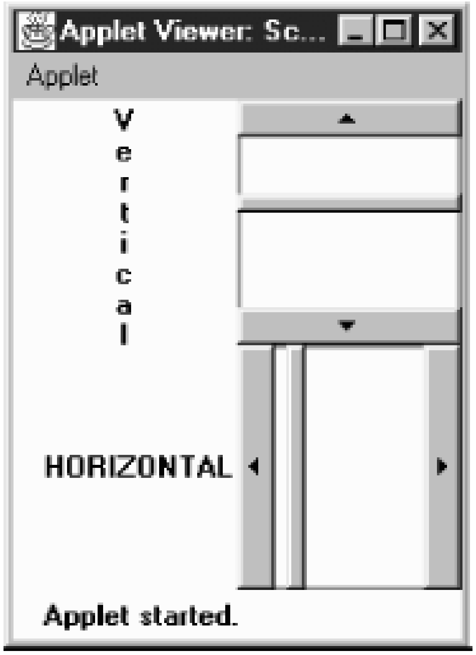
```
// Java 1.1 only
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
public class PopupTest extends Applet implements ActionListener {
    PopupMenu popup;
    public void init() {
        MenuItem mi;
        popup = new PopupMenu("Title Goes Here");
        popup.add(mi = new MenuItem ("Undo"));
        mi.addActionListener (this);
        popup.addSeparator();
        popup.add(mi = new MenuItem("Cut")).setEnabled(false);
        mi.addActionListener (this);
        popup.add(mi = new MenuItem("Copy")).setEnabled(false);
        mi.addActionListener (this);
        popup.add(mi = new MenuItem ("Paste"));
        mi.addActionListener (this);
        popup.add(mi = new MenuItem("Delete")).setEnabled(false);
        mi.addActionListener (this);
        popup.addSeparator();
        popup.add(mi = new MenuItem ("Select All"));
        mi.addActionListener (this);
        add (popup);
        resize(200, 200);
        enableEvents (AWTEvent.MOUSE_EVENT_MASK);
    }
    protected void processMouseEvent (MouseEvent e) {
        if (e.isPopupTrigger())
            popup.show(e.getComponent(), e.getX(), e.getY());
        super.processMouseEvent (e);
    }
    public void actionPerformed(ActionEvent e) {
        System.out.println (e);
    }
}
```

11.2 Scrolling An Image

Example 11.1 is a Java application that displays any image in the current directory in a viewing area. The viewing area scrolls to accommodate larger images; the user can use the scrollbars or keypad keys to scroll the image. In Java 1.1, it is trivial to implement this example with a `ScrollPane`; however, if you're using 1.0, you don't have this luxury. Even if you're using 1.1, this example shows a lot about how to use scrollbars.

Our application uses a `Dialog` to select which file to display; a `FilenameFilter` limits the list to image files. We use a menu to let the user request a file list or exit the program. After the user picks a file, the application loads it into the display area. Figure 11.3 shows the main scrolling window.

Figure 11.3: Scrolling an image



Applet started.

The code for the scrolling image consists of a `ScrollingImage` class, plus several helper classes. It places everything into the file *ScrollingImage.java* for compilation.

Example 11.1: Source Code for Scrolling an Image

```
import java.awt.*;
import java.io.FilenameFilter;
import java.io.File;
```

The first class contains the `FilenameFilter` used to select relevant filenames: that is, files that are likely to contain GIF, JPEG, or XBM images. If the filename has an appropriate ending, the `accept()` method returns `true`; otherwise, it returns `false`.

```
// True for files ending in jpeg/jpg/gif/xbm
class ImageFileFilter implements FilenameFilter {
    public boolean accept (File dir, String name) {
        String tempname = name.toLowerCase();
        return (tempname.endsWith ("jpg") || tempname.endsWith ("jpeg") ||
                tempname.endsWith ("gif") || tempname.endsWith ("xbm"));
    }
}
```

The `ImageListDialog` class displays a list of files from which the user can select. Instead of using `FileDialog`, we created a customized `List` to prevent the user from roaming around the entire hard drive; choices are limited to appropriate files in the current directory. When the user selects an entry (by double-clicking), the image is then displayed in the scrolling area.

```
class ImageListDialog extends Dialog {
    private String name = null;
    private String entries[];
    private List list;
    ImageListDialog (Frame f) {
        super (f, "Image List", true);
        File dir = new File (System.getProperty("user.dir"));
        entries = dir.list (new ImageFileFilter());
        list = new List (10, false);
        for (int i=0;i<entries.length;i++) {
            list.addItem (entries[i]);
        }
        add ("Center", list);
        pack();
    }
    public String getName () {
        return name;
    }
    public boolean action (Event e, Object o) {
        name = (String)e.arg;
        ((ScrollingImage)getParent()).processImage();
        dispose();
        return true;
    }
}
```

The code in this class is straightforward. The constructor reads the current directory from the system property list, uses the `list()` method of the `File` class to create a list of files that match our filename filter, and then creates a `List` object that lists these files. `getName()` returns the name of the selected file. `action()` is called when the user selects a file; it sets the name of the selected file from the `arg` field of the `Event` and then calls the `processImage()` method of its parent container. The parent container of our `ImageListDialog` is the `ScrollingImage` class we are defining; its `processImage()` method displays a scrollable image.

The next class, `ImageCanvas`, is the `Canvas` that the image is drawn onto. We use a separate `Canvas` rather than drawing directly onto the `Frame` so that the scrollbars do not overlap the edges of the image. You will notice that the `paint()` method uses negative `x` and `y` values. This starts the drawing outside the `Canvas` area; as a result, the `Canvas` displays only part of the image. This is how we do the actual scrolling. (`xPos`, `yPos`) are the values given to us from the scrollbars; by positioning the image at (`-xPos`, `-yPos`), we ensure that the point (`xPos`, `yPos`) appears in the upper left corner of the canvas.

```
class ImageCanvas extends Canvas {
    Image image;
    int xPos, yPos;
    public void redraw (int xPos, int yPos, Image image) {
        this.xPos = xPos;
        this.yPos = yPos;
        this.image = image;
        repaint();
    }
    public void paint (Graphics g) {
        if (image != null)
            g.drawImage (image, -xPos, -yPos, this);
    }
}
```

`ScrollingImage` provides the framework for the rest of the program. It provides a menu to bring up the `Dialog` to choose the file, the scrollbars to scroll the scrolling canvas, and the image canvas area. This class also implements event handling methods to capture the scrollbar events, paging keys, and menu events.

```
public class ScrollingImage extends Frame {
    static Scrollbar horizontal, vertical;
    ImageCanvas center;
    int xPos, yPos;
    Image image;
    ImageListDialog ild;
    ScrollingImage () {
        super ("Image Viewer");
        add ("Center", center = new ImageCanvas ());
        add ("South",  horizontal = new Scrollbar (Scrollbar.HORIZONTAL));
        add ("East", vertical = new Scrollbar (Scrollbar.VERTICAL));
        Menu m = new Menu ("File", true);
        m.add ("Open");
        m.add ("Close");
        m.add ("-");
        m.add ("Quit");
        MenuBar mb = new MenuBar();
        mb.add (m);
        setMenuBar (mb);
        resize (400, 300);
    }
    public static void main (String args[]) {
        ScrollingImage si = new ScrollingImage ();
        si.show();
    }
    public boolean handleEvent (Event e) {
         if (e.id == Event.WINDOW_DESTROY) {
            System.exit(0);
        } else if (e.target instanceof Scrollbar) {
            if (e.target == horizontal) {
                xPos = ((Integer)e.arg).intValue();
            } else if (e.target == vertical) {
                yPos = ((Integer)e.arg).intValue();
            }
            center.redraw (xPos, yPos, image);
        }
        return super.handleEvent (e);
    }
```

This `handleEvent()` method kills the program if the user used the windowing system to exit from it (`WINDOW_DESTROY`). More to the point, if a `Scrollbar` generated the event, `handleEvent()` figures out if it was the horizontal or vertical scrollbar, saves its value as the `x` or `y` position, and redraws the image in the new location. Finally, it calls `super.handleEvent()`; as we will see in the following code, other events that we care about (key events and menu events) we don't want to handle here--we would rather handle them normally, in `action()` and `keyDown()` methods.

```
    public void processImage () {
        image = getToolkit().getImage (ild.getName());
        MediaTracker tracker = new MediaTracker (this);
        tracker.addImage (image, 0);
        try {
            tracker.waitForAll();
        } catch (InterruptedException ie) {
        }
        xPos = 0;
        yPos = 0;
        int imageHeight = image.getHeight (this);
        int imageWidth = image.getWidth (this);
        vertical.setValues (0, 5, 0, imageHeight);
        horizontal.setValues (0, 5, 0, imageWidth);
        center.redraw (xPos, yPos, image);
    }
```

`processImage()` reads the image's filename, calls `getImage()`, and sets up a `MediaTracker` to wait for the image to load. Once the image has loaded, it reads the height and width, and uses these to set the maximum values for the vertical and horizontal scrollbars by calling `setValues()`. The scrollbars' minimum and initial values are both 0. The size of the scrollbar "handle" is set to 5, rather than trying to indicate the visible portion of the image.

```
    public boolean action (Event e, Object o) {
        if (e.target instanceof MenuItem) {
            if ("Open".equals (o)) {
                // If showing already, do not show again
                if ((ild == null) || (!ild.isShowing())) {
                    ild = new ImageListDialog (this);
                    ild.show();
                }
            } else if ("Close".equals(o)) {
                image = null;
                center.redraw (xPos, yPos, image);
            } else if ("Quit".equals(o)) {
                System.exit(0);
            }
            return true;
        }
        return false;
    }
```

`action()` handles menu events. If the user selected Open, it displays the dialog box that selects a file. Close redraws the canvas with a null image; Quit exits the program. If any of these events occurred, `action()` returns `true`, indicating that the event was fully handled. If any other event occurred, `action()` returns `false`, so that the system will deliver the event to any other methods that might be interested in it.

```
    public boolean keyDown (Event e, int key) {
        if (e.id == Event.KEY_ACTION) {
            Scrollbar target = null;
            switch (key) {
```

```
            case Event.HOME:
                target = vertical;
                vertical.setValue(vertical.getMinimum());
                break;
            case Event.END:
                target = vertical;
                vertical.setValue(vertical.getMaximum());
                break;
            case Event.PGUP:
                target = vertical;
                vertical.setValue(vertical.getValue()
                         - vertical.getPageIncrement());
                break;
            case Event.PGDN:
                target = vertical;
                vertical.setValue(vertical.getValue()
                         + vertical.getPageIncrement());
                break;
            case Event.UP:
                target = vertical;
                vertical.setValue(vertical.getValue()
                         - vertical.getLineIncrement());
                break;
            case Event.DOWN:
                target = vertical;
                vertical.setValue(vertical.getValue()
                         + vertical.getLineIncrement());
                break;
            case Event.LEFT:
                target = horizontal;
                if (e.controlDown())
                    horizontal.setValue(horizontal.getValue() -
                        horizontal.getPageIncrement());
                else
                    horizontal.setValue(horizontal.getValue() -
                        horizontal.getLineIncrement());
                break;
            case Event.RIGHT:
                target = horizontal;
                if (e.controlDown())
                    horizontal.setValue(horizontal.getValue() +
                        horizontal.getPageIncrement());
                else
                    horizontal.setValue(horizontal.getValue() +
                        horizontal.getLineIncrement());
                    break;
                default:
                    return false;
            }
            Integer value = new Integer (target.getValue());
            postEvent (new Event ((Object)target,
                       Event.SCROLL_ABSOLUTE, (Object)value));
            return true;
        }
        return false;
    }
}
```

`keyDown()` isn't really necessary, but it adds a nice extension to our scrollbars: in addition to using the mouse, the user can scroll with the arrow keys. Pressing an arrow key generates a `KEY_ACTION` event. If we have one of these events, we check what kind of key it was, then compute a new scrollbar value, then call `setValue()` to set the appropriate scrollbar to this value. For example, if the user presses the page up key, we read the page increment, add it to the current value of the vertical scrollbar, and then set the vertical scrollbar accordingly. (Note that this works even though nondefault page and line increments aren't implemented correctly.) The one trick here is that we have to get the rest of the program to realize that the scrollbar values have changed. To do so, we create a new `SCROLL_ABSOLUTE` event, and call `postEvent()` to deliver it.

PREVIOUS | HOME | NEXT
Scrollbar | BOOK INDEX | The Adjustable Interface

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

11.3 The Adjustable Interface

The `Adjustable` interface is new to Java 1.1. It provides the method signatures required for an object that lets you adjust a bounded integer value. It is currently implemented by `Scrollbar` and returned by two methods within `ScrollPane`.

Constants of the Adjustable Interface

There are two direction specifiers for `Adjustable`.

public final static int HORIZONTAL ★

> `HORIZONTAL` is the constant for horizontal orientation.

public final static int VERTICAL ★

> `VERTICAL` is the constant for vertical orientation.

Methods of the Adjustable Interface

public abstract int getOrientation () ★

> The `getOrientation()` method is for returning the current orientation of the adjustable object, either `Adjustable.HORIZONTAL` or `Adjustable.VERTICAL`.
>
> `setOrientation()` is not part of the interface. Not all adjustable objects need to be able to alter orientation. For example, `Scrollbar` instances can change their orientation, but each `Adjustable` instance associated with a `ScrollPane` has a fixed, unchangeable orientation.

public abstract int getVisibleAmount () ★

The `getVisibleAmount()` method lets you retrieve the size of the visible slider of the adjustable object.

public abstract void setVisibleAmount (int amount) ★

The `setVisibleAmount()` method lets you change the size of the visible slider to `amount`.

public abstract int getValue () ★

The `getValue()` method lets you retrieve the current value of the adjustable object.

public abstract void setValue (int value) ★

The `setValue()` method lets you change the value of the adjustable object to `value`.

public abstract int getMinimum ()

The `getMinimum()` method lets you retrieve the current minimum setting for the object.

public abstract void setMinimum (int minimum) ★

The `setMinimum()` method lets you change the minimum value of the adjustable object to `minimum`.

public abstract int getMaximum () ★

The `getMaximum()` method lets you retrieve the current maximum setting for the object.

public abstract void setMaximum (int maximum) ★

The `setMaximum()` method lets you change the maximum value of the adjustable object to `maximum`.

public abstract int getUnitIncrement () ★

The `getUnitIncrement()` method lets you retrieve the current line increment.

public abstract void setUnitIncrement (int amount) ★

The `setUnitIncrement()` method lets you change the line increment amount of the adjustable object to `amount`.

public abstract int getBlockIncrement () ★

The `getBlockIncrement()` method lets you retrieve the current page increment.

public abstract void setBlockIncrement (int amount) ★

The `setBlockIncrement()` method lets you change the paging increment amount of the adjustable object to `amount`.

public abstract void addAdjustmentListener(AdjustmentListener listener) ★

The `addAdjustmentListener()` method lets you register `listener` as an object interested in being notified when an `AdjustmentEvent` passes through the `EventQueue` with this `Adjustable` object as its target.

public abstract void removeAdjustmentListener(ItemListener listener) ★

The `removeAdjustmentListener()` method removes `listener` as a interested listener. If `listener` is not registered, nothing happens.

11.4 ScrollPane

A ScrollPane is a Container with built-in scrollbars that can be used to scroll its contents. In the current implementation, a ScrollPane can hold only one Component and has no layout manager. The component within a ScrollPane is always given its preferred size. While the scrollpane's inability to hold multiple components sounds like a deficiency, it isn't; there's no reason you can't put a Panel inside a ScrollPane, put as many components as you like inside the Panel, and give the Panel any layout manager you wish.

Scrolling is handled by the ScrollPane peer, so processing is extremely fast. In Example 11.1, the user moves a Scrollbar to trigger a scrolling event, and the peer sends the event to the Java program to find someone to deal with it. Once it identifies the target, it posts the event, then tries to find a handler. Eventually, the applet's handleEvent() method is called to reposition the ImageCanvas. The new position is then given to the peer, which finally redisplays the Canvas. Although most of the real work is behind the scenes, it is still happening. With ScrollPane, the peer generates and handles the event itself, which is much more efficient.

ScrollPane Methods

Constants

The ScrollPane class contains three constants that can be used to control its scrollbar display policy. The constants are fairly self-explanatory. The constants are used in the constructor for a ScrollPane instance.

public static final int SCROLLBARS_AS_NEEDED ★

> SCROLLBARS_AS_NEEDED is the default scrollbar display policy. With this policy, the ScrollPane displays each scrollbar only if the Component is too large in the scrollbar's direction.

public static final int SCROLLBARS_ ALWAYS ★

> With the SCROLLBARS_ALWAYS display policy, the ScrollPane should always display both scrollbars, whether or not they are needed.

public static final int SCROLLBARS_ NEVER ★

> With the SCROLLBARS_NEVER display policy, the ScrollPane should never display scrollbars, even when the object is bigger than the ScrollPane's area. When using this mode, you should provide some means for the

user to scroll, either through a button outside the container or by listening for events happening within the container.

Constructors

public ScrollPane () ★

The first constructor creates an instance of `ScrollPane` with the default scrollbar display policy setting, `SCROLLBARS_AS_NEEDED`.

public ScrollPane (int scrollbarDisplayPolicy) ★

The other constructor creates an instance of `ScrollPane` with a scrollbar setting of `scrollbarDisplayPolicy`. If `scrollbarDisplayPolicy` is not one of the class constants, this constructor throws the `IllegalArgumentException` run-time exception.

Layout methods

public final void setLayout(LayoutManager mgr) ★

The `setLayout()` method of `ScrollPane` throws an `AWTError`. It overrides the `setLayout()` method of `Container` to prevent you from changing a `ScrollPane`'s layout manager.

public void doLayout () ★
public void layout () ☆

The `doLayout()` method of `ScrollPane` shapes the contained object to its preferred size.

`layout()` is another name for this method.

public final void addImpl(Component comp, Object constraints, int index) ★

The `addImpl()` method of `ScrollPane` permits only one object to be added to the `ScrollPane`. It overides the `addImpl()` method of `Container` to enforce the `ScrollPane`'s limitations on adding components. If `index > 0`, `addImpl()` throws the run-time exception `IllegalArgumentException`. If a component is already within the `ScrollPane`, it is removed before `comp` is added. The `constraints` parameter is ignored.

Scrolling methods

public int getScrollbarDisplayPolicy() ★

The `getScrollbarDisplayPolicy()` method retrieves the current display policy, as set by the constructor. You cannot change the policy once it has been set. The return value is one of the class constants: `SCROLLBARS_AS_NEEDED`, `SCROLLBARS_ALWAYS`, or `SCROLLBARS_NEVER`.

public Dimension getViewportSize() ★

The `getViewportSize()` method returns the current size of the `ScrollPane`, less any `Insets`, as a `Dimension` object. The size is given in pixels and has an initial value of 100 x 100.

public int getHScrollbarHeight() ★

The `getHScrollbarHeight()` method retrieves the height in pixels of a horizontal scrollbar. The value returned is without regard to the display policy; that is, you may be given a height even if the scrollbar is not displayed. This method may return 0 if the scrollbar's height cannot be calculated at this time (no peer) or if you are using the `SCROLLBARS_NEVER` display policy.

The width of a horizontal scrollbar is just `getViewportSize().width`.

public int getVScrollbarWidth() ★

The `getVScrollbarWidth()` method retrieves the width in pixels of a vertical scrollbar. The value returned is without regard to the display policy; that is, you may be given a width even if the scrollbar is not displayed. This method may return 0 if the scrollbar's width cannot be calculated at this time (no peer) or if you are using the `SCROLLBARS_NEVER` display policy.

The height of a vertical scrollbar is just `getViewportSize().height`.

public Adjustable getHAdjustable() ★

The `getHAdjustable()` method returns the adjustable object representing the horizontal scrollbar (or `null` if it is not present). Through the methods of `Adjustable`, you can get the different settings of the scrollbar.

The object that this method returns is an instance of the package private class `ScrollPaneAdjustable`, which implements the `Adjustable` interface. this class allows you to register listeners for the scrollpane's events and inquire about various properties of the pane's scrollbars. It does not let you set some scrollbar properties; the `setMinimum()`, `setMaximum()`, and `setVisibleAmount()` methods throw an `AWTError` when called.

public Adjustable getVAdjustable() ★

The `getVAdjustable()` method returns the adjustable object representing the vertical scrollbar (or `null` if it is not present). Through the methods of `Adjustable`, you can get the different settings of the scrollbar.

The object that this method returns is an instance of the package private class `ScrollPaneAdjustable`, which implements the `Adjustable` interface. this class allows you to register listeners for the scrollpane's events and inquire about various properties of the pane's scrollbars. It does not let you set some scrollbar properties; the `setMinimum()`, `setMaximum()`, and `setVisibleAmount()` methods throw an `AWTError` when called.

public void setScrollPosition(int x, int y) ★

This `setScrollPosition()` method moves the `ScrollPane` to the designated location if possible. The `x` and `y` arguments are scrollbar settings, which should be interpreted in terms of the minimum and maximum values given to you by the horizontal and vertical `Adjustable` objects (returned by the previous two methods). If the `ScrollPane` does not have a child component, this method throws the run-time exception `NullPointerException`. You can also move the `ScrollPane` by calling the `Adjustable.setValue()` method of one of the scrollpane's `Adjustable` objects.

public void setScrollPosition(Point p) ★

This `setScrollPosition()` method calls the previous with parameters of `p.x`, and `p.y`.

public Point getScrollPosition() ★

The `getScrollPosition()` method returns the current position of both the scrollpane's `Adjustable` objects as a `Point`. If there is no component within the `ScrollPane`, `getScrollPosition()` throws the `NullPointerException` run-time exception. Another way to get this information is by calling the `Adjustable.getValue()` method of each `Adjustable` object.

Miscellaneous methods

public void printComponents (Graphics g) ★

The `printComponents()` method of `ScrollPane` prints the single component it contains. This is done by clipping the context `g` to the size of the display area and calling the contained component's `printAll()` method.

public synchronized void addNotify () ★

The `addNotify()` method creates the `ScrollPane` peer. If you override this method, call `super.addNotify()` first, then add your customizations for the new class. You will then be able to do everything you need with the information about the newly created peer.

protected String paramString () ★

`ScrollPane` doesn't have its own `toString()` method; so when you call the `toString()` method of a `ScrollPane`, you are actually calling the `Component.toString()` method. This in turn calls `paramString()`, which builds the string to display. For a `ScrollPane`, `paramString()` adds the current scroll position, insets, and scrollbar display policy. For example:

```
java.awt.ScrollPane[scrollpane0,0,0,0x0,invalid,ScrollPosition=(0,0),
        Insets=(0,0,0,0),ScrollbarDisplayPolicy=always]
```

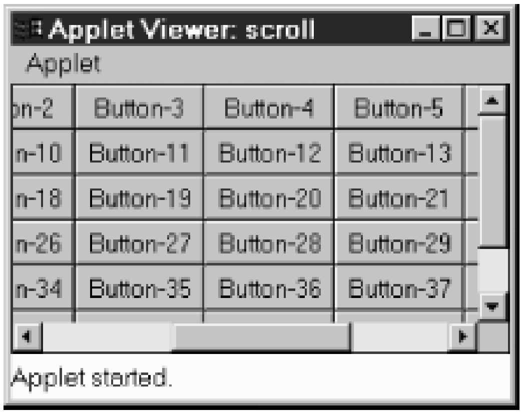
ScrollPane Events

The `ScrollPane` peer deals with the scrolling events for you. It is not necessary to catch or listen for these events. As with any other `Container`, you can handle the 1.0 events of the object you contain or listen for 1.1 events that happen within you.

Using a ScrollPane

The following applet demonstrates one way to use a `ScrollPane`. Basically, you place the object you want to scroll in the `ScrollPane` by calling the `add()` method. This can be a `Panel` with many objects on it or a `Canvas` with an image drawn on it. You then add as many objects as you want to the `Panel` or scribble on the `Canvas` to your heart's delight. No scrolling event handling is necessary. That is all there is to it. To make this example a little more interesting, whenever you select a button, the `ScrollPane` scrolls to a randomly selected position. Figure 11.4 displays the screen.

Figure 11.4: A ScrollPane containing many buttons



Here's the code:

```
// Java 1.1 only
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class scroll extends Applet implements ActionListener, ContainerListener {
    ScrollPane sp = new ScrollPane (ScrollPane.SCROLLBARS_ALWAYS);
    public void init () {
        setLayout (new BorderLayout ());
        Panel p = new Panel(new GridLayout (7, 8));
        p.addContainerListener (this);
        for (int j=0;j<50;j++)
            p.add (new Button ("Button-" + j));
        sp.add (p);
        add (sp, "Center");
    }
    public void componentAdded(ContainerEvent e) {
        if (e.getID() == ContainerEvent.COMPONENT_ADDED) {
            if (e.getChild() instanceof Button) {
                Button b = (Button)e.getChild();
                b.addActionListener(this);
            }
        }
    }
    public void componentRemoved(ContainerEvent e) {
    }
    public void actionPerformed (ActionEvent e) {
        Component c = sp.getComponent();
            Dimension d = c.getSize();
            sp.setScrollPosition ((int)(Math.random()*d.width),
                (int)(Math.random()*d.height));
    }
}
```

Working with the `ScrollPane` itself is easy; we just create one, add a `Panel` to it, set the `Panel`'s layout manager to `GridLayout`, and add a lot of buttons to the `Panel`. The applet itself is the action listener for all the buttons; when anybody clicks a button, `actionPerformed()` is called, which generates a new random position based on the viewport size and sets the new scrolling position accordingly by calling `setScrollPosition()`.

The more interesting part of this applet is the way it works with buttons. Instead of directly adding a listener for each button, we add a `ContainerListener` to the containing panel and let it add listeners. Although this may seem like extra work here, it demonstrates how you can use container events to take actions whenever someone adds or removes a component. At first glance, you might ask why I didn't just call `enableEvents(AWTEvent.CONTAINER_EVENT_MASK)` and override the applet's `processContainerEvent()` to attach the listeners. If we were only adding our components to the applet, that would work great. Unfortunately, the applet is not notified when buttons are added to an unrelated panel. It would be notified only when the panel was added to the applet.

PREVIOUS | HOME | NEXT
The Adjustable Interface | BOOK INDEX | Image Processing

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

12.2 ColorModel

A color model determines how colors are represented within AWT. `ColorModel` is an abstract class that you can subclass to specify your own representation for colors. AWT provides two concrete subclasses of `ColorModel` that you can use to build your own color model; they are `DirectColorModel` and `IndexColorModel`. These two correspond to the two ways computers represent colors internally.

Most modern computer systems use 24 bits to represent each pixel. These 24 bits contain 8 bits for each primary color (red, green, blue); each set of 8 bits represents the intensity of that color for the particular pixel. This arrangement yields the familiar "16 million colors" that you see in advertisements. It corresponds closely to Java's direct color model.

However, 24 bits per pixel, with something like a million pixels on the screen, adds up to a lot of memory. In the dark ages, memory was expensive, and devoting this much memory to a screen buffer cost too much. Therefore, designers used fewer bits--possibly as few as three, but more often eight--for each pixel. Instead of representing the colors directly in these bits, the bits were an index into a color map. Graphics programs would load the color map with the colors they were interested in and then represent each pixel by using the index of the appropriate color in the map. For example, the value 1 might represent fuschia; the value 2 might represent puce. Full information about how to display each color (the red, green, and blue components that make up fuschia or puce) is contained only in the color map. This arrangement corresponds closely to Java's indexed color model.

Because Java is platform-independent, you don't need to worry about how your computer or the user's computer represents colors. Your programs can use an indexed or direct color map as appropriate. Java will do the best it can to render the colors you request. Of course, if you use 5,000 colors on a computer that can only display 256, Java is going to have to make compromises. It will decide which colors to put in the color map and which colors are close enough to the colors in the color map, but that's done behind your back.

Java's default color model uses 8 bits per pixel for red, green, and blue, along with another 8 bits for alpha (transparency) level. However, as I said earlier, you can create your own `ColorModel` if you want to work in some other scheme. For example, you could create a grayscale color model for black and white pictures, or an HSB (hue, saturation, brightness) color model if you are more comfortable working with this system. Your color model's job will be to take a pixel value in your representation and translate that value into the corresponding alpha, red, green, and blue values. If you are working with a grayscale image, your image producer could deliver grayscale values to the image consumer, plus a `ColorModel` that tells the consumer how to render these gray values in terms of ARGB components.

ColorModel Methods

Constructors

public ColorModel (int bits)

> There is a single constructor for `ColorModel`. It has one parameter, `bits`, which describes the number of bits required per pixel of an image. Since this is an abstract class, you cannot call this constructor directly. Since each pixel value must be stored within an integer, the maximum value for `bits` is 32. If you request more, you get 32.

Pseudo-constructors

public static ColorModel getRGBdefault()

> The `getRGBdefault()` method returns the default `ColorModel,` which has 8 bits for each of the components alpha, red, green, and blue. The order the pixels are stored in an integer is 0xAARRGGBB, or alpha in highest order byte, down to blue in the lowest.

Other methods

public int getPixelSize ()

> The `getPixelSize()` method returns the number of bits required for each pixel as described by this color model. That is, it returns the number of bits passed to the constructor.

public abstract int getAlpha (int pixel)

> The `getAlpha()` method returns the alpha component of `pixel` for a color model. Its range must be between 0 and 255, inclusive. A value of 0 means the pixel is completely transparent and the background will appear through the pixel. A value of 255 means the pixel is opaque and you cannot see the background behind it.

public abstract int getRed (int pixel)

> The `getRed()` method returns the red component of `pixel` for a color model. Its range must be between 0 and 255, inclusive. A value of 0 means the pixel has no red in it. A value of 255 means red is at maximum intensity.

public abstract int getGreen (int pixel)

> The `getGreen()` method returns the green component of `pixel` for a color model. Its range must be between 0 and 255, inclusive. A value of 0 means the pixel has no green in it. A value of 255 means green is at maximum intensity.

public abstract int getBlue (int pixel)

The `getBlue()` method returns the blue component of `pixel` for a color model. Its range must be between 0 and 255, inclusive. A value of 0 means the pixel has no blue in it. A value of 255 means blue is at maximum intensity.

public int getRGB(int pixel)

The `getRGB()` method returns the color of `pixel` in the default RGB color model. If a subclass has changed the ordering or size of the different color components, `getRGB()` will return the pixel in the RGB color model (0xAARRGGBB order). In theory, the subclass does not need to override this method, unless it wants to make it final. Making this method final may yield a significant performance improvement.

public void finalize ()

The garbage collector calls `finalize()` when it determines that the `ColorModel` object is no longer needed. `finalize()` frees any internal resources that the `ColorModel` object has used.

DirectColorModel

The `DirectColorModel` class is a concrete subclass of `ColorModel`. It specifies a color model in which each pixel contains all the color information (alpha, red, green, and blue values) explicitly. Pixels are represented by 32-bit (`int`) quantities; the constructor lets you change which bits are allotted to each component.

All of the methods in this class, except constructors, are final, because of assumptions made by the implementation. You can create subclasses of `DirectColorModel`, but you can't override any of its methods. However, you should not need to develop your own subclass. Just create an instance of `DirectColorModel` with the appropriate constructor. Any subclassing results in serious performance degradation, because you are going from fast, static final method calls to dynamic method lookups.Constructors

public DirectColorModel (int bits, int redMask, int greenMask, int blueMask, int alphaMask)

This constructor creates a `DirectColorModel` in which `bits` represents the total number of bits used to represent a pixel; it must be less than or equal to 32. The `redMask`, `greenMask`, `blueMask`, and `alphaMask` specify where in a pixel each color component exists. Each of the bit masks must be contiguous (e.g., red cannot be the first, fourth, and seventh bits of the pixel), must be smaller than 2^bits, and should not exceed 8 bits. (You cannot display more than 8 bits of data for any color component, but the mask can be larger.) Combined, the masks together should be `bits` in length. The default RGB color model is:

```
new DirectColorModel (32, 0x00ff0000, 0x0000ff00, 0x000000ff, 0xff000000)
```

The run-time exception `IllegalArgumentException` is thrown if any of the following occur:

- The bits that are set in a mask are not contiguous.

- Mask bits overlap (i.e., the same bit is set in two or more masks).

- The number of mask bits exceeds `bits`.

public DirectColorModel (int bits, int redMask, int greenMask, int blueMask)

This constructor for `DirectColorModel` calls the first with an alpha mask of 0, which means that colors in this color model have no transparency component. All colors will be fully opaque with an alpha value of 255. The same restrictions for the red, green, and blue masks apply.

Methods

final public int getAlpha (int pixel)

The `getAlpha()` method returns the alpha component of `pixel` for the color model as a number from 0 to 255, inclusive. A value of 0 means the pixel is completely transparent, and the background will appear through the pixel. A value of 255 means the pixel is opaque, and you cannot see the background behind it.

final public int getRed (int pixel)

The `getRed()` method returns the red component of `pixel` for the color model. Its range is from 0 to 255. A value of 0 means the pixel has no red in it. A value of 255 means red is at maximum intensity.

final public int getGreen (int pixel)

The `getGreen()` method returns the green component of `pixel` for the color model. Its range is from 0 to 255. A value of 0 means the pixel has no green in it. A value of 255 means green is at maximum intensity.

final public int getBlue (int pixel)

The `getBlue()` method returns the blue component of `pixel` for the color model. Its range is from 0 to 255. A value of 0 means the pixel has no blue in it. A value of 255 means blue is at maximum intensity.

final public int getRGB (int pixel)

The `getRGB()` method returns the color of `pixel` in the default RGB color model. If a subclass has changed the ordering or size of the different color components, `getRGB()` will return the pixel in the RGB color model (0xAARRGGBB order). The `getRGB()` method in this subclass is identical to the method in `ColorModel` but overrides it to make it final.

Other methods

final public int getAlphaMask ()

The `getAlphaMask()` method returns the `alphaMask` from the `DirectColorModel` constructor (or 0 if constructor did not have `alphaMask`). The `alphaMask` specifies which bits in the pixel represent the alpha transparency component of the color model.

final public int getRedMask ()

The getRedMask() method returns the redMask from the DirectColorModel constructor. The redMask specifies which bits in the pixel represent the red component of the color model.

final public int getGreenMask ()

The getGreenMask() method returns the greenMask from the DirectColorModel constructor. The greenMask specifies which bits in the pixel represent the green component of the color model.

final public int getBlueMask ()

The getBlueMask() method returns the blueMask from the DirectColorModel constructor. The blueMask specifies which bits in the pixel represent the blue component of the color model.

IndexColorModel

The IndexColorModel is another concrete subclass of ColorModel. It specifies a ColorModel that uses a color map lookup table (with a maximum size of 256), rather than storing color information in the pixels themselves. Pixels are represented by an index into the color map, which is at most an 8-bit quantity. Each entry in the color map gives the alpha, red, green, and blue components of some color. One entry in the map can be designated "transparent." This is called the "transparent pixel"; the alpha component of this map entry is ignored.

All of the methods in this class, except constructors, are final because of assumptions made by the implementation. You shouldn't need to create subclasses; you can if necessary, but you can't override any of the IndexColorModel methods. Example 12.2 (later in this chapter) uses an IndexColorModel. Constructors

There are two sets of constructors for IndexColorModel. The first two constructors use a single-byte array for the color map. The second group implements the color map with separate byte arrays for each color component.

public IndexColorModel (int bits, int size, byte colorMap[], int start, boolean hasalpha, int transparent)

This constructor creates an IndexColorModel. bits is the number of bits used to represent each pixel and must not exceed 8. size is the number of elements in the map; it must be less than 2^bits. hasalpha should be true if the color map includes alpha (transparency) components and false if it doesn't. transparent is the location of the transparent pixel in the map (i.e., the pixel value that is considered transparent). If there is no transparent pixel, set transparent to -1.

The colorMap describes the colors used to paint pixels. start is the index within the colorMap array at which the map begins; prior elements of the array are ignored. An entry in the map consists of three or four consecutive bytes, representing the red, green, blue, and (optionally) alpha components. If hasalpha is false, a map entry consists of three bytes, and no alpha components are present; if hasalpha is true, map entries consist of four bytes, and all four components must be present.

For example, consider a pixel whose value is `p`, and a color map with a `hasalpha` set to `false`. Therefore, each element in the color map occupies three consecutive array elements. The red component of that pixel will be located at `colorMap[start + 3*p]`; the green component will be at `colorMap[start + 3*p + 1]`; and so on. The value of size may be smaller than 2^bits, meaning that there may be pixel values with no corresponding entry in the color map. These pixel values (i.e., `size <= p` < 2^bits) are painted with the color components set to 0; they are transparent if `hasalpha` is `true`, opaque otherwise.

If `bits` is too large (greater than 8), `size` is too large (greater than 2^bits), or the `colorMap` array is too small to hold the map, the run-time exception `ArrayIndexOutOfBoundsException` will be thrown.

public IndexColorModel (int bits, int size, byte colorMap[], int start, boolean hasalpha)

This version of the `IndexColorModel` constructor calls the previous constructor with a `transparent` index of -1; that is, there is no transparent pixel. If `bits` is too large (greater than 8), or `size` is too large (greater than 2^bits), or the `colorMap` array is too small to hold the map, the run-time exception, `ArrayIndexOutOfBoundsException` will be thrown.

public IndexColorModel (int bits, int size, byte red[], byte green[], byte blue[], int transparent)

The second set of constructors for `IndexColorModel` is similar to the first group, with the exception that these constructors use three or four separate arrays (one per color component) to represent the color map, instead of a single array.

The `bits` parameter still represents the number of bits in a pixel. `size` represents the number of elements in the color map. `transparent` is the location of the transparent pixel in the map (i.e., the pixel value that is considered transparent). If there is no transparent pixel, set `transparent` to -1.

The `red`, `green`, and `blue` arrays contain the color map itself. These arrays must have at least `size` elements. They contain the red, green, and blue components of the colors in the map. For example, if a pixel is at position `p`, `red[p]` contains the pixel's red component; `green[p]` contains the green component; and `blue[p]` contains the blue component. The value of `size` may be smaller than 2^bits, meaning that there may be pixel values with no corresponding entry in the color map. These pixel values (i.e., `size <= p <` 2^bits) are painted with the color components set to 0.

If `bits` is too large (greater than 8), `size` is too large (greater than 2^bits), or the red, green, and blue arrays are too small to hold the map, the run-time exception `ArrayIndexOutOfBoundsException` will be thrown.

public IndexColorModel (int bits, int size, byte red[], byte green[], byte blue[])

This version of the `IndexColorModel` constructor calls the previous one with a `transparent` index of -1; that is, there is no transparent pixel. If `bits` is too large (greater than 8), `size` is too large (greater than 2^bits), or the red, green, and blue arrays are too small to hold the map, the run-time exception `ArrayIndexOutOfBoundsException` will be thrown.

public IndexColorModel (int bits, int size, byte red[], byte green[], byte blue[], byte alpha[])

Like the previous constructor, this version creates an `IndexColorModel` with no transparent pixel. It differs from the previous constructor in that it supports transparency; the array alpha contains the map's transparency values. If `bits` is too large (greater than 8), `size` is too large (greater than 2^bits), or the red, green, blue, and alpha arrays are too small to hold the map, the run-time exception `ArrayIndexOutOfBoundsException` will be thrown.

Methods

final public int getAlpha (int pixel)

The `getAlpha()` method returns the alpha component of `pixel` for a color model, which is a number between 0 and 255, inclusive. A value of 0 means the pixel is completely transparent and the background will appear through the pixel. A value of 255 means the pixel is opaque and you cannot see the background behind it.

final public int getRed (int pixel)

The `getRed()` method returns the red component of `pixel` for a color model, which is a number between 0 and 255, inclusive. A value of 0 means the pixel has no red in it. A value of 255 means red is at maximum intensity.

final public int getGreen (int pixel)

The `getGreen()` method returns the green component of `pixel` for a color model, which is a number between 0 and 255, inclusive. A value of 0 means the pixel has no green in it. A value of 255 means green is at maximum intensity.

final public int getBlue (int pixel)

The `getBlue()` method returns the blue component of `pixel` for a color model, which is a number between 0 and 255, inclusive. A value of 0 means the pixel has no blue in it. A value of 255 means blue is at maximum intensity.

final public int getRGB (int pixel)

The `getRGB()` method returns the color of `pixel` in the default RGB color model. If a subclass has changed the ordering or size of the different color components, `getRGB()` will return the pixel in the RGB color model (0xAARRGGBB order). This version of `getRGB` is identical to the version in the `ColorModel` class but overrides it to make it final.

Other methods

final public int getMapSize()

The `getMapSize()` method returns the size of the color map (i.e., the number of distinct colors).

final public int getTransparentPixel ()

The `getTransparentPixel()` method returns the color map index for the transparent pixel in the color model. If no transparent pixel exists, it returns -1. It is not possible to change the transparent pixel after the color model has been created.

final public void getAlphas (byte alphas[])

The `getAlphas()` method copies the alpha components of the `ColorModel` into elements 0 through `getMapSize()-1` of the `alphas` array. Space must already be allocated in the `alphas` array.

final public void getReds (byte reds[])

The `getReds()` method copies the red components of the `ColorModel` into elements 0 through `getMapSize()-1` of the `reds` array. Space must already be allocated in the `reds` array.

final public void getGreens (byte greens[])

The `getGreens()` method copies the green components of the `ColorModel` into elements 0 through `getMapSize()-1` of the `greens` array. Space must already be allocated in the `greens` array.

final public void getBlues (byte blues[])

The `getBlues()` method copies the blue components of the `ColorModel` into elements 0 through `getMapSize()-1` of the `blues` array. Space must already be allocated in the `blues` array.

12.3 ImageProducer

The ImageProducer interface defines the methods that ImageProducer objects must implement. Image producers serve as sources for pixel data; they may compute the data themselves or interpret data from some external source, like a GIF file. No matter how it generates the data, an image producer's job is to hand that data to an image consumer, which usually renders the data on the screen. The methods in the ImageProducer interface let ImageConsumer objects register their interest in an image. The business end of an ImageProducer--that is, the methods it uses to deliver pixel data to an image consumer--are defined by the ImageConsumer interface. Therefore, we can summarize the way an image producer works as follows:

- It waits for image consumers to register their interest in an image.
- As image consumers register, it stores them in a Hashtable, Vector, or some other collection mechanism.
- As image data becomes available, it loops through all the registered consumers and calls their methods to transfer the data.

There's a sense in which you have to take this process on faith; image consumers are usually well hidden. If you call createImage(), an image consumer will eventually show up.

Every Image has an ImageProducer associated with it; to acquire a reference to the producer, use the getSource() method of Image.

Because an ImageProducer must call methods in the ImageConsumer interface, we won't show an example of a full-fledged producer until we have discussed ImageConsumer.

ImageProducer Interface

Methods

public void addConsumer (ImageConsumer ic)

> The addConsumer() method registers ic as an ImageConsumer interested in the Image information. Once an ImageConsumer is registered, the ImageProducer can deliver Image pixels immediately or

wait until `startProduction()` has been called.

Note that one image may have many consumers; therefore, `addConsumer()` usually stores image consumers in a collection like a `Vector` or `Hashtable`. There is one notable exception: if the producer has the image data in memory, `addConsumer()` can deliver the image to the consumer immediately. When `addConsumer()` returns, it has finished with the consumer. In this case, you don't need to manage a list of consumers, because there is only one image consumer at a time. (In this case, `addConsumer()` should be implemented as a synchronized method.)

public boolean isConsumer (ImageConsumer ic)

The `isConsumer()` method checks to see if `ic` is a registered `ImageConsumer` for this `ImageProducer`. If `ic` is registered, `true` is returned. If `ic` is not registered, `false` is returned.

public void removeConsumer (ImageConsumer ic)

The `removeConsumer()` method removes `ic` as a registered `ImageConsumer` for this `ImageProducer`. If `ic` was not a registered `ImageConsumer`, nothing should happen. This is not an error that should throw an exception. Once `ic` has been removed from the registry, the `ImageProducer` should no longer send data to it.

public void startProduction (ImageConsumer ic)

The `startProduction()` method registers `ic` as an `ImageConsumer` interested in the `Image` information and tells the `ImageProducer` to start sending the `Image` data immediately. The `ImageProducer` sends the image data to `ic` and all other registered `ImageConsumer` objects, through `addConsumer()`.

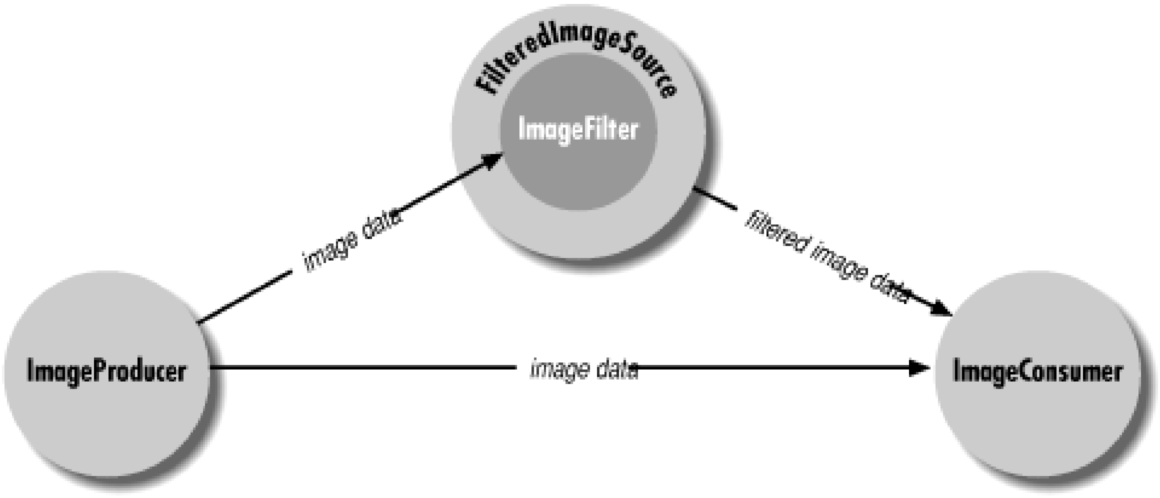
public void requestTopDownLeftRightResend (ImageConsumer ic)

The `requestTopDownLeftRightResend()` method is called by the `ImageConsumer` `ic` requesting that the `ImageProducer` retransmit the `Image` data in top-down, left-to-right order. If the `ImageProducer` is unable to send the data in that order or always sends the data in that order (like with `MemoryImageSource`), it can ignore the call.

FilteredImageSource

The `FilteredImageSource` class combines an `ImageProducer` and an `ImageFilter` to create a new `Image`. The image producer generates pixel data for an original image. The `FilteredImageSource` takes this data and uses an `ImageFilter` to produce a modified version: the image may be scaled, clipped, or rotated, or the colors shifted, etc. The `FilteredImageSource` is the image producer for the new image. The `ImageFilter` object transforms the original image's data to yield the new image; it implements the `ImageConsumer` interface. We cover the `ImageConsumer` interface in ImageConsumer and the `ImageFilter` class in ImageFilter. Figure 12.1 shows the relationship between an `ImageProducer`, `FilteredImageSource`, `ImageFilter`, and the `ImageConsumer`.

Figure 12.1: Image producers, filters, and consumers



Constructors

public FilteredImageSource (ImageProducer original, ImageFilter filter)

> The `FilteredImageSource` constructor creates an image producer that combines an image, `original`, and a filter, `filter`, to create a new image. The `ImageProducer` of the original image is the constructor's first parameter; given an `Image`, you can acquire its `ImageProducer` by using the `getSource()` method. The following code shows how to create a new image from an original. ImageFilter shows several extensive examples of image filters.

```
Image image = getImage (new URL
      ("http://www.ora.com/graphics/headers/homepage.gif"));
Image newOne = createImage (new FilteredImageSource
      (image.getSource(), new SomeImageFilter()));
```

ImageProducer interface methods

The `ImageProducer` interface methods maintain an internal table for the image consumers. Since this is private, you do not have direct access to it.

public synchronized void addConsumer (ImageConsumer ic)

> The `addConsumer()` method adds `ic` as an `ImageConsumer` interested in the pixels for this image.

public synchronized boolean isConsumer (ImageConsumer ic)

> The `isConsumer()` method checks to see if `ic` is a registered `ImageConsumer` for this `ImageProducer`. If `ic` is registered, `true` is returned. If not registered, `false` is returned.

public synchronized void removeConsumer (ImageConsumer ic)

The `removeConsumer()` method removes `ic` as a registered `ImageConsumer` for this `ImageProducer`.

public void startProduction (ImageConsumer ic)

The `startProduction()` method registers `ic` as an `ImageConsumer` interested in the `Image` information and tells the `ImageProducer` to start sending the `Image` data immediately.

public void requestTopDownLeftRightResend (ImageConsumer ic)

The `requestTopDownLeftRightResend()` method registers `ic` as an `ImageConsumer` interested in the `Image` information and requests the `ImageProducer` to retransmit the `Image` data in top-down, left-to-right order.

MemoryImageSource

The `MemoryImageSource` class allows you to create images completely in memory; you generate pixel data, place it in an array, and hand that array and a `ColorModel` to the `MemoryImageSource` constructor. The `MemoryImageSource` is an image producer that can be used with a consumer to display the image on the screen. For example, you might use a `MemoryImageSource` to display a Mandelbrot image or some other image generated by your program. You could also use a `MemoryImageSource` to modify a pre-existing image; use `PixelGrabber` to get the image's pixel data, modify that data, and then use a `MemoryImageSource` as the producer for the modified image. Finally, you can use `MemoryImageSource` to simplify implementation of a new image type; you can develop a class that reads an image in some unsupported format from a local file or the network; interprets the image file and puts pixel data into an array; and uses a `MemoryImageSource` to serve as an image producer. This is simpler than implementing an image producer yourself, but it isn't quite as flexible; you lose the ability to display partial images as the data becomes available.

In Java 1.1, `MemoryImageSource` supports multiframe images to animate a sequence. In earlier versions, it was necessary to create a dynamic `ImageFilter` to animate the image. Constructors

There are six constructors for `MemoryImageSource`, each with slightly different parameters. They all create an image producer that delivers some array of data to an image consumer. The constructors are:

public MemoryImageSource (int w, int h, ColorModel cm, byte pix[], int off, int scan)
public MemoryImageSource (int w, int h, ColorModel cm, byte pix[], int off, int scan, Hashtable props)
public MemoryImageSource (int w, int h, ColorModel cm, int pix[], int off, int scan)
public MemoryImageSource (int w, int h, ColorModel cm, int pix[], int off, int scan, Hashtable props)
public MemoryImageSource (int w, int h, int pix[], int off, int scan)
public MemoryImageSource (int w, int h, int pix[], int off, int scan, Hashtable props)

The parameters that might be present are:

`w`

Width of the image being created, in pixels.

`h`

Height of the image being created, in pixels.

`cm`

The `ColorModel` that describes the color representation used in the pixel data. If this parameter is not present, the `MemoryImageSource` uses the default RGB color model (`ColorModel.getRGBDefault()`).

`pix[]`

The array of pixel information to be converted into an image. This may be either a `byte` array or an `int` array, depending on the color model. If you're using a direct color model (including the default RGB color model), `pix` is usually an `int` array; if it isn't, it won't be able to represent all 16 million possible colors. If you're using an indexed color model, the array should be a `byte` array. However, if you use an `int` array with an indexed color model, the `MemoryImageSource` ignores the three high-order bytes because an indexed color model has at most 256 entries in the color map. In general: if your color model requires more than 8 bits of data per pixel, use an `int` array; if it requires 8 bits or less, use a `byte` array.

`off`

The first pixel used in the array (usually 0); prior pixels are ignored.

`scan`

The number of pixels per line in the array (usually equal to `w`). The number of pixels per scan line in the array may be larger than the number of pixels in the scan line. Extra pixels in the array are ignored.

`props`

A `Hashtable` of the properties associated with the image. If this argument isn't present, the constructor assumes there are no properties.

The pixel at location (`x`, `y`) in the image is located at `pix[y * scan + x + off]`. ImageProducer interface methods

In Java 1.0, the `ImageProducer` interface methods maintain a single internal variable for the image consumer because the image is delivered immediately and synchronously. There is no need to worry about multiple consumers; as soon as one registers, you give it the image, and you're done. These methods keep track of this single `ImageConsumer`.

In Java 1.1, `MemoryImageSource` supports animation. One consequence of this new feature is that it isn't always possible to deliver all the image's data immediately. Therefore, the class maintains a list of image consumers that are notified when each frame is generated. Since this list is private, you do not have direct access to it.

public synchronized void addConsumer (ImageConsumer ic)

> The `addConsumer()` method adds `ic` as an `ImageConsumer` interested in the pixels for this image.

public synchronized boolean isConsumer (ImageConsumer ic)

> The `isConsumer()` method checks to see if `ic` is a registered `ImageConsumer` for this `ImageProducer`. If `ic` is registered, `true` is returned. If `ic` is not registered, `false` is returned.

public synchronized void removeConsumer (ImageConsumer ic)

> The `removeConsumer()` method removes `ic` as a registered `ImageConsumer` for this `ImageProducer`.

public void startProduction (ImageConsumer ic)

> The `startProduction()` method calls `addConsumer()`.

public void requestTopDownLeftRightResend (ImageConsumer ic)

> The `requestTopDownLeftRightResend()` method does nothing since in-memory images are already in this format or are multiframed, with each frame in this format.

Animation methods

In Java 1.1, `MemoryImageSource` supports animation; it can now pass multiple frames to interested image consumers. This feature mimics GIF89a's multiframe functionality. (If you have GIF89a animations, you can display them using `getImage()` and `drawImage()`; you don't have to build a complicated creature using `MemoryImageSource`.) . An animation example follows in [Example 12.3] (later in this chapter).

public synchronized void setAnimated(boolean animated) ★

> The `setAnimated()` method notifies the `MemoryImageSource` if it will be in animation mode (`animated` is `true`) or not (`animated` is `false`). By default, animation is disabled; you must call this method to generate an image sequence. To prevent losing data, call this method immediately after calling the `MemoryImageSource` constructor.

public synchronized void setFullBufferUpdates(boolean fullBuffers) ★

> The `setFullBufferUpdates()` method controls how image updates are done during an animation. It is ignored if you are not creating an animation. If `fullBuffers` is `true`, this method tells the `MemoryImageSource` that it should always send all of an image's data to the consumers whenever it received new data (by a call to `newPixels()`). If `fullBuffers` is `false`, the `MemoryImageSource` sends only the changed portion of the image and notifies consumers (by a call to `ImageConsumer.setHints()`) that frames sent will be complete.

Like `setAnimated()`, `setFullBufferUpdates()` should be called immediately after calling the `MemoryImageSource` constructor, before the animation is started.

To do the actual animation, you update the image array `pix[]` that was specified in the constructor and call one of the overloaded `newPixels()` methods to tell the `MemoryImageSource` that you have changed the image data. The parameters to `newPixels()` determine whether you are animating the entire image or just a portion of the image. You can also supply a new array to take pixel data from, replacing `pix[]`. In any case, `pix[]` supplies the initial image data (i.e., the first frame of the animation).

If you have not called `setAnimated(true)`, calls to any version of `newPixels()` are ignored.

public void newPixels() ★

The version of `newPixels()` with no parameters tells the `MemoryImageSource` to send the entire pixel data (frame) to all the registered image consumers again. Data is taken from the original array `pix[]`. After the data is sent, the `MemoryImageSource` notifies consumers that a frame is complete by calling `imageComplete(ImageConsumer.SINGLEFRAMEDONE)`, thus updating the display when the image is redisplayed. Remember that in many cases, you don't need to update the entire image; updating part of the image saves CPU time, which may be crucial for your application. To update part of the image, call one of the other versions of `newPixels()`.

public synchronized void newPixels(int x, int y, int w, int h) ★

This `newPixels()` method sends part of the image in the array `pix[]` to the consumers. The portion of the image sent has its upper left corner at the point (`x`, `y`), width `w` and height `h`, all in pixels. Changing part of the image rather than the whole thing saves considerably on system resources. Obviously, it is appropriate only if most of the image is still. For example, you could use this method to animate the steam rising from a cup of hot coffee, while leaving the cup itself static (an image that should be familiar to anyone reading JavaSoft's Web site). After the data is sent, consumers are notified that a frame is complete by a call to `imageComplete(ImageConsumer.SINGLEFRAMEDONE)`, thus updating the display when the image is redisplayed.

If `setFullBufferUpdates()` was called, the entire image is sent, and the dimensions of the bounding box are ignored.

public synchronized void newPixels(int x, int y, int w, int h, boolean frameNotify) ★

This `newPixels()` method is identical to the last, with one exception: consumers are notified that new image data is available only when `frameNotify` is `true`. This method allows you to generate new image data in pieces, updating the consumers only once when you are finished.

If `setFullBufferUpdates()` was called, the entire image is sent, and the dimensions of the bounding box are ignored.

public synchronized void newPixels(byte[] newpix, ColorModel newmodel, int offset, int scansize) ★

public synchronized void newPixels(int[] newpix, ColorModel newmodel, int offset, int scansize) ★

> These `newPixels()` methods change the source of the animation to the `byte` or `int` array `newpix[]`, with a `ColorModel` of `newmodel`. `offset` marks the beginning of the data in `newpix` to use, while `scansize` states the number of pixels in `newpix` per line of `Image` data. Future calls to other versions of `newPixels()` should modify `newpix[]` rather than `pix[]`.

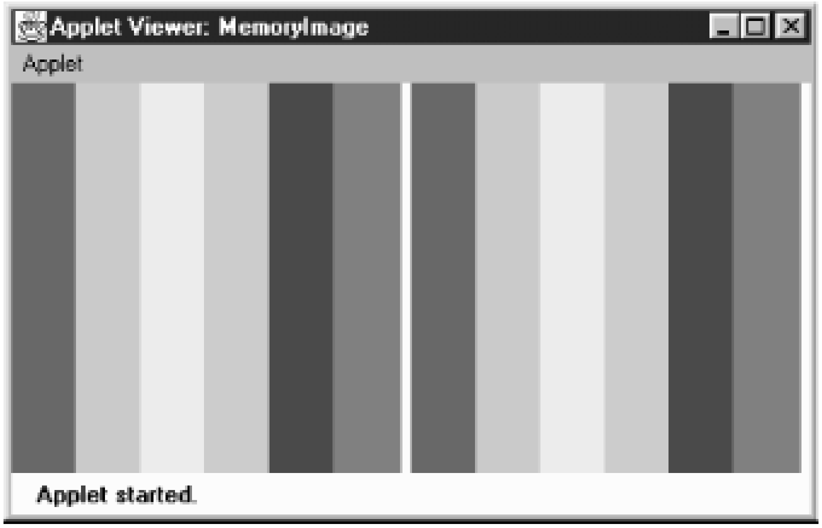
Using MemoryImageSource to create a static image

You can create an image by generating an integer or byte array in memory and converting it to an image with `MemoryImageSource`. The following `MemoryImage` applet generates two identical images that display a series of color bars from left to right. Although the images look the same, they were generated differently: the image on the left uses the default `DirectColorModel`; the image on the right uses an `IndexColorModel`.

Because the image on the left uses a `DirectColorModel`, it stores the actual color value of each pixel in an array of integers (`rgbPixels[]`). The image on the right can use a byte array (`indPixels[]`) because the `IndexColorModel` puts the color information in its color map instead of the pixel array; elements of the pixel array need to be large enough only to address the entries in this map. Images that are based on `IndexColorModel` are generally more efficient in their use of space (integer vs. byte arrays, although `IndexColorModel` requires small support arrays) and in performance (if you filter the image).

The output from this example is shown in Figure 12.2. The source is shown in Example 12.2.

Figure 12.2: MemoryImage applet output



Example 12.2: MemoryImage Test Program

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
public class MemoryImage extends Applet {
    Image i, j;
    int width = 200;
    int height = 200;
    public void init () {
        int rgbPixels[] = new int [width*height];
        byte indPixels[] = new byte [width*height];
        int index = 0;
        Color colorArray[] = {Color.red, Color.orange, Color.yellow,
                Color.green, Color.blue, Color.magenta};
        int rangeSize = width / colorArray.length;
        int colorRGB;
        byte colorIndex;
        byte reds[]   = new byte[colorArray.length];
        byte greens[] = new byte[colorArray.length];
        byte blues[]  = new byte[colorArray.length];
        for (int i=0;i<colorArray.length;i++) {
            reds[i]   = (byte)colorArray[i].getRed();
            greens[i] = (byte)colorArray[i].getGreen();
            blues[i]  = (byte)colorArray[i].getBlue();
        }
        for (int y=0;y<height;y++) {
            for (int x=0;x<width;x++) {
                if (x < rangeSize) {
                    colorRGB = Color.red.getRGB();
                    colorIndex = 0;
                } else if (x < (rangeSize*2)) {
                    colorRGB = Color.orange.getRGB();
                    colorIndex = 1;
                } else if (x < (rangeSize*3)) {
                    colorRGB = Color.yellow.getRGB();
                    colorIndex = 2;
                } else if (x < (rangeSize*4)) {
                    colorRGB = Color.green.getRGB();
                    colorIndex = 3;
                } else if (x < (rangeSize*5)) {
                    colorRGB = Color.blue.getRGB();
                    colorIndex = 4;
                } else {
                    colorRGB = Color.magenta.getRGB();
                    colorIndex = 5;
                }
                rgbPixels[index] = colorRGB;
                indPixels[index] = colorIndex;
                index++;
            }
        }
        i = createImage (new MemoryImageSource (width, height, rgbPixels,
            0, width));
        j = createImage (new MemoryImageSource (width, height,
            new IndexColorModel (8, colorArray.length, reds, greens, blues),
            indPixels, 0, width));
    }
    public void paint (Graphics g) {
        g.drawImage (i, 0, 0, this);
        g.drawImage (j, width+5, 0, this);
    }
}
```

Almost all of the work is done in `init()` (which, in a real applet, isn't a terribly good idea; ideally `init()` should be lightweight). Previously, we explained the color model's use for the images on the left and the right. Toward the end of `init()`, we create the images `i` and `j` by calling `createImage()` with a `MemoryImageSource` as the image producer. For image `i`, we used the simplest `MemoryImageSource` constructor, which uses the default RGB color model. For `j`, we called the `IndexColorModel` constructor within the `MemoryImageSource` constructor, to create a color map that has only six entries: one for each of the colors we use. Using MemoryImageSource for animation

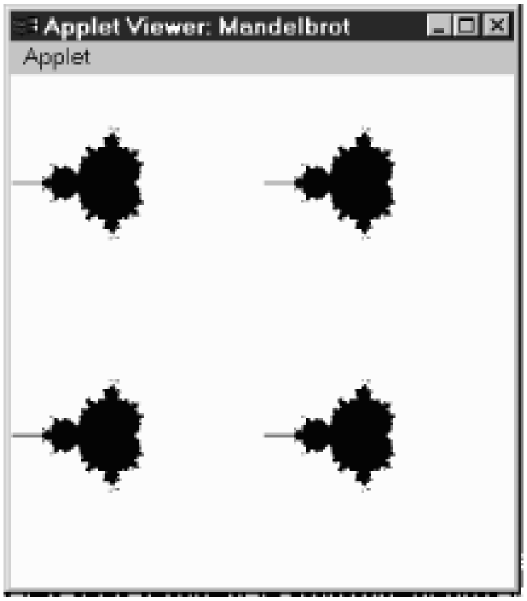
As we've seen, Java 1.1 gives you the ability to create an animation using a `MemoryImageSource` by updating the image data in memory; whenever you have finished an update, you can send the resulting frame to the consumers. This technique gives you a way to do animations that consume very little memory, since you keep overwriting the original image. The applet in Example 12.3 demonstrates `MemoryImageSource`'s animation capability by creating a Mandelbrot image in memory, updating the image as new points are added. Figure 12.3 shows the results, using four consumers to display the image four times.

Example 12.3: Mandelbrot Program

```
// Java 1.1 only
import java.awt.*;
import java.awt.image.*;
import java.applet.*;
public class Mandelbrot extends Applet implements Runnable {
    Thread animator;
    Image im1, im2, im3, im4;
    public void start() {
        animator = new Thread(this);
        animator.start();
    }
    public synchronized void stop() {
        animator = null;
    }
    public void paint(Graphics g) {
        if (im1 != null)
            g.drawImage(im1, 0, 0, null);
        if (im2 != null)
            g.drawImage(im2, 0, getSize().height / 2, null);
        if (im3 != null)
            g.drawImage(im3, getSize().width / 2, 0, null);
        if (im4 != null)
```

```
            g.drawImage(im4, getSize().width / 2, getSize().height / 2, null);
    }
    public void update (Graphics g) {
        paint (g);
    }
    public synchronized void run() {
        Thread.currentThread().setPriority(Thread.MIN_PRIORITY);
        int width = getSize().width / 2;
        int height = getSize().height / 2;
        byte[] pixels = new byte[width * height];
        int index = 0;
        int iteration=0;
        double a, b, p, q, psq, qsq, pnew, qnew;
        byte[] colorMap = {(byte)255, (byte)255, (byte)255, // white
                           (byte)0, (byte)0, (byte)0};      // black
        MemoryImageSource mis = new MemoryImageSource(
            width, height,
            new IndexColorModel (8, 2, colorMap, 0, false, -1),
            pixels, 0, width);
        mis.setAnimated(true);
        im1 = createImage(mis);
        im2 = createImage(mis);
        im3 = createImage(mis);
        im4 = createImage(mis);
        // Generate Mandelbrot
        final int ITERATIONS = 16;
        for (int y=0; y<height; y++) {
            b = ((double)(y-64))/32;
            for (int x=0; x<width; x++) {
                a = ((double)(x-64))/32;
                p=q=0;
                iteration = 0;
                while (iteration < ITERATIONS) {
                    psq = p*p;
                    qsq = q*q;
                    if ((psq + qsq) >= 4.0)
                        break;
                    pnew = psq - qsq + a;
                    qnew = 2*p*q+b;
                    p = pnew;
                    q = qnew;
                    iteration++;
                }
                if (iteration == ITERATIONS) {
                    pixels[index] = 1;
                    mis.newPixels(x, y, 1, 1);
                    repaint();
                }
                index++;
            }
        }
    }
}
```

Figure 12.3: Mandelbrot output



Most of the applet in Example 12.3 should be self-explanatory. The `init()` method starts the thread in which we do our computation. `paint()` just displays the four images we create. All the work, including the computation, is done in the thread's `run()` method. `run()` starts by setting up a color map, creating a `MemoryImageSource` with animation enabled and creating four images using that source as the producer. It then does the computation, which I won't explain; for our purposes, the interesting part is what happens when we've computed a pixel. We set the appropriate byte in our data array, `pixels[]`, and then call `newPixels()`, giving the location of the new pixel and its size (1 by 1) as arguments. Thus, we redraw the images for every new pixel. In a real application, you would probably compute a somewhat larger chunk of new data before updating the screen, but the same principles apply.

12.4 ImageConsumer

The `ImageConsumer` interface specifies the methods that must be implemented to receive data from an `ImageProducer`. For the most part, that is the only context in which you need to know about the `ImageConsumer` interface. If you write an image producer, it will be handed a number of obscure objects, about which you know nothing except that they implement `ImageConsumer`, and that you can therefore call the methods discussed in this section to deliver your data. The chances that you will ever implement an image consumer are rather remote, unless you are porting Java to a new environment. It is more likely that you will want to subclass `ImageFilter`, in which case you may need to implement some of these methods. But most of the time, you will just need to know how to hand your data off to the next element in the chain.

The `java.awt.image` package includes two classes that implement `ImageConsumer: PixelGrabber` and `ImageFilter` (and its subclasses). These classes are unique in that they don't display anything on the screen. `PixelGrabber` takes the image data and stores it in a pixel array; you can use this array to save the image in a file, generate a new image, etc. `ImageFilter`, which is used in conjunction with `FilteredImageSource`, modifies the image data; the `FilteredImageSource` sends the modified image to another consumer, which can further modify or display the new image. When you draw an image on the screen, the JDK's `ImageRepresentation` class is probably doing the real work. This class is part of the `sun.awt.image` package. You really don't need to know anything about it, although you may see `ImageRepresentation` mentioned in a stack trace if you try to filter beyond the end of a pixel array.

ImageConsumer Interface

Constants

There are two sets of constants for `ImageConsumer`. One set represents those that can be used for the `imageComplete()` method. The other is used with the `setHints()` method. See the descriptions of those methods on how to use them.

The first set of flags is for the `imageComplete()` method:

public static final int IMAGEABORTED

> The `IMAGEABORTED` flag signifies that the image creation process was aborted and the image is not complete. In the image production process, an abort could mean multiple things. It is possible that retrying the production would succeed.

public static final int IMAGEERROR

> The `IMAGEERROR` flag signifies that an error was encountered during the image creation process and the image is not complete. In the image production process, an error could mean multiple things. More than likely, the image file or pixel data is invalid, and retrying won't succeed.

public static final int SINGLEFRAMEDONE

> The `SINGLEFRAMEDONE` flag signifies that a frame other than the last has completed loading. There are additional frames to display, but a new frame is available and is complete. For an example of this flag in use, see the dynamic `ImageFilter` example in Example 12.8.

public static final int STATICIMAGEDONE

> The `STATICIMAGEDONE` flag signifies that the image has completed loading. If this is a multiframe image, all frames have been generated. For an example of this flag in use, see the dynamic `ImageFilter` example in Example 12.8.

The following set of flags can be ORed together to form the single parameter to the `setHints()` method. Certain flags do not make sense set together, but it is the responsibility of the concrete `ImageConsumer` to enforce this.

public static final int COMPLETESCANLINES

> The `COMPLETESCANLINES` flag signifies that each call to `setPixels()` will deliver at least one complete scan line of pixels to this consumer.

public static final int RANDOMPIXELORDER

> The `RANDOMPIXELORDER` flag tells the consumer that pixels are not provided in any particular order. Therefore, the consumer cannot perform optimization that depends on pixel delivery order. In the absence of both `COMPLETESCANLINES` and `RANDOMPIXELORDER`, the `ImageConsumer` should assume pixels will arrive in `RANDOMPIXELORDER`.

public static final int SINGLEFRAME

> The `SINGLEFRAME` flag tells the consumer that this image contains a single non-changing frame. This is the case with most image formats. An example of an image that does not contain a single frame is the multiframe GIF89a image.

public static final int SINGLEPASS

> The `SINGLEPASS` flag tells the consumer to expect each pixel once and only once. Certain image formats, like progressive JPEG images, deliver a single image several times, with each pass yielding a sharper image.

public static final int TOPDOWNLEFTRIGHT

> The final `setHints()` flag, `TOPDOWNLEFTRIGHT`, tells the consumer to expect the pixels in a top-down, left-right order. This flag will almost always be set.

Methods

The interface methods are presented in the order in which they are normally called by an `ImageProducer`.

void setDimensions (int width, int height)

> The `setDimensions()` method should be called once the `ImageProducer` knows the `width` and `height` of the image. This is the actual `width` and `height`, not necessarily the scaled size. It is the consumer's responsibility to do the scaling and resizing.

void setProperties (Hashtable properties)

> The `setProperties()` method should only be called by the `ImageProducer` if the image has any properties that should be stored for later retrieval with the `getProperty()` method of `Image`. Every image format has its own property set. One property that tends to be common is the "comment" property. `properties` represents the `Hashtable` of properties for the image; the name of each property is used as the `Hashtable` key.

void setColorModel (ColorModel model)

> The `setColorModel()` method gives the `ImageProducer` the opportunity to tell the `ImageConsumer` that the `ColorModel` `model` will be used for the majority of pixels in the image. The `ImageConsumer` may use this information for optimization. However, each call to `setPixels()` contains its own `ColorModel`, which isn't necessarily the same as the color model given here. In other words, `setColorModel()` is only advisory; it does not guarantee that all (or any) of the pixels in the image will use this model. Using different color models for different parts of an image is possible, but not recommended.

void setHints (int hints)

> An `ImageProducer` should call the `setHints()` method prior to any `setPixels()` calls. The `hints` are formed by ORing the constants `COMPLETESCANLINES`, `RANDOMPIXELORDER`, `SINGLEFRAME`, `SINGLEPASS`, and `TOPDOWNLEFTRIGHT`. These hints give the image consumer information about the order in which the producer will deliver pixels. When the `ImageConsumer` is receiving pixels, it can take advantage of these hints for optimization.

void setPixels (int x, int y, int width, int height, ColorModel model, byte pixels[], int offset, int scansize)

> An ImageProducer calls the `setPixels()` method to deliver the image pixel data to the `ImageConsumer`. The bytes are delivered a rectangle at a time. (`x`, `y`) represents the top left corner of the rectangle; its dimensions are `width` x `height`. `model` is the `ColorModel` used for this set of pixels; different calls to `setPixels()` may use different color models. The pixels themselves are taken from the byte array `pixels`. `offset` is the first element of the pixel array that will be used. `scansize` is the length of the scan lines in the array. In most cases, you want the consumer to render all the pixels on the scan line; in this case, `scansize` will equal `width`. However, there are cases in which you want the consumer to ignore part of the scan line; you may be clipping an image, and the ends of the scan line fall outside the clipping region. In this case, rather than copying the pixels you want into a new array, you can specify a `width` that is smaller than `scansize`.

That's a lot of information, but it's easy to summarize. A pixel located at point (`x1`, `y1`) within the rectangle being delivered to the consumer is located at position `((y1 - y) * scansize + (x1 - x) + offset)` within the array `pixels[]`. Figure 12.4 shows how the pixels delivered by `setPixels()` fit into the complete image; Figure 12.5 shows how pixels are stored within the array.

Figure 12.4: Delivering pixels for an image

[Graphic: Figure 12-4]

Figure 12.5: Storing pixels in an array

[Graphic: Figure 12-5]

void setPixels (int x, int y, int width, int height, ColorModel model, int pixels[], int offset, int scansize)

The second `setPixels()` method is similar to the first. `pixels[]` is an array of `int`s; this is necessary when you have more than eight bits of data per pixel.

void imageComplete (int status)

The `ImageProducer` calls `imageComplete()` to tell an `ImageConsumer` that it has transferred a complete image. The status argument is a flag that describes exactly why the `ImageProducer` has finished. It may have one of the following values: `IMAGEABORTED` (if the image production was aborted); `IMAGEERROR` (if an error in producing the image occurred); `SINGLEFRAMEDONE` (if a single frame of a multiframe image has been completed); or `STATICIMAGEDONE` (if all pixels have been delivered). When `imageComplete()` gets called, the `ImageConsumer` should call the image producer's `removeConsumer()` method, unless it wants to receive additional frames (status of `SINGLEFRAMEDONE`).

PPMImageDecoder

Now that we have discussed the `ImageConsumer` interface, we're finally ready to give an example of a full-fledged `ImageProducer`. This producer uses the methods of the `ImageConsumer` interface to communicate with image consumers; image consumers use the `ImageProducer` interface to register themselves with this producer.

Our image producer will interpret images in the PPM format.[1] PPM is a simple image format developed by Jef Poskanzer as part of the *pbmplus* image conversion package. A PPM file starts with a header consisting of the image type, the image's width and height in pixels, and the maximum value of any RGB component. The header is entirely in ASCII. The pixel data follows the header; it is either in binary (if the image type is P6) or ASCII (if the image type is P3). The pixel data is simply a series of bytes describing the color of each pixel, moving left to right and top to bottom. In binary format, each pixel is represented by three bytes: one for red, one for green, and one for blue. In ASCII format, each pixel is represented by three numeric values, separated by white space (space, tab, or newline). A comment may occur anywhere in the file, but it would be surprising to see one outside of the header. Comments start with # and continue to the end of the line. ASCII format files are obviously much larger than binary files. There is no compression on either file type.

> [1] For more information about PPM and the *pbmplus* package, see *Encyclopedia of Graphics File Formats*, by James D. Murray and William VanRyper (from O'Reilly & Associates). See also http://www.acme.com/.

The `PPMImageDecoder` source is listed in Example 12--4. The applet that uses this class is shown in Example 12.5. You can reuse a lot of the code in the `PPMImageDecoder` when you implement your own image producers.

Example 12.4: PPMImageDecoder Source

```
import java.awt.*;
import java.awt.image.*;
import java.util.*;
import java.io.*;
public class PPMImageDecoder implements ImageProducer {
/* Since done in-memory, only one consumer */
    private ImageConsumer consumer;
    boolean loadError = false;
    int width;
    int height;
    int store[][];
    Hashtable props = new Hashtable();
/* Format of Ppm file is single pass/frame, w/ complete scan lines in order */
    private static int PpmHints = (ImageConsumer.TOPDOWNLEFTRIGHT |
                                   ImageConsumer.COMPLETESCANLINES |
                                   ImageConsumer.SINGLEPASS |
                                   ImageConsumer.SINGLEFRAME);
```

The class starts by declaring class variables and constants. We will use the variable `PpmHints` when we call `setHints()`. Here, we set this variable to a collection of "hint" constants that indicate we will produce pixel data in top-down, left-right order; we will always send complete scan lines; we will make only one pass over the pixel data (we will send each pixel once); and there is one frame per image (i.e., we aren't producing a multiframe sequence).

The next chunk of code implements the `ImageProducer` interface; consumers use it to request image data:

```
/* There is only a single consumer. When it registers, produce image. */
/* On error, notify consumer. */
    public synchronized void addConsumer (ImageConsumer ic) {
        consumer = ic;
        try {
            produce();
        }catch (Exception e) {
            if (consumer != null)
                consumer.imageComplete (ImageConsumer.IMAGEERROR);
        }
        consumer = null;
    }
/* If consumer passed to routine is single consumer, return true, else false. */
    public synchronized boolean isConsumer (ImageConsumer ic) {
        return (ic == consumer);
    }
/* Disables consumer if currently consuming. */
    public synchronized void removeConsumer (ImageConsumer ic) {
        if (consumer == ic)
            consumer = null;
    }
/* Production is done by adding consumer. */
    public void startProduction (ImageConsumer ic) {
        addConsumer (ic);
    }
    public void requestTopDownLeftRightResend (ImageConsumer ic) {
        // Not needed.  The data is always in this format.
    }
```

The previous group of methods implements the `ImageProducer` interface. They are quite simple, largely because of the way this `ImageProducer` generates images. It builds the image in memory before delivering it to the consumer; you must call the `readImage()` method (discussed shortly) before you can create an image with this consumer. Because the image is in memory before any consumers can register their interest, we can write an `addConsumer()` method that registers a consumer and delivers all the data to that consumer before returning. Therefore, we don't need to manage a list of consumers in a `Hashtable` or some other collection object. We can store the current consumer in an instance variable `ic` and forget about any others: only one consumer exists at a time. To make sure that only one consumer exists at a time, we synchronize the `addConsumer()`, `isConsumer()`, and `removeConsumer()` methods. Synchronization prevents another consumer from registering itself before the current consumer has finished. If you write an `ImageProducer` that builds the image in memory before delivering it, you can probably use this code verbatim.

`addConsumer()` is little more than a call to the method `produce()`, which handles "consumer relations": it delivers the pixels to the consumer using the methods in the `ImageConsumer` interface. If `produce()` throws an exception, `addConsumer()` calls `imageComplete()` with an `IMAGEERROR` status code. Here's the code for the `produce()` method:

```
/* Production Process:
        Prerequisite: Image already read into store array. (readImage)
                      props / width / height already set (readImage)
        Assumes RGB Color Model - would need to filter to change.
        Sends Ppm Image data to consumer.
        Pixels sent one row at a time.
*/
    private void produce () {
        ColorModel cm = ColorModel.getRGBdefault();
        if (consumer != null) {
            if (loadError) {
                consumer.imageComplete (ImageConsumer.IMAGEERROR);
            } else {
                consumer.setDimensions (width, height);
                consumer.setProperties (props);
                consumer.setColorModel (cm);
                consumer.setHints (PpmHints);
                for (int j=0;j<height;j++)
                    consumer.setPixels (0, j, width, 1, cm, store[j], 0, width);
                consumer.imageComplete (ImageConsumer.STATICIMAGEDONE);
            }
        }
    }
```

`produce()` just calls the `ImageConsumer` methods in order: it sets the image's dimensions, hands off an empty `Hashtable` of properties, sets the color model (the default RGB model) and the hints, and then calls `setPixels()` once for each row of pixel data. The data is in the integer array `store[][]`, which has already been loaded by the `readImage()` method (defined in the following code). When the data is delivered, the method `setPixels()` calls `imageComplete()` to indicate that the image has been finished successfully.

```
/* Allows reading to be from internal byte array, in addition to disk/socket */
    public void readImage (byte b[]) {
        readImage (new ByteArrayInputStream (b));
    }
/* readImage reads image data from Stream */
/* parses data for PPM format             */
/* closes inputstream when done           */
    public void readImage (InputStream is) {
        long tm = System.currentTimeMillis();
        boolean raw=false;
        DataInputStream dis = null;
        BufferedInputStream bis = null;
        try {
            bis = new BufferedInputStream (is);
            dis = new DataInputStream (bis);
            String word;
            word = readWord (dis);
            if ("P6".equals (word)) {
                raw = true;
            } else if ("P3".equals (word)) {
                raw = false;
            } else {
                throw (new AWTException ("Invalid Format " + word));
            }
            width = Integer.parseInt (readWord (dis));
            height = Integer.parseInt (readWord (dis));
            // Could put comments in props - makes readWord more complex
            int maxColors = Integer.parseInt (readWord (dis));
            if ((maxColors < 0) || (maxColors > 255)) {
                throw (new AWTException ("Invalid Colors " + maxColors));
            }
            store = new int[height][width];
            if (raw) {                   // binary format (raw) pixel data
                byte row[] = new byte [width*3];
                for (int i=0;i<height;i++){
                    dis.readFully (row);
                    for (int j=0,k=0;j<width;j++,k+=3) {
                        int red = row[k];
                        int green = row[k+1];
                        int blue = row[k+2];
                        if (red < 0)
                            red +=256;
                        if (green < 0)
                            green +=256;
                        if (blue < 0)
                            blue +=256;
                        store[i][j] = (0xff<< 24) | (red << 16) |
                                      (green << 8) | blue;
                    }
                }
            } else {                     // ASCII pixel data
                for (int i=0;i<height;i++) {
                    for (int j=0;j<width;j++) {
                        int red = Integer.parseInt (readWord (dis));
                        int green = Integer.parseInt (readWord (dis));
                        int blue = Integer.parseInt (readWord (dis));
                        store[i][j] = (0xff<< 24) | (red << 16) |
                                      (green << 8) | blue;
                    }
                }
            }
        } catch (IOException io) {
            loadError = true;
            System.out.println ("IO Exception " + io.getMessage());
        } catch (AWTException awt) {
            loadError = true;
            System.out.println ("AWT Exception " + awt.getMessage());
        } catch (NoSuchElementException nse) {
            loadError = true;
            System.out.println ("No Such Element Exception " + nse.getMessage());
        } finally {
            try {
                if (dis != null)
```

```
                dis.close();
            if (bis != null)
                bis.close();
            if (is != null)
                is.close();
        } catch (IOException io) {
            System.out.println ("IO Exception " + io.getMessage());
        }
    }
    System.out.println ("Done in " + (System.currentTimeMillis() - tm)
                        + " ms");
}
```

`readImage()` reads the image data from an `InputStream` and converts it into the array of pixel data that `produce()` transfers to the consumer. Code using this class must call `readImage()` to process the data before calling `createImage()`; we'll see how this works shortly. Although there is a lot of code in `readImage()`, it's fairly simple. (It would be much more complex if we were dealing with an image format that compressed the data.) It makes heavy use of `readWord()`, a utility method that we'll discuss next; `readWord()` returns a word of ASCII text as a string.

`readImage()` starts by converting the `InputStream` into a `DataInputStream`. It uses `readWord()` to get the first word from the stream. This should be either "P6" or "P3", depending on whether the data is in binary or ASCII. It then uses `readWord()` to save the image's width and height and the maximum value of any color component. Next, it reads the color data into the `store[][]` array. The ASCII case is simple because we can use `readWord()` to read ASCII words conveniently; we read red, green, and blue words, convert them into `ints`, and pack the three into one element (one pixel) of `store[][]`. For binary data, we read an entire scan line into the byte array `row[]`, using `readFully()`; then we start a loop that packs this scan line into one row of `store[][]`. A little additional complexity is in the inner loop because we must keep track of two arrays (`row[]` and `store[][]`). We read red, green, and blue components from `row[]`, converting Java's signed bytes to unsigned data by adding 256 to any negative values; finally, we pack these components into one element of `store[][]`.

```
/* readWord returns a word of text from stream          */
/* Ignores PPM comment lines.                           */
/* word defined to be something wrapped by whitespace   */
    private String readWord (InputStream is) throws IOException {
        StringBuffer buf = new StringBuffer();
        int b;
        do {// get rid of leading whitespace
            if ((b=is.read()) == -1)
                throw new EOFException();
            if ((char)b == '#') { // read to end of line - ppm comment
                DataInputStream dis = new DataInputStream (is);
                dis.readLine();
                b = ' ';  // ensure more reading
            }
        }while (Character.isSpace ((char)b));
        do {
            buf.append ((char)(b));
            if ((b=is.read()) == -1)
```

```
                throw new EOFException();
        } while (!Character.isSpace ((char)b));  // reads first space
        return buf.toString();
    }
}
```

`readWord()` is a utility method that reads one ASCII word from an `InputStream`. A word is a sequence of characters that aren't spaces; space characters include newlines and tabs in addition to spaces. This method also throws out any comments (anything between # and the end of the line). It collects the characters into a `StringBuffer`, converting the `StringBuffer` into a `String` when it returns.

Example 12.5: PPMImageDecoder Test Program

```
import java.awt.Graphics;
import java.awt.Color;
import java.awt.image.ImageConsumer;
import java.awt.Image;
import java.awt.MediaTracker;
import java.net.URL;
import java.net.MalformedURLException;
import java.io.InputStream;
import java.io.IOException;
import java.applet.Applet;
public class ppmViewer extends Applet {
    Image image = null;
    public void init () {
        try {
            String file = getParameter ("file");
            if (file != null) {
                URL imageurl = new URL (getDocumentBase(), file);
                InputStream is = imageurl.openStream();
                PPMImageDecoder ppm = new PPMImageDecoder ();
                ppm.readImage (is);
                image = createImage (ppm);
                repaint();
            }
        } catch (MalformedURLException me) {
            System.out.println ("Bad URL");
         } catch (IOException io) {
            System.out.println ("Bad File");
        }
    }
    public void paint (Graphics g) {
        g.drawImage (image, 0, 0, this);
    }
}
```

The applet we use to test our `ImageProducer` is very simple. It creates a URL that points to an appropriate PPM file and gets an `InputStream` from that URL. It then creates an instance of our `PPMImageDecoder`; calls `readImage()` to load the image and generate pixel data; and finally, calls `createImage()` with our `ImageProducer` as an argument to create an `Image` object, which we draw in `paint()`.

PixelGrabber

The `PixelGrabber` class is a utility for converting an image into an array of pixels. This is useful in many situations. If you are writing a drawing utility that lets users create their own graphics, you probably want some way to save a drawing to a file. Likewise, if you're implementing a shared whiteboard, you'll want some way to transmit images across the Net. If you're doing some kind of image processing, you may want to read and alter individual pixels in an image. The `PixelGrabber` class is an `ImageConsumer` that can capture a subset of the current pixels of an `Image`. Once you have the pixels, you can easily save the image in a file, send it across the Net, or work with individual points in the array. To recreate the `Image` (or a modified version), you can pass the pixel array to a `MemoryImageSource`.

Prior to Java 1.1, `PixelGrabber` saves an array of pixels but doesn't save the image's width and height--that's your responsibility. You may want to put the width and height in the first two elements of the pixel array and use an offset of 2 when you store (or reproduce) the image.

Starting with Java 1.1, the grabbing process changes in several ways. You can ask the `PixelGrabber` for the image's size or color model. You can grab pixels asynchronously and abort the grabbing process before it is completed. Finally, you don't have to preallocate the pixel data array. Constructors

public PixelGrabber (ImageProducer ip, int x, int y, int width, int height, int pixels[], int offset, int scansize)

> The first `PixelGrabber` constructor creates a new `PixelGrabber` instance. The `PixelGrabber` uses `ImageProducer ip` to store the unscaled cropped rectangle at position (`x`, `y`) of size `width` x `height` into the `pixels` array, starting at `offset` within `pixels`, and each row starting at increments of `scansize` from that.
>
> As shown in Figure 12.5, the position (`x1`, `y1`) would be stored in `pixels[]` at position `(y1 - y) * scansize + (x1 - x) + offset.` Calling `grabPixels()` starts the process of writing pixels into the array.
>
> The `ColorModel` for the pixels copied into the array is always the default RGB model: that is, 32 bits per pixel, with 8 bits for alpha, red, green, and blue components.

public PixelGrabber (Image image, int x, int y, int width, int height, int pixels[], int offset, int scansize)

> This version of the `PixelGrabber` constructor gets the `ImageProducer` of the `Image image` through `getSource()`; it then calls the previous constructor to create the `PixelGrabber`.

public PixelGrabber (Image image, int x, int y, int width, int height, boolean forceRGB) ★

> This version of the constructor does not require you to preallocate the pixel array and lets you preserve the color model of the original image. If `forceRGB` is `true`, the pixels of `image` are converted to the default RGB model when grabbed. If `forceRGB` is `false` and all the pixels of image use one `ColorModel`, the original color model of `image` is preserved.
>
> As with the other constructors, the `x`, `y`, `width`, and `height` values define the bounding box to grab. However,

there's one special case to consider. Setting `width` or `height` to -1 tells the `PixelGrabber` to take the `width` and `height` from the image itself. In this case, the grabber stores all the pixels below and to the right of the point (`x`, `y`). If (`x`, `y`) is outside of the image, you get an empty array.

Once the pixels have been grabbed, you get the pixel data via the `getPixels()` method described in "Other methods." To get the `ColorModel`, see the `getColorModel()` method.

ImageConsumer interface methods

public void setDimensions (int width, int height)

In Java 1.0, the `setDimensions()` method of `PixelGrabber` ignores the `width` and `height`, since this was set by the constructor.

With Java 1.1, `setDimensions()` is called by the image producer to give it the dimensions of the original image. This is how the `PixelGrabber` finds out the image's size if the constructor specified -1 for the image's width or height.

public void setHints (int hints)

The `setHints()` method ignores the `hints`.

public void setProperties (Hashtable properties)

The `setProperties()` method ignores the `properties`.

public void setColorModel (ColorModel model)

The `setColorModel()` method ignores the `model`.

public void setPixels (int x, int y, int w, int h, ColorModel model, byte pixels[], int offset, int scansize)

The `setPixels()` method is called by the `ImageProducer` to deliver pixel data for some image. If the pixels fall within the portion of the image that the `PixelGrabber` is interested in, they are stored within the array passed to the `PixelGrabber` constructor. If necessary, the `ColorModel` is used to convert each pixel from its original representation to the default RGB representation. This method is called when each pixel coming from the image producer is represented by a byte.

public void setPixels (int x, int y, int w, int h, ColorModel model, int pixels[], int offset, int scansize)

The second `setPixels()` method is almost identical to the first; it is used when each pixel coming from the image producer is represented by an `int`.

public synchronized void imageComplete (int status)

The `imageComplete()` method uses `status` to determine if the pixels were successfully delivered. The `PixelGrabber` then notifies anyone waiting for the pixels from a `grabPixels()` call.

Grabbing methods

public synchronized boolean grabPixels (long ms) throws InterruptedException

> The `grabPixels()` method starts storing pixel data from the image. It doesn't return until all pixels have been loaded into the pixels array or until `ms` milliseconds have passed. The return value is `true` if all pixels were successfully acquired. Otherwise, it returns `false` for the abort, error, or timeout condition encountered. The exception `InterruptedException` is thrown if another thread interrupts this one while waiting for pixel data.

public boolean grabPixels () throws InterruptedException

> This `grabPixels()` method starts storing pixel data from the image. It doesn't return until all pixels have been loaded into the pixels array. The return value is `true` if all pixels were successfully acquired. It returns `false` if it encountered an abort or error condition. The exception `InterruptedException` is thrown if another thread interrupts this one while waiting for pixel data.

public synchronized void startGrabbing() ★

> The `startGrabbing()` method provides an asynchronous means of grabbing the pixels. This method returns immediately; it does not block like the `grabPixels()` methods described previously. To find out when the `PixelGrabber` has finished, call `getStatus()`.

public synchronized void abortGrabbing() ★

> The `abortGrabbing()` method allows you to stop grabbing pixel data from the image. If a thread is waiting for pixel data from a `grabPixels()` call, it is interrupted and `grabPixels()` throws an `InterruptedException`.

Other methods

public synchronized int getStatus() ★
public synchronized int status () ☆

> Call the `getStatus()` method to find out whether a `PixelGrabber` succeeded in grabbing the pixels you want. The return value is a set of `ImageObserver` flags ORed together. `ALLBITS` and `FRAMEBITS` indicate success; which of the two you get depends on how the image was created. `ABORT` and `ERROR` indicate that problems occurred while the image was being produced.
>
> `status()`is the Java 1.0 name for this method.

public synchronized int getWidth() ★

> The `getWidth()` method reports the width of the image data stored in the destination buffer. If you set width to -1 when you called the `PixelGrabber` constructor, this information will be available only after the grabber has received the information from the image producer (`setDimensions()`). If the width is not available yet, `getWidth()` returns -1.

The width of the resulting image depends on several factors. If you specified the width explicitly in the constructor, the resulting image has that width, no questions asked--even if the position at which you start grabbing is outside the image. If you specified -1 for the width, the resulting width will be the difference between the `x` position at which you start grabbing (set in the constructor) and the actual image width; for example, if you start grabbing at `x`=50 and the original image width is 100, the width of the resulting image is 50. If `x` falls outside the image, the resulting width is 0.

public synchronized int getHeight() ★

The `getHeight()` method reports the height of the image data stored in the destination buffer. If you set height to -1 when you called the `PixelGrabber` constructor, this information will be available only after the grabber has received the information from the image producer (`setDimensions()`). If the height is not available yet, `getHeight()` returns -1.

The height of the resulting image depends on several factors. If you specified the height explicitly in the constructor, the resulting image has that height, no questions asked--even if the position at which you start grabbing is outside the image. If you specified -1 for the height, the resulting height will be the difference between the `y` position at which you start grabbing (set in the constructor) and the actual image height; for example, if you start grabbing at `y`=50 and the original image height is 100, the height of the resulting image is 50. If `y` falls outside the image, the resulting height is 0.

public synchronized Object getPixels() ★

The `getPixels()` method returns an array of pixel data. If you passed a pixel array to the constructor, you get back your original array object, with the data filled in. If, however, the array was not previously allocated, you get back a new array. The size of this array depends on the image you are grabbing and the portion of that image you want. If size and image format are not known yet, this method returns `null`. If the `PixelGrabber` is still grabbing pixels, this method returns an array that may change based upon the rest of the image. The type of the array you get is either `int[]` or `byte[]`, depending on the color model of the image. To find out if the `PixelGrabber` has finished, call `getStatus()`.

public synchronized ColorModel getColorModel() ★

The `getColorModel()` method returns the color model of the image. This could be the default RGB `ColorModel` if a pixel buffer was explicitly provided, `null` if the color model is not known yet, or a varying color model until all the pixel data has been grabbed. After all the pixels have been grabbed, `getColorModel()` returns the actual color model used for the `getPixels()`array. It is best to wait until grabbing has finished before you ask for the `ColorModel`; to find out, call `getStatus()`.

Using PixelGrabber to modify an image

You can modify images by combining a `PixelGrabber` with `MemoryImageSource`. Use `getImage()` to load an image from the Net; then use `PixelGrabber` to convert the image into an array. Modify the data in the array any way you please; then use `MemoryImageSource` as an image producer to display the new image.

Example 12.6 demonstrates the use of the `PixelGrabber` and `MemoryImageSource` to rotate, flip, and mirror an image. (We could also do the rotations with a subclass of `ImageFilter`, which we will discuss next.) The output is shown in Figure 12.6. When working with an image that is loaded from a local disk or the network, remember to wait until the image is loaded before grabbing its pixels. In this example, we use a `MediaTracker` to wait for the image to load.

Example 12.6: Flip Source

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
public class flip extends Applet {
    Image i, j, k, l;
    public void init () {
        MediaTracker mt = new MediaTracker (this);
        i = getImage (getDocumentBase(), "ora-icon.gif");
        mt.addImage (i, 0);
    try {
        mt.waitForAll();
        int width = i.getWidth(this);
        int height = i.getHeight(this);
        int pixels[] = new int [width * height];
        PixelGrabber pg = new PixelGrabber
        (i, 0, 0, width, height, pixels, 0, width);
        if (pg.grabPixels() && ((pg.status() &
            ImageObserver.ALLBITS) !=0)) {
            j = createImage (new MemoryImageSource (width, height,
                     rowFlipPixels (pixels, width, height), 0, width));
            k = createImage (new MemoryImageSource (width, height,
                 colFlipPixels (pixels, width, height), 0, width));
            l = createImage (new MemoryImageSource (height, width,
                 rot90Pixels (pixels, width, height), 0, height));
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

Figure 12.6: Flip output

[Graphic: Figure 12-6]

The `try` block in Example 12.6 does all the interesting work. It uses a `PixelGrabber` to grab the entire image into the array `pixels[]`. After calling `grabPixels()`, it checks the `PixelGrabber` status to make sure that the image was stored correctly. It then generates three new images based on the first by calling `createImage()` with a `MemoryImageSource` object as an argument. Instead of using the original array, the `MemoryImageSource` objects call several utility methods to manipulate the array: `rowFlipPixels()`, `colFlipPixels()`, and `rot90Pixels()`. These methods all return integer arrays.

```
public void paint (Graphics g) {
    g.drawImage (i, 10, 10, this); // regular
    if (j != null)
        g.drawImage (j, 150, 10, this); // rowFlip
    if (k != null)
        g.drawImage (k, 10, 60, this); // colFlip
    if (l != null)
        g.drawImage (l, 150, 60, this); // rot90
}
private int[] rowFlipPixels (int pixels[], int width, int height) {
    int newPixels[] = null;
    if ((width*height) == pixels.length) {
        newPixels = new int [width*height];
        int newIndex=0;
        for (int y=height-1;y>=0;y--)
            for (int x=width-1;x>=0;x--)
                newPixels[newIndex++]=pixels[y*width+x];
    }
    return newPixels;
}
```

`rowFlipPixels()` creates a mirror image of the original, flipped horizontally. It is nothing more than a nested loop that copies the original array into a new array.

```
    private int[] colFlipPixels (int pixels[], int width, int height) {
        ...
    }
private int[] rot90Pixels (int pixels[], int width, int height) {
        ...
    }
}
```

`colFlipPixels()` and `rot90Pixels()` are fundamentally similar to `rowFlipPixels()`; they just copy the original pixel array into another array, and return the result. `colFlipPixels()` generates a vertical mirror image; `rot90Pixels()` rotates the image by 90 degrees counterclockwise. Grabbing data asynchronously

To demonstrate the new methods introduced by Java 1.1 for `PixelGrabber`, the following program grabs the pixels and reports information about the original image on mouse clicks. It takes its data from the image used in Figure 12.6.

```
// Java 1.1 only
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
import java.awt.event.*;
public class grab extends Applet {
    Image i;
    PixelGrabber pg;
    public void init () {
        i = getImage (getDocumentBase(), "ora-icon.gif");
        pg  = new PixelGrabber (i, 0, 0, -1, -1, false);
        pg.startGrabbing();
        enableEvents (AWTEvent.MOUSE_EVENT_MASK);
    }
    public void paint (Graphics g) {
        g.drawImage (i, 10, 10, this);
    }
    protected void processMouseEvent(MouseEvent e) {
        if (e.getID() == MouseEvent.MOUSE_CLICKED) {
            System.out.println ("Status: " + pg.getStatus());
            System.out.println ("Width:  " + pg.getWidth());
            System.out.println ("Height: " + pg.getHeight());
            System.out.println ("Pixels: " +
               (pg.getPixels() instanceof byte[] ? "bytes" : "ints"));
            System.out.println ("Model:  " + pg.getColorModel());
        }
        super.processMouseEvent (e);
    }
}
```

This applet creates a `PixelGrabber` without specifying an array, then starts grabbing pixels. The grabber allocates its own array, but we never bother to ask for it since we don't do anything with the data itself: we only report the grabber's status. (If we wanted the data, we'd call `getPixels()`.) Sample output from a single mouse click, after the image loaded, would appear something like the following:

```
Status: 27
```

```
Width:  120
Height: 38
Pixels: bytes
Model:  java.awt.image.IndexColorModel@1ed34
```

You need to convert the status value manually to the corresponding meaning by looking up the status codes in `ImageObserver`. The value 27 indicates that the 1, 2, 8, and 16 flags are set, which translates to the `WIDTH`, `HEIGHT`, `SOMEBITS`, and `FRAMEBITS` flags, respectively.

PREVIOUS | HOME | NEXT
ImageProducer | BOOK INDEX | ImageFilter

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

12.5 ImageFilter

Image filters provide another way to modify images. An `ImageFilter` is used in conjunction with a `FilteredImageSource` object. The `ImageFilter`, which implements `ImageConsumer` (and `Cloneable`), receives data from an `ImageProducer` and modifies it; the `FilteredImageSource`, which implements `ImageProducer`, sends the modified data to the new consumer. As Figure 12.1 shows, an image filter sits between the original `ImageProducer` and the ultimate `ImageConsumer`.

The `ImageFilter` class implements a "null" filter that does nothing to the image. To modify an image, you must use a subclass of `ImageFilter`, by either writing one yourself or using a subclass provided with AWT, like the `CropImageFilter`. Another `ImageFilter` subclass provided with AWT is the `RGBImageFilter`; it is useful for filtering an image on the basis of a pixel's color. Unlike the `CropImageFilter`, `RGBImageFilter` is an abstract class, so you need to create your own subclass to use it. Java 1.1 introduces two more image filters, `AreaAveragingScaleFilter` and `ReplicateScaleFilter`. Other filters must be created by subclassing `ImageFilter` and providing the necessary methods to modify the image as necessary.

`ImageFilter`s tend to work on a pixel-by-pixel basis, so large `Image` objects can take a considerable amount of time to filter, depending on the complexity of the filtering algorithm. In the simplest case, filters generate new pixels based upon the color value and location of the original pixel. Such filters can start delivering data before they have loaded the entire image. More complex filters may use internal buffers to store an intermediate copy of the image so the filter can use adjacent pixel values to smooth or blend pixels together. These filters may need to load the entire image before they can deliver any data to the ultimate consumer.

To use an `ImageFilter`, you pass it to the `FilteredImageSource` constructor, which serves as an `ImageProducer` to pass the new pixels to their consumer. The following code runs the image *logo.jpg* through an image filter, `SomeImageFilter`, to produce a new image. The constructor for `SomeImageFilter` is called within the constructor for `FilteredImageSource`, which in turn is the only argument to `createImage()`.

```
Image image = getImage (new URL (
     "http://www.ora.com/images/logo.jpg"));
Image newOne = createImage (new FilteredImageSource (image.getSource(),
                               new SomeImageFilter()));
```

ImageFilter Methods

Variables

protected ImageConsumer consumer;

> The actual `ImageConsumer` for the image. It is initialized automatically for you by the `getFilterInstance()` method.

Constructor

public ImageFilter ()

> The only constructor for `ImageFilter` is the default one, which takes no arguments. Subclasses can provide their own constructors if they need additional information.

ImageConsumer interface methods

public void setDimensions (int width, int height)

> The `setDimensions()` method of `ImageFilter` is called when the `width` and `height` of the original image are known. It calls `consumer.setDimensions()` to tell the next consumer the dimensions of the filtered image. If you subclass `ImageFilter` and your filter changes the image's dimensions, you should override this method to compute and report the new dimensions.

public void setProperties (Hashtable properties)

> The `setProperties()` method is called to provide the image filter with the property list for the original image. The image filter adds the property `filters` to the list and passes it along to the next consumer. The value given for the `filters` property is the result of the image filter's `toString()` method; that is, the `String` representation of the current filter. If `filters` is already set, information about this `ImageFilter` is appended to the end. Subclasses of `ImageFilter` may add other properties.

public void setColorModel (ColorModel model)

> The `setColorModel()` method is called to give the `ImageFilter` the color model used for most of the pixels in the original image. It passes this color model on to the next consumer. Subclasses may override this method if they change the color model.

public void setHints (int hints)

> The `setHints()` method is called to give the `ImageFilter` hints about how the producer will deliver pixels. This method passes the same set of hints to the next consumer. Subclasses must override this method if they need to provide different hints; for example, if they are delivering pixels in a different order.

public void setPixels (int x, int y, int width, int height, ColorModel model, byte pixels[], int offset, int scansize)
public void setPixels (int x, int y, int width, int height, ColorModel model, int pixels[], int offset, int scansize)

> The `setPixels()` method receives pixel data from the `ImageProducer` and passes all the information on to the `ImageConsumer`. (`x`, `y`) is the top left corner of the bounding rectangle for the pixels. The bounding

rectangle has size `width` x `height`. The `ColorModel` for the new image is `model`. `pixels` is the byte or integer array of the pixel information, starting at `offset` (usually 0), with scan lines of size `scansize` (usually `width`).

public void imageComplete (int status)

The `imageComplete()` method receives the completion `status` from the `ImageProducer` and passes it along to the `ImageConsumer`.

If you subclass `ImageFilter`, you will probably override the `setPixels()` methods. For simple filters, you may be able to modify the pixel array and deliver the result to `consumer.setPixels()` immediately. For more complex filters, you will have to build a buffer containing the entire image; in this case, the call to `imageComplete()` will probably trigger filtering and pixel delivery.

Cloneable interface methods

public Object clone ()

The `clone()` method creates a clone of the `ImageFilter`. The `getFilterInstance()` function uses this method to create a copy of the `ImageFilter`. Cloning allows the same filter instance to be used with multiple `Image` objects.

Other methods

public ImageFilter getFilterInstance (ImageConsumer ic)

`FilteredImageSource` calls `getFilterInstance()` to register `ic` as the `ImageConsumer` for an instance of this filter; to do so, it sets the instance variable `consumer`. In effect, this method inserts the `ImageFilter` between the image's producer and the consumer. You have to override this method only if there are special requirements for the insertion process. This default implementation just calls `clone()`.

public void resendTopDownLeftRight (ImageProducer ip)

The `resendTopDownLeftRight()` method tells the `ImageProducer` `ip` to try to resend the image data in the top-down, left-to-right order. If you override this method and your `ImageFilter` has saved the image data internally, you may want your `ImageFilter` to resend the data itself, rather than asking the `ImageProducer`. Otherwise, your subclass may ignore the request or pass it along to the `ImageProducer` `ip`.

Subclassing ImageFilter: A blurring filter

When you subclass `ImageFilter`, there are very few restrictions on what you can do. We will create a few subclasses that show some of the possibilities. This `ImageFilter` generates a new pixel by averaging the pixels around it. The result is a blurred version of the original. To implement this filter, we have to save all the pixel data into a buffer; we can't start delivering pixels until the entire image is in hand. Therefore, we override `setPixels()` to build the buffer; we override `imageComplete()` to produce the new pixels and deliver them.

Before looking at the code, here are a few hints about how the filter works; it uses a few tricks that may be helpful in other situations. We need to provide two versions of `setPixels()`: one for integer arrays, and the other for byte arrays. To avoid duplicating code, both versions call a single method, `setThePixels()`, which takes an `Object` as an argument, instead of a pixel array; thus it can be called with either kind of pixel array. Within the method, we check whether the pixels argument is an instance of `byte[]` or `int[]`. The body of this method uses another trick: when it reads the `byte[]` version of the pixel array, it ANDs the value with 0xff. This prevents the byte value, which is signed, from being converted to a negative `int` when used as an argument to `cm.getRGB()`.

The logic inside of `imageComplete()` gets a bit hairy. This method does the actual filtering, after all the data has arrived. Its job is basically simple: compute an average value of the pixel and the eight pixels surrounding it (i.e., a 3x3 rectangle with the current pixel in the center). The problem lies in taking care of the edge conditions. We don't always want to average nine pixels; in fact, we may want to average as few as four. The `if` statements figure out which surrounding pixels should be included in the average. The pixels we care about are placed in `sumArray[]`, which has nine elements. We keep track of the number of elements that have been saved in the variable `sumIndex` and use a helper method, `avgPixels()`, to compute the average. The code might be a little cleaner if we used a `Vector`, which automatically counts the number of elements it contains, but it would probably be much slower.

Example 12.7 shows the code for the blurring filter.

Example 12.7: Blur Filter Source

```
import java.awt.*;
import java.awt.image.*;
public class BlurFilter extends ImageFilter {
    private int savedWidth, savedHeight, savedPixels[];
    private static ColorModel defaultCM = ColorModel.getRGBdefault();
    public void setDimensions (int width, int height) {
        savedWidth=width;
        savedHeight=height;
        savedPixels=new int [width*height];
        consumer.setDimensions (width, height);
    }
```

We override `setDimensions()` to save the original image's height and width, which we use later.

```
    public void setColorModel (ColorModel model) {
    // Change color model to model you are generating
        consumer.setColorModel (defaultCM);
    }
    public void setHints (int hintflags) {
    // Set new hints, but preserve SINGLEFRAME setting
        consumer.setHints (TOPDOWNLEFTRIGHT | COMPLETESCANLINES |
                           SINGLEPASS | (hintflags & SINGLEFRAME));
    }
```

This filter always generates pixels in the same order, so it sends the hint flags `TOPDOWNLEFTRIGHT`, `COMPLETESCANLINES`, and `SINGLEPASS` to the consumer, regardless of what the image producer says. It sends the `SINGLEFRAME` hint only if the producer has sent it.

```
    private void setThePixels (int x, int y, int width, int height,
            ColorModel cm, Object pixels, int offset, int scansize) {
        int sourceOffset = offset;
        int destinationOffset = y * savedWidth + x;
        boolean bytearray = (pixels instanceof byte[]);
        for (int yy=0;yy<height;yy++) {
            for (int xx=0;xx<width;xx++)
                if (bytearray)
                    savedPixels[destinationOffset++]=
                        cm.getRGB(((byte[])pixels)[sourceOffset++]&0xff);
                else
                    savedPixels[destinationOffset++]=
                        cm.getRGB(((int[])pixels)[sourceOffset++]);
            sourceOffset += (scansize - width);
            destinationOffset += (savedWidth - width);
        }
    }
```

`setThePixels()` saves the pixel data for the image in the array `savedPixels[]`. Both versions of `setPixels()` call this method. It doesn't pass the pixels along to the image consumer, since this filter can't process the pixels until the entire image is available.

```
public void setPixels (int x, int y, int width, int height,
       ColorModel cm, byte pixels[], int offset, int scansize) {
    setThePixels (x, y, width, height, cm, pixels, offset, scansize);
}
public void setPixels (int x, int y, int width, int height,
       ColorModel cm, int pixels[], int offset, int scansize) {
    setThePixels (x, y, width, height, cm, pixels, offset, scansize);
}
public void imageComplete (int status) {
    if ((status == IMAGEABORTED) || (status == IMAGEERROR)) {
        consumer.imageComplete (status);
        return;
    } else {
        int pixels[] = new int [savedWidth];
        int position, sumArray[], sumIndex;
        sumArray = new int [9]; // maxsize - vs. Vector for performance
        for (int yy=0;yy<savedHeight;yy++) {
            position=0;
            int start = yy * savedWidth;
            for (int xx=0;xx<savedWidth;xx++) {
                sumIndex=0;
                                                                 //  xx     yy
                sumArray[sumIndex++] = savedPixels[start+xx];  // center center
                if (yy != (savedHeight-1))                       // center bottom
                    sumArray[sumIndex++] = savedPixels[start+xx+savedWidth];
                if (yy != 0)                                   // center top
                    sumArray[sumIndex++] = savedPixels[start+xx-savedWidth];
                if (xx != (savedWidth-1))                      // right  center
                    sumArray[sumIndex++] = savedPixels[start+xx+1];
                if (xx != 0)                                   // left   center
                    sumArray[sumIndex++] = savedPixels[start+xx-1];
                if ((yy != 0) && (xx != 0))                    // left   top
                    sumArray[sumIndex++] = savedPixels[start+xx-savedWidth-1];
                if ((yy != (savedHeight-1)) && (xx != (savedWidth-1)))
                    //                                           right  bottom
                    sumArray[sumIndex++] = savedPixels[start+xx+savedWidth+1];
                if ((yy != 0) && (xx != (savedWidth-1)))       //right  top
                    sumArray[sumIndex++] = savedPixels[start+xx-savedWidth+1];
                if ((yy != (savedHeight-1)) && (xx != 0))      //left   bottom
                    sumArray[sumIndex++] = savedPixels[start+xx+savedWidth-1];
                pixels[position++] = avgPixels(sumArray, sumIndex);
            }
            consumer.setPixels (0, yy, savedWidth, 1, defaultCM,
                                pixels, 0, savedWidth);
        }
        consumer.imageComplete (status);
    }
}
```

`imageComplete()` does the actual filtering after the pixels have been delivered and saved. If the producer reports that an error occurred, this method passes the error flags to the consumer and returns. If not, it builds a new array, `pixels[]`, which contains the filtered pixels, and delivers these to the consumer.

Previously, we gave an overview of how the filtering process works. Here are some details. (`xx`, `yy`) represents the current point's `x` and `y` coordinates. The point (`xx`, `yy`) must always fall within the image; otherwise, our loops are constructed incorrectly. Therefore, we can copy (`xx`, `yy`) into the `sumArray[]` for averaging without any tests. For the point's eight neighbors, we check whether the neighbor falls in the image; if so, we add it to `sumArray[]`. For example, the point just below (`xx`, `yy`) is at the bottom center of the 3x3 rectangle of points we are averaging. We know that `xx` falls within the image; `yy` falls within the image if it doesn't equal `savedHeight-1`. We do similar tests for the other points.

Even though we're working with a rectangular image, our arrays are all one-dimensional so we have to convert a coordinate pair (`xx`, `yy`) into a single array index. To help us do the bookkeeping, we use the local variable `start` to keep track of the start of the current scan line. Then `start` + `xx` is the current point; `start` + `xx` + `savedWidth` is the point immediately below; `start` + `xx` + `savedWidth-1` is the point below and to the left; and so on.

`avgPixels()` is our helper method for computing the average value that we assign to the new pixel. For each pixel in the `pixels[]` array, it extracts the red, blue, green, and alpha components; averages them separately, and returns a new ARGB value.

```
    private int avgPixels (int pixels[], int size) {
        float redSum=0, greenSum=0, blueSum=0, alphaSum=0;
        for (int i=0;i<size;i++)
```

```
            try {
                int pixel = pixels[i];
                redSum   += defaultCM.getRed   (pixel);
                greenSum += defaultCM.getGreen (pixel);
                blueSum  += defaultCM.getBlue  (pixel);
                alphaSum += defaultCM.getAlpha (pixel);
            } catch (ArrayIndexOutOfBoundsException e) {
                System.out.println ("Ooops");
            }
        int redAvg   = (int)(redSum   / size);
        int greenAvg = (int)(greenSum / size);
        int blueAvg  = (int)(blueSum  / size);
        int alphaAvg = (int)(alphaSum / size);
        return ((0xff << 24) | (redAvg << 16) |
                (greenAvg << 8)  | (blueAvg << 0));
    }
}
```

Producing many images from one: dynamic ImageFilter

The `ImageFilter` framework is flexible enough to allow you to return a sequence of images based on an original. You can send back one frame at a time, calling the following when you are finished with each frame:

```
consumer.imageComplete(ImageConsumer.SINGLEFRAMEDONE);
```

After you have generated all the frames, you can tell the consumer that the sequence is finished with the `STATICIMAGEDONE` constant. In fact, this is exactly what the new animation capabilities of `MemoryImageSource` use.

In Example 12.8, the `DynamicFilter` lets the consumer display an image. After the image has been displayed, the filter gradually overwrites the image with a specified color by sending additional image frames. The end result is a solid colored rectangle. Not too exciting, but it's easy to imagine interesting extensions: you could use this technique to implement a fade from one image into another. The key points to understand are:

- This filter does not override `setPixels()`, so it is extremely fast. In this case, we want the original image to reach the consumer, and there is no reason to save the image in a buffer.

- Filtering takes place in the image-fetching thread, so it is safe to put the filter-processing thread to sleep if the image is coming from disk. If the image is in memory, filtering should not sleep because there will be a noticeable performance lag in your program if it does. The `DynamicFilter` class has a delay parameter to its constructor that lets you control this behavior.

- This subclass overrides `setDimensions()` to save the image's dimensions for its own use. It needs to override `setHints()` because it sends pixels to the consumer in a nonstandard order: it sends the original image, then goes back and starts sending overlays. Likewise, this subclass overrides `resendTopDownLeftRight()` to do nothing because there is no way the original `ImageProducer` can replace all the changes with the original `Image`.

- `imageComplete()` is where all the fun happens. Take a special look at the status flags that are returned.

Example 12.8: DynamicFilter Source

```
import java.awt.*;
import java.awt.image.*;
public class DynamicFilter extends ImageFilter {
    Color overlapColor;
    int   delay;
    int   imageWidth;
    int   imageHeight;
    int   iterations;
    DynamicFilter (int delay, int iterations, Color color) {
        this.delay      = delay;
        this.iterations = iterations;
        overlapColor    = color;
    }
    public void setDimensions (int width, int height) {
        imageWidth  = width;
        imageHeight = height;
        consumer.setDimensions (width, height);
    }
    public void setHints (int hints) {
        consumer.setHints (ImageConsumer.RANDOMPIXELORDER);
    }
    public void resendTopDownLeftRight (ImageProducer ip) {
    }
    public void imageComplete (int status) {
        if ((status == IMAGEERROR) || (status == IMAGEABORTED)) {
            consumer.imageComplete (status);
            return;
        } else {
            int xWidth = imageWidth / iterations;
            if (xWidth <= 0)
                xWidth = 1;
            int newPixels[] = new int [xWidth*imageHeight];
            int iColor = overlapColor.getRGB();
            for (int x=0;x<(xWidth*imageHeight);x++)
                newPixels[x] = iColor;
            int t=0;
            for (;t<(imageWidth-xWidth);t+=xWidth) {
                consumer.setPixels(t, 0, xWidth, imageHeight,
                        ColorModel.getRGBdefault(), newPixels, 0, xWidth);
                consumer.imageComplete (ImageConsumer.SINGLEFRAMEDONE);
                try {
                    Thread.sleep (delay);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            int left = imageWidth-t;
            if (left > 0) {
                consumer.setPixels(imageWidth-left, 0, left, imageHeight,
                        ColorModel.getRGBdefault(), newPixels, 0, xWidth);
                consumer.imageComplete (ImageConsumer.SINGLEFRAMEDONE);
            }
            consumer.imageComplete (STATICIMAGEDONE);
        }
    }
}
```

The `DynamicFilter` relies on the default `setPixels()` method to send the original image to the consumer. When the original image has been transferred, the image producer calls this filter's `imageComplete()` method, which does the real work. Instead of relaying the completion status to the consumer, `imageComplete()` starts generating its own data: solid rectangles that are all in the `overlapColor` specified in the constructor. It sends these rectangles to the consumer by calling `consumer.setPixels()`. After each rectangle, it calls `consumer.imageComplete()` with the `SINGLEFRAMEDONE` flag, meaning that it has just finished one frame of a multi-frame sequence. When the rectangles have completely covered the image, the method `imageComplete()` finally notifies the consumer that the entire image sequence has been transferred by sending the `STATICIMAGEDONE` flag.

The following code is a simple applet that uses this image filter to produce a new image:

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
public class DynamicImages extends Applet {
    Image i, j;
    public void init () {
        i = getImage (getDocumentBase(), "rosey.jpg");
        j = createImage (new FilteredImageSource (i.getSource(),
                        new DynamicFilter(250, 10, Color.red)));
    }
    public void paint (Graphics g) {
        g.drawImage (j, 10, 10, this);
    }
}
```

One final curiosity: the `DynamicFilter` doesn't make any assumptions about the color model used for the original image. It sends its overlays with the default RGB color model. Therefore, this is one case in which an `ImageConsumer` may see calls to `setPixels()` that use different color models.

RGBImageFilter

`RGBImageFilter` is an abstract subclass of `ImageFilter` that provides a shortcut for building the most common kind of image filters: filters that independently modify the pixels of an existing image, based only on the pixel's position and color. Because `RGBImageFilter` is an abstract class, you must subclass it before you can do anything. The only method your subclass must provide is `filterRGB()`, which produces a new pixel value based on the original pixel and its location. A handful of additional methods are in this class; most of them provide the behind-the-scenes framework for funneling each pixel through the `filterRGB()` method.

If the filtering algorithm you are using does not rely on pixel position (i.e., the new pixel is based only on the old pixel's color), AWT can apply an optimization for images that use an `IndexColorModel`: rather than filtering individual pixels, it can filter the image's color map. In order to tell AWT that this optimization is okay, add a constructor to the class definition that sets the `canFilterIndexColorModel` variable to `true`. If `canFilterIndexColorModel` is `false` (the default) and an `IndexColorModel` image is sent through the filter, nothing happens to the image. Variables

protected boolean canFilterIndexColorModel

> Setting the `canFilterIndexColorModel` variable permits the `ImageFilter` to filter `IndexColorModel` images. The default value is `false`. When this variable is `false`, `IndexColorModel` images are not filtered. When this variable is `true`, the `ImageFilter` filters the colormap instead of the individual pixel values.

protected ColorModel newmodel

> The `newmodel` variable is used to store the new `ColorModel` when `canFilterIndexColorModel` is `true` and the `ColorModel` actually is of type `IndexColorModel`. Normally, you do not need to access this variable, even in subclasses.

protected ColorModel origmodel

> The `origmodel` variable stores the original color model when filtering an `IndexColorModel`. Normally, you do not need to access this variable, even in subclasses.

Constructors

public RGBImageFilter ()--called by subclass

> The only constructor for `RGBImageFilter` is the implied constructor with no parameters. In most subclasses of `RGBImageFilter`, the constructor has to initialize only the `canFilterIndexColorModel` variable.

ImageConsumer interface methods

public void setColorModel (ColorModel model)

> The `setColorModel()` method changes the `ColorModel` of the filter to `model`. If `canFilterIndexColorModel` is `true` and `model` is of type `IndexColorModel`, a filtered version of `model` is used instead.

public void setPixels (int x, int y, int w, int h, ColorModel model, byte pixels[], int off, int scansize) *public void setPixels (int x, int y, int w, int h, ColorModel model, int pixels[], int off, int scansize)*

> If necessary, the `setPixels()` method converts the `pixels` buffer to the default RGB `ColorModel` and then filters them with `filterRGBPixels()`. If `model` has already been converted, this method just passes the pixels along to the consumer's `setPixels()`.

Other methods

The only method you care about here is `filterRGB()`. All subclasses of RGBImageFilter *must* override this method. It is very difficult to imagine situations in which you would override (or even call) the other methods in this group. They are helper methods that funnel pixels through `filterRGB()`.

public void substituteColorModel (ColorModel oldModel, ColorModel newModel)

> `substituteColorModel()` is a helper method for `setColorModel()`. It initializes the protected variables of `RGBImageFilter`. The `origmodel` variable is set to `oldModel` and the `newmodel` variable is set to `newModel`.

public IndexColorModel filterIndexColorModel (IndexColorModel icm)

> `filterIndexColorModel()` is another helper method for `setColorModel()`. It runs the entire color table of `icm` through `filterRGB()` and returns the filtered `ColorModel` for use by `setColorModel()`.

public void filterRGBPixels (int x, int y, int width, int height, int pixels[], int off, int scansize)

> `filterRGBPixels()` is a helper method for `setPixels()`. It filters each element of the `pixels` buffer through `filterRGB()`, converting pixels to the default RGB `ColorModel` first. This method changes the values in the `pixels` array.

public abstract int filterRGB (int x, int y, int rgb)

> `filterRGB()` is the one method that `RGBImageFilter` subclasses must implement. The method takes the `rgb` pixel value at position (`x`, `y`) and returns the converted pixel value in the default RGB `ColorModel`. Coordinates of (-1, -1) signify that a color table entry is being filtered instead of a pixel.

A transparent image filter that extends RGBImageFilter

Creating your own `RGBImageFilter` is fairly easy. One of the more common applications for an `RGBImageFilter` is to make images transparent by setting the alpha component of each pixel. To do so, we extend the abstract `RGBImageFilter` class. The filter in Example 12.9 makes the entire image translucent, based on a percentage passed to the class constructor. Filtering is independent of position, so the constructor can set the `canFilterIndexColorModel` variable. A constructor with no arguments uses a default alpha value of 0.75.

Example 12.9: TransparentImageFilter Source

```
import java.awt.image.*;
class TransparentImageFilter extends RGBImageFilter {
    float alphaPercent;
    public TransparentImageFilter () {
        this (0.75f);
    }
    public TransparentImageFilter (float aPercent)
            throws IllegalArgumentException {
        if ((aPercent < 0.0) || (aPercent > 1.0))
            throw new IllegalArgumentException();
        alphaPercent = aPercent;
        canFilterIndexColorModel = true;
    }
    public int filterRGB (int x, int y, int rgb) {
        int a = (rgb >> 24) & 0xff;
        a *= alphaPercent;
        return ((rgb & 0x00ffffff) | (a << 24));
    }
}
```

CropImageFilter

The `CropImageFilter` is an `ImageFilter` that crops an image to a rectangular region. When used with `FilteredImageSource`, it produces a new image that consists of a portion of the original image. The cropped region must be completely within the original image. It is never necessary to subclass this class. Also, using the 10 or 11 argument version of `Graphics.drawImage()` introduced in Java 1.1 precludes the need to use this filter, unless you need to save the resulting cropped image.

If you crop an image and then send the result through a second `ImageFilter`, the pixel array received by the filter will be the size of the original `Image`, with the `offset` and `scansize` set accordingly. The `width` and `height` are set to the cropped values; the result is a smaller `Image` with the same amount of data. `CropImageFilter` keeps the full pixel array around, partially empty. Constructors

public CropImageFilter (int x, int y, int width, int height) ★

> The constructor for `CropImageFilter` specifies the rectangular area of the old image that makes up the new image. The (`x`, `y`) coordinates specify the top left corner for the cropped image; `width` and `height` must be positive or the resulting image will be empty. If the (`x`, `y`) coordinates are outside the original image area, the resulting image is empty. If (`x`, `y`) starts within the image but the rectangular area of size `width` x `height` goes beyond the original image, the part that extends outside will be black. (Remember the color black has pixel values of 0 for red, green, and blue.)

ImageConsumer interface methods

public void setProperties (Hashtable properties) ★

> The `setProperties()` method adds the `croprect` image property to the properties list. The bounding

Rectangle, specified by the (x, y) coordinates and width x height size, is associated with this property. After updating properties, this method sets the properties list of the consumer.

public void setDimensions (int width, int height) ★

The setDimensions() method of CropImageFilter ignores the width and height parameters to the function call. Instead, it relies on the size parameters in the constructor.

public void setPixels (int x, int y, int w, int h, ColorModel model, byte pixels[], int offset, int scansize) ★
public void setPixels (int x, int y, int w, int h, ColorModel model, int pixels[], int offset, int scansize) ★

These setPixels() methods check to see what portion of the pixels array falls within the cropped area and pass those pixels along.

Cropping an image with CropImageFilter

Example 12.10 uses a CropImageFilter to extract the center third of a larger image. No subclassing is needed; the CropImageFilter is complete in itself. The output is displayed in Figure 12.7.

Example 12.10: Crop Applet Source

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
public class Crop extends Applet {
    Image i, j;
    public void init () {
        MediaTracker mt = new MediaTracker (this);
        i = getImage (getDocumentBase(), "rosey.jpg");
        mt.addImage (i, 0);
        try {
            mt.waitForAll();
            int width        = i.getWidth(this);
            int height       = i. getHeight(this);
            j = createImage (new FilteredImageSource (i.getSource(),
                                new CropImageFilter (width/3, height/3,
                                                     width/3, height/3)));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    public void paint (Graphics g) {
        g.drawImage (i, 10, 10, this);                  // regular
        if (j != null) {
            g.drawImage (j, 10, 90, this);             // cropped
        }
    }
}
```

Figure 12.7: Image cropping example output.

[Graphic: Figure 12-7]

TIP:

You can use `CropImageFilter` to help improve your animation performance or just the general download time of images. Without `CropImageFilter`, you can use `Graphics.clipRect()` to clip each image of an image strip when drawing. Instead of clipping each `Image` (each time), you can use `CropImageFilter` to create a new `Image` for each cell of the strip. Or for times when an image strip is inappropriate, you can put all your images within one image file (in any order whatsoever), and use `CropImageFilter` to get each out as an `Image` .

ReplicateScaleFilter

Back in Chapter 2, *Simple Graphics* we introduced you to the `getScaledInstance()` method. This method uses a new image filter that is provided with Java 1.1. The `ReplicateScaleFilter` and its subclass, `AreaAveragingScaleFilter`, allow you to scale images before calling `drawImage()`. This can greatly speed your programs because you don't have to wait for the call to `drawImage()` before performing scaling.

The `ReplicateScaleFilter` is an `ImageFilter` that scales by duplicating or removing rows and columns. When used with `FilteredImageSource`, it produces a new image that is a scaled version of the original. As you can guess, `ReplicateScaleFilter` is very fast, but the results aren't particularly pleasing aesthetically. It is great if you want to magnify a checkerboard but not that useful if you want to scale an image of your Aunt Polly. Its subclass, `AreaAveragingScaleFilter`, implements a more time-consuming algorithm that is more suitable when image quality is a concern. Constructor

public ReplicateScaleFilter (int width, int height)

> The constructor for `ReplicateScaleFilter` specifies the size of the resulting image. If either parameter is -1, the resulting image maintains the same aspect ratio as the original image.

ImageConsumer interface methods

public void setProperties (Hashtable properties)

The `setProperties()` method adds the `rescale` image property to the properties list. The value of the rescale property is a quoted string showing the image's new width and height, in the form `` `<width>x<height>` ``, where the width and height are taken from the constructor. After updating `properties`, this method sets the properties list of the consumer.

public void setDimensions (int width, int height)

The `setDimensions()` method of `ReplicateScaleFilter` passes the new width and height from the constructor along to the consumer. If either of the constructor's parameters are negative, the size is recalculated proportionally. If both are negative, the size becomes `width` x `height`.

public void setPixels (int x, int y, int w, int h, ColorModel model, int pixels[], int offset, int scansize)
public void setPixels (int x, int y, int w, int h, ColorModel model, byte pixels[], int offset, int scansize)

The `setPixels()` method of `ReplicateScaleFilter` checks to see which rows and columns of `pixels` to pass along.

AreaAveragingScaleFilter

The `AreaAveragingScaleFilter` subclasses `ReplicateScaleFilter` to provide a better scaling algorithm. Instead of just dropping or adding rows and columns, `AreaAveragingScaleFilter` tries to blend pixel values when creating new rows and columns. The filter works by replicating rows and columns to generate an image that is a multiple of the original size. Then the image is resized back down by an algorithm that blends the pixels around each destination pixel. AreaAveragingScaleFilter methods

Because this filter subclasses `ReplicateScaleFilter`, the only methods it includes are those that override methods of `ReplicateScaleFilter`. Constructors

public AreaAveragingScaleFilter (int width, int height) ★

The constructor for `AreaAveragingScaleFilter` specifies the size of the resulting image. If either parameter is -1, the resulting image maintains the same aspect ratio as the original image.

ImageConsumer interface methods

public void setHints (int hints) ★

The `setHints()` method of `AreaAveragingScaleFilter` checks to see if some optimizations can be performed based upon the value of the `hints` parameter. If they can't, the image filter has to cache the pixel data until it receives the entire image.

public void setPixels (int x, int y, int w, int h, ColorModel model, byte pixels[], int offset, int scansize) ★
public void setPixels (int x, int y, int w, int h, ColorModel model, int pixels[], int offset, int scansize) ★

The `setPixels()` method of `AreaAveragingScaleFilter` accumulates the pixels or passes them along based upon the available hints. If `setPixels()` accumulates the pixels, this filter passes them along to the consumer when appropriate.

Cascading Filters

It is often a good idea to perform complex filtering operations by using several filters in a chain. This technique requires the system to perform several passes through the image array, so it may be slower than using a single complex filter; however, cascading filters yield code that is easier to understand and quicker to write--particularly if you already have a collection of image filters from other projects.

For example, assume you want to make a color image transparent and then render the image in black and white. The easy way to do this task is to apply a filter that converts color to a gray value and then apply the `TransparentImageFilter` we developed in Example 12.9. Using this strategy, we have to develop only one very simple filter. Example 12.11 shows the source for the `GrayImageFilter`; Example 12.12 shows the applet that applies the two filters in a daisy chain.

Example 12.11: GrayImageFilter Source

```
import java.awt.image.*;
public class GrayImageFilter extends RGBImageFilter {
    public GrayImageFilter () {
        canFilterIndexColorModel = true;
    }
    public int filterRGB (int x, int y, int rgb) {
        int gray  = (((rgb & 0xff0000) >> 16) +
                     ((rgb & 0x00ff00) >> 8) +
                     (rgb & 0x0000ff)) / 3;
        return (0xff000000 | (gray << 16) | (gray <<  8) |  gray);
    }
}
```

Example 12.12: DrawingImages Source

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
public class DrawingImages extends Applet {
    Image i, j, k, l;
    public void init () {
        i = getImage (getDocumentBase(), "rosey.jpg");
        GrayImageFilter gif = new GrayImageFilter ();
        j = createImage (new FilteredImageSource (i.getSource(), gif));
        TransparentImageFilter tf = new TransparentImageFilter (.5f);
        k = createImage (new FilteredImageSource (j.getSource(), tf));
        l = createImage (new FilteredImageSource (i.getSource(), tf));
    }
    public void paint (Graphics g) {
        g.drawImage (i, 10, 10, this);                  // regular
        g.drawImage (j, 270, 10, this);                 // gray
        g.drawImage (k, 10, 110, Color.red, this);      // gray - transparent
        g.drawImage (l, 270, 110, Color.red, this);     // transparent
    }
}
```

Granted, neither the `GrayImageFilter` or the `TransparentImageFilter` are very complex, but consider the savings you would get if you wanted to blur an image, crop it, and then render the result in grayscale. Writing a filter that does all three is not a task for the faint of heart; remember, you can't subclass `RGBImageFilter` or `CropImageFilter` because the result does not depend purely on each pixel's color and position. However, you can solve the problem easily by cascading the filters developed in this chapter.

PREVIOUS ImageConsumer | HOME BOOK INDEX | NEXT AWT Exceptions and Errors

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

13.2 IllegalComponentStateException

`IllegalComponentStateException` is a subclass of `IllegalStateException`; both are new to Java 1.1. This exception is used when you try to do something with a `Component` that is not yet appropriate. With the standard AWT components, this can happen only in three instances:

- If you call `setCaretPosition()` to set the cursor position of a text component before the component's peer exists.
- If you call `getLocale()` to get the locale of a component that does not have one and is not in a container that has one.
- If you call `getLocationOnScreen()` for a component that is not showing.

In these cases, the operation isn't fundamentally illegal; you are just trying to perform it before the component is ready. When you create your own components, you should consider using this exception for similar cases.

Since `IllegalComponentStateException` is a subclass of `Run-TimeException`, you do not have to enclose method calls that might throw this exception within `try/catch` blocks. However, catching this exception isn't a bad idea, since it should be fairly easy to correct the problem and retry the operation.

IllegalComponentStateException Method

Constructor

public IllegalComponentStateException () ★

The first constructor creates an `IllegalComponentStateException` instance with no detail message.

public IllegalComponentStateException (String message) ★

This constructor creates an `IllegalComponentStateException` with a detail message of `message`. This message can be retrieved using `getMessage()`, which it inherits from `Exception` (and is required by the `Throwable` interface).

IllegalComponentStateException Example

The following code throws an `IllegalComponentStateException`. The `Exception` occurs because the `TextField` peer does not exist when `setCaretPosition()` is called. `setCaretPosition()` throws an `IllegalComponentStateException`, and the next statement never executes.

```
import java.awt.TextField;
public class illegal {
    public static void main (String[] args) {
        new TextField().setCaretPosition (24);
        System.out.println ("Never gets here");
    }
}
```

PREVIOUS HOME NEXT
AWTException BOOK INDEX AWTError

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

13.3 AWTError

AWTError is a subclass of Error that is used when a serious run-time error has occurred within AWT. For example, an AWTError is thrown if the default Toolkit cannot be initialized or if you try to create a FileDialog within Netscape Navigator (since that program does not permit local file system access). When an AWTError is thrown and not caught, the virtual machine stops your program. You may throw this Error to indicate a serious run-time problem in any subclass of the AWT classes. Using AWTError is slightly preferable to creating your own Error because you don't have to provide another class file. Since it is part of Java, AWTError is guaranteed to exist on the run-time platform.

Methods are not required to declare that they throw AWTError. If you throw an error that is not caught, it will eventually propagate to the top level of the system.

AWTError Method

Constructor

public AWTError (String message)

> The sole constructor creates an AWTError with a detail message of message. This message can be retrieved using getMessage(), which it inherits from Error (and is required by the Throwable interface). If you do not want a detailed message, message may be null.

Throwing an AWTError

The code in Example 13.1 throws an AWTError if it is executed with this command:

```
java -Dawt.toolkit=foo throwme
```

The error occurs because the Java interpreter tries to use the toolkit foo, which does not exist (assuming that class `foo` does not exist in your `CLASSPATH`). Therefore, `getDefaultToolkit()` throws an `AWTError`, and the next statement never executes.

Example 13.1: The throwme class

```
import java.awt.Toolkit;
public class throwme {
    public static void main (String[] args) {
        System.out.println (Toolkit.getDefaultToolkit());
        System.out.println ("Never Gets Here");
    }
}
```

PREVIOUS | HOME | NEXT
IllegalComponentStateException | BOOK INDEX | And Then There Were Applets

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

14.2 AudioClip Interface

Once an audio file is loaded into memory with `getAudioClip()`, you use the `AudioClip` interface to work with it. Methods

Three methods define the `AudioClip` interface. The class that implements these methods depends on the run-time environment; the class is probably `sun.applet.AppletAudioClip` or `netscape.applet.AppletAudioClip`.

If you play an audio clip anywhere within your `Applet`, you should call the `AudioClip` `stop()` method within the `stop()` method of the applet. This ensures that the audio file will stop playing when the user leaves your web page. Stopping audio clips is a must if you call `loop()` to play the sound continuously; if you don't stop an audio clip, the user will have to exit the browser to get the sound to stop playing.

Applets can play audio clips simultaneously. Based upon the user's actions, you may want to play a sound file in the background continuously, while playing other files.

void play ()

> The `play()` method plays the audio clip once from the beginning.

void loop ()

> The `loop()` method plays the audio clip continuously. When it gets to the end-of-file marker, it resets itself to the beginning.

void stop ()

> The `stop()` method stops the applet from playing the audio clip.

Using an AudioClip

The applet in Example 14.2 loads three audio files in the `init()` method. The `start()` method plays Dino barking in the background as a continuous loop. Whenever the browser calls `paint()`, Fred yells "Wilma," and when you click the mouse anywhere, the call to `mouseDown()` plays Fred yelling, "Yabba-Dabba-Doo." If you try real hard, all three can play at once. Before playing any audio clip, the applet makes sure that the clip is not null--that is, that the clip loaded correctly. `stop()` stops all clips from playing; you should make sure that applets stop all audio clips before the viewer leaves the web page.

Example 14.2: AudioClip Usage

```
import java.net.*;
import java.awt.*;
import java.applet.*;
public class AudioTestExample extends Applet{
    AudioClip audio1, audio2, audio3;
    public void init () {
        audio1 = getAudioClip (getCodeBase(), "audio/flintstones.au");
        audio2 = getAudioClip (getCodeBase(), "audio/dino.au");
        audio3 = getAudioClip (getCodeBase(), "audio/wilma.au");
    }
    public boolean mouseDown (Event e, int x, int y) {
        if (audio1 != null)
            audio1.play();
        return true;
    }
    public void start () {
        if (audio2 != null)
            audio2.loop();
    }
    public void paint (Graphics g) {
        if (audio3 != null)
            audio3.play();
    }
    public void stop () {
        if (audio1 != null)
            audio1.stop();
        if (audio2 != null)
            audio2.stop();
        if (audio3 != null)
```

```
            audio3.stop();
    }
}
```

PREVIOUS HOME NEXT
What's a Java Applet? BOOK INDEX AppletContext Interface

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

14.3 AppletContext Interface

The `AppletContext` interface provides the means to control the browser environment where the applet is running. Methods

Some of these methods are so frequently used that they are also provided within the `Applet` class.

public abstract AudioClip getAudioClip (URL url)

> The `getAudioClip()` method loads the audio file located at `url`. `url` must be a complete and valid URL. Upon success, `getAudioClip()` returns an instance of a class that implements the `AudioClip` interface. You can then call methods in the `AudioClip` interface (see AudioClip Interface) to play the clip. If an error occurs during loading (e.g., because the file was not found or the URL was invalid), `getAudioClip()` returns `null`.

public abstract Image getImage (URL url)

> The `getImage()` method loads the image file located at `url`. `url` must be a complete and valid URL. The method returns a system-specific object that subclasses `Image` and returns immediately. The `Image` is not loaded until needed. A call to `prepareImage()`, `MediaTracker`, or `drawImage()` forces loading to start.

public abstract Applet getApplet (String name)

> The `getApplet()` method fetches the `Applet` from the current HTML page named `name`, which can be the applet's class name or the name provided in the `NAME` parameter of the `<APPLET>` tag. `getApplet()` returns `null` if the applet does not exist in the current context. This method allows you to call methods of other applets within the same context, loaded by the same `ClassLoader`. For example:

```
MyApplet who = (MyApplet)getAppletContext().getApplet("hey");
who.method();
```

TIP:

Netscape Navigator 3.0 restricts which applets can communicate with each other. Internet Explorer seems to have a similar restriction. For applets to communicate, they must:

- Have the same `CODEBASE`.
- Have the same or no `ARCHIVES` tag.
- Have `MAYSCRIPT` tags and appear in the same frame; alternatively, neither applet may have a `MAYSCRIPT` tag.

If these conditions are not met and you try to cast the return value of `getApplet()` or `getApplets()` to the appropriate class, either the cast will throw a `ClassCastException`; or nothing will happen, and the method will not continue beyond the point of the failure.

public abstract Enumeration getApplets ()

The `getApplets()` method gathers all the `Applets` in the current context, loaded by the same `ClassLoader`, into a collection and returns the `Enumeration`. You can then cycle through them to perform some operation collectively. For example:

```
Enumeration e = getAppletContext().getApplets();
while (e.hasMoreElements()) {
    Object o = e.nextElement();
    if (o instance of MyApplet) {
        MyApplet a = (Object)o;
        a.MyAppletMethod();
    }
}
```

TIP:

If you want communication between applets on one page, be aware that there is no guarantee which applet will start first. Communications must be synchronized by using a controlling class or continual polling.

public abstract void showDocument (URL url)

The `showDocument()` method shows `url` in the current browser window. The browser may ignore the request if it so desires.

public abstract void showDocument (URL url, String frame)

The `showDocument()` method shows `url` in a browser window specified by `frame`. Different `frame` values and the results are shown in Table 14.1. The browser may ignore the request, as *appletviewer* does.

```
try {
    URL u = new URL (getDocumentBase(), (String) file);
    getAppletContext().showDocument (u, "_blank");
} catch (Exception e) {
}
```

Table 14.1: Target Values

Target String	**Results**
`_blank`	Show `url` new browser window with no name.
`_parent`	Show `url` in the parent frame of the current window.
`_self`	Replace current `url` with `url` (i.e., display in the current window).
`_top`	Show `url` in top-most frame.
`name`	Show `url` in new browser window named `name`.

public abstract void showStatus (String message)

The `showStatus()` method displays `message` on the browser's status line, if it has one. How to display this string is up to the browser, and the browser can overwrite it whenever it wants. You should use `showStatus()` only for messages that the user can afford to miss.

PREVIOUS | HOME | NEXT
AudioClip Interface | BOOK INDEX | AppletStub Interface

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

14.4 AppletStub Interface

The `AppletStub` interface provides a way to get information from the run-time browser environment. The `Applet` class provides methods with similar names that call these methods. Methods

public abstract boolean isActive ()

> The `isActive()` method returns the current state of the applet. While an applet is initializing, it is not active, and calls to `isActive()` return `false`. The system marks the applet active just prior to calling `start()`; after this point, calls to `isActive()` return `true`.

public abstract URL getDocumentBase ()

> The `getDocumentBase()` method returns the complete URL of the HTML file that loaded the applet. This method can be used with the `getImage()` or `getAudioClip()` methods to load an image or audio file relative to the HTML file.

public abstract URL getCodeBase ()

> The `getCodeBase()` method returns the complete URL of the *.class* file that contains the applet. This method can be used with the `getImage()` method or the `getAudioClip()` method to load an image or audio file relative to the *.class* file.

public abstract String getParameter (String name)

> The `getParameter()` method allows you to get parameters from `<PARAM>` tags within the `<APPLET>` tag of the HTML file that loaded the applet. The `name` parameter of `getParameter()` must match the name string of the `<PARAM>` tag; `name` is case insensitive. The return value of `getParameter()` is the value associated with `name`; it is always a `String` regardless of the type of data in the tag. If `name` is not found within the `<PARAM>` tags

of the `<APPLET>`, `getParameter()` returns `null`.

public abstract AppletContext getAppletContext ()

The `getAppletContext()` method returns the current `AppletContext` of the applet. This is part of the stub that is set by the system when `setStub()` is called.

public abstract void appletResize (int width, int height)

The `appletResize()` method is called by the resize method of the `Applet` class. The method changes the size of the applet space to `width` x `height`. The browser must support changing the applet space; if it doesn't, the size remains unchanged.

PREVIOUS AppletContext Interface | HOME BOOK INDEX | NEXT Audio in Applications

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

14.5 Audio in Applications

The rest of this chapter describes how to use audio in your applications. Because the audio support discussed so far has been provided by the browser, applications that don't run in the context of a browser must use a different set of classes to work with audio. These classes are within the `sun.audio` package. Although the `sun.*` package hierarchy is not necessarily included by other vendors, the `sun.audio` classes discussed here are provided with Netscape Navigator 2.0/3.0 and Internet Explorer 3.0. Therefore, you can use these classes within applets, too. This section ends by developing a `SunAudioClip` class that has an interface similar to the applet's audio interface; you can use it to minimize coding differences between applets and applications.

AudioData

The `AudioData` class holds a clip of 8000 Hz µLaw audio data. This data can be used to construct an `AudioDataStream` or `ContinuousAudioDataStream`, which can then be played with the `AudioPlayer`. Constructor

public AudioData (byte buffer[])

> The `AudioData` constructor accepts a byte array `buffer` and creates an instance of `AudioData`. The buffer should contain 8000 Hz µLaw audio data.

Methods

There are no methods for `AudioData`.

AudioStream

`AudioStream` subclasses `FilterInputStream`, which extends `InputStream`. Using an `InputStream` lets you move back and forth (rewind and fast forward) within an audio file, in addition to playing the audio data from start to finish. Constructors

public AudioStream (InputStream in) throws IOException

> The `AudioStream` constructor has `InputStream in` as its parameter and can throw `IOException` on error. In the following code, we get an input stream by opening a *.au* file. Another common way to construct an `AudioStream` is to use the stream associated with a URL through the URL's `openStream()` method.

```
FileInputStream fis = new FileInputStream ("/usr/openwin/demo/sounds/1.au");
AudioStream audiostream = new AudioStream (fis);
```

or:

```
AudioStream audiostream = new AudioStream (savedUrl.openStream());
```

If you are constructing the audio data yourself, you would use a `ByteArrayInputStream`. Whatever the source of the data, the input stream should provide data in Sun's *.au* format.

Methods

public int read (byte buffer[], int offset, int length) throws IOException

The `read()` method for `AudioStream` reads an array of bytes into `buffer`. `offset` is the first element of `buffer` that is used. `length` is the maximum number of bytes to read. This method blocks until some input is available. `read()` returns the actual number of bytes read. If the end of stream is encountered and no bytes were read, `read()` returns -1. Ordinarily, you `read()` an `AudioStream` only if you want to modify the audio data in some way.

public int getLength()

The `getLength()` method returns the length of the audio data contained within the `AudioStream`, excluding any header information in the file.

public AudioData getData () throws IOException

The `getData()` method of `AudioStream` is the most important and most frequently used. It reads the data from the input stream and creates an `AudioData` instance. As the following code shows, you can create an `AudioStream` and get the `AudioData` with one statement.

```
AudioData audiodata = new AudioStream (aUrl.openStream()).getData();
```

AudioDataStream

Constructors

public AudioDataStream (AudioData data)

This constructor creates an `AudioDataStream` from an `AudioData` object `data`. The resulting `AudioDataStream` is a subclass of `ByteArrayInputStream` and can be played by the `AudioPlayer.start()` method.

Methods

There are no methods for `AudioDataStream`.

ContinuousAudioDataStream

Constructors

public ContinuousAudioDataStream (AudioData data)

> This constructor creates a continuous stream of audio from `data`. The resulting `ContinuousAudioDataStream` is a subclass of `AudioDataStream` and, therefore, of `ByteArrayInputStream`. It can be played by `AudioPlayer.start()`; whenever the player reaches the end of the continuous audio data stream, it restarts from the beginning.

Methods

public int read ()

> This `read()` method of `ContinuousAudioDataStream` overrides the `read()` method in `ByteArrayInputStream` to rewind back to the beginning of the stream when end-of-file is reached. This method is used by the system when it reads the `InputStream`; it is rarely called directly. `read()` never returns -1 since it loops back to the beginning on end-of-file.

public int read (byte buffer[], int offset, int length)

> This `read()` method of `ContinuousAudioDataStream` overrides the `read()` method in `ByteArrayInputStream` to rewind back to the beginning of the stream when end-of-file is reached. This method is used by the system when it reads the `InputStream`; it is rarely called directly. `read()` returns the actual number of bytes read. `read()` never returns -1 since it loops back to the beginning on end-of-file.

AudioStreamSequence

Constructors

public AudioStreamSequence (Enumeration e)

> The constructor for `AudioStreamSequence` accepts an `Enumeration e`(normally the elements of a `Vector` of `AudioStream`s) as its sole parameter. The constructor converts the sequence of audio streams into a single stream to be played in order. An example follows:

```
Vector v = new Vector ();
v.addElement (new AudioStream (url1.openStream ());
v.addElement (new AudioStream (url2.openStream ());
AudioStreamSequence audiostream = new AudioStreamSequence (v.elements ());
```

Methods

public int read ()

> This `read()` method of `AudioStreamSequence` overrides the `read()` method in `InputStream` to start

the next stream when end-of-file is reached. This method is used by the system when it reads the `InputStream` and is rarely called directly. If the end of all streams is encountered and no bytes were read, `read()` returns -1. Otherwise, `read()` returns the character read.

public int read (byte buffer[], int offset, int length)

This `read()` method of `AudioStreamSequence` overrides the `read()` method in `InputStream` to start the next stream when end-of-file is reached. This method is used by the system when it reads the `InputStream` and is rarely called directly. `read()` returns the actual number of bytes read. If the end of all streams is encountered and no bytes were read, `read()` returns -1.

AudioPlayer

The `AudioPlayer` class is the workhorse of the `sun.audio` package. It is used to play all the streams that were created with the other classes. There is no constructor for `AudioPlayer`; it just extends `Thread` and provides `start()` and `stop()` methods. Variable

public final static AudioPlayer player

`player` is the default audio player. This audio player is initialized automatically when the class is loaded; you do not have to initialize it (in fact, you can't because it is final) or call the constructor yourself.

Methods

public synchronized void start (InputStream in)

The `start()` method starts a thread that plays the `InputStream in`. Stream `in` continues to play until there is no more data or it is stopped. If `in` is a `ContinuousAudioDataStream`, the playing continues until `stop()` (described next) is called.

public synchronized void stop (InputStream in)

The `stop()` method stops the player from playing `InputStream in`. Nothing happens if the stream `in` is no longer playing or was never started.

SunAudioClip Class Definition

The class in Example 14.3 is all you need to play audio files in applications. It implements the `java.applet.AudioClip` interface, so the methods and functionality will be familiar. The test program in `main()` demonstrates how to use the class. Although the class itself can be used in applets, provided your users have the `sun.audio` package available, it is geared towards application users.

Example 14.3: The SunAudioClip Class

```
import java.net.URL;
import java.io.FileInputStream;
import sun.audio.*;
```

```
public class SunAudioClip implements java.applet.AudioClip {
    private AudioData audiodata;
    private AudioDataStream audiostream;
    private ContinuousAudioDataStream continuousaudiostream;
    static int length;
    public SunAudioClip (URL url) throws java.io.IOException {
        audiodata = new AudioStream (url.openStream()).getData();
        audiostream = null;
        continuousaudiostream = null;
    }
    public SunAudioClip (String filename) throws java.io.IOException {
        FileInputStream fis = new FileInputStream (filename);
        AudioStream audioStream = new AudioStream (fis);
        audiodata = audioStream.getData();
        audiostream = null;
        continuousaudiostream = null;
    }
    public void play () {
        audiostream = new AudioDataStream (audiodata);
        AudioPlayer.player.start (audiostream);
    }
    public void loop () {
        continuousaudiostream = new ContinuousAudioDataStream (audiodata);
        AudioPlayer.player.start (continuousaudiostream);
    }
    public void stop () {
        if (audiostream != null)
            AudioPlayer.player.stop (audiostream);
        if (continuousaudiostream != null)
            AudioPlayer.player.stop (continuousaudiostream);
    }
    public static void main (String args[]) throws Exception {
        URL url1 = new URL ("http://localhost:8080/audio/1.au");
        URL url2 = new URL ("http://localhost:8080/audio/2.au");
        SunAudioClip sac1 = new SunAudioClip (url1);
        SunAudioClip sac2 = new SunAudioClip (url2);
        SunAudioClip sac3 = new SunAudioClip ("1.au");
        sac1.play ();
        sac2.loop ();
        sac3.play ();
        try {// Delay for loop
            Thread.sleep (2000);
        } catch (InterruptedException ie) {}
        sac2.stop();
    }
}
```

16.3 ClipboardOwner Interface

Classes that need to place objects on a clipboard must implement the `ClipboardOwner` interface. An object becomes the clipboard owner by placing something on a `Clipboard` and remains owner as long as that object stays on the clipboard; it loses ownership when someone else writes to the clipboard. The `ClipboardOwner` interface provides a way to receive notification when you lose ownership--that is, when the object you placed on the clipboard is replaced by something else. Methods

public abstract void lostOwnership(Clipboard clipboard, Transferable contents) ★

> The `lostOwnership()` method tells the owner of `contents` that it is no longer on the given `clipboard`. It is usually implemented as an empty stub but is available for situations in which you have to know.

16.4 Clipboard

The `Clipboard` class is a repository for a `Transferable` object and can be used for cut, copy, and paste operations. You can work with a private clipboard by creating your own instance of `Clipboard`, or you can work with the system clipboard by asking the `Toolkit` for it:

```
Toolkit.getDefaultToolkit().getSystemClipboard()
```

When working with the system clipboard, native applications have access to information created within Java programs and vice versa. Access to the system clipboard is controlled by the `SecurityManager` and is restricted within applets.

Clipboard Methods

Variables

protected ClipboardOwner owner ★

> The `owner` instance variable represents the current owner of `contents`. When something new is placed on the clipboard, the previous owner is notified by a call to the `lostOwnership()` method. The owner usually ignores this notification. However, the clipboard's `contents` are passed back to `owner` in case some special processing or comparison needs to be done.

protected Transferable contents ★

> The `contents` instance variable is the object currently on the clipboard; it was placed on the clipboard by `owner`. To retrieve the current contents, use the `getContents()` method.

Constructors

public Clipboard(String name) ★

The constructor for `Clipboard` allows you to create a private clipboard named `name`. This clipboard is not accessible outside of your program and has no security constraints placed upon it.

Miscellaneous methods

public String getName() ★

The `getName()` method fetches the clipboard's name. For private clipboards, this is the name given in the constructor. The name of the system clipboard is "System".

public synchronized Transferable getContents(Object requester) ★

The `getContents()` method allows you to retrieve the current contents of the clipboard. This is the method you would call when the user selects Paste from a menu.

Once you have the `Transferable` data, you try to get the data in whatever flavor you want by calling the `Transferable.getTransferData()` method, possibly after calling `Transferable.isDataFlavorSupported()`. The `requester` represents the object that is requesting the clipboard's contents; it is usually just `this`, since the current object is making the request.

public synchronized void setContents(Transferable contents, ClipboardOwner owner) ★

The `setContents()` method changes the contents of the clipboard to `contents` and changes the clipboard's owner to `owner`. You would call this method when the user selects Cut or Copy from a menu. The `owner` parameter represents the object that owns `contents`. This object must implement the `ClipboardOwner` interface; it will be notified by a call to `lostOwnership()` when something else is placed on the clipboard.

16.5 StringSelection

`StringSelection` is a convenience class that can be used for copy and paste operations on Unicode text strings (`String`). It implements both the `ClipboardOwner` and `Transferable` interfaces, so it can be used both as the contents of the clipboard and as its owner. For example, if `s` is a `StringSelection`, you can call `Clipboard.setContents(s,s)`. `StringSelection` supports both `stringFlavor` and `plainTextFlavor` and doesn't do anything when it loses clipboard ownership.

StringSelection Methods

Constructors

public StringSelection(String data) ★

> The constructor creates an instance of `StringSelection` containing `data`. You can use this object to place the data on a clipboard.

Miscellaneous methods

public DataFlavor[] getTransferDataFlavors() ★

> The `getTransferDataFlavors()` method returns a two-element `DataFlavor` array consisting of `DataFlavor.stringFlavor` and `DataFlavor.plainTextFlavor`. This means that you can paste a `StringSelection` as either a Java `String` or as plain text (i.e., the MIME type `plain/text`).

public boolean isDataFlavorSupported(DataFlavor flavor) ★

> The `isDataFlavorSupported()` method is returns `true` if `flavor` is either

`DataFlavor.stringFlavor` or `DataFlavor.plainTextFlavor`; it returns `false` for any other flavor.

public Object getTransferData(DataFlavor flavor) throws UnsupportedFlavorException, IOException ★

The `getTransferData()` method returns an object from which you can get the data on the clipboard; the object's type is determined by the `flavor` parameter. This method returns a `String` containing the data on the clipboard if `flavor` is `DataFlavor.stringFlavor`; it returns a `StringBufferInputStream` from which you can read the data on the clipboard if you ask for `DataFlavor.plainTextFlavor`. Otherwise, `getTransferData()` throws an `UnsupportedFlavorException`.

public void lostOwnership(Clipboard clipboard, Transferable contents) ★

The `lostOwnership()` method of `StringSelection` is an empty stub; it does nothing when you lose ownership. If you want to know when you've lost ownership of string data placed on the clipboard, write a subclass of `StringSelection` and override this method.

PREVIOUS Clipboard | HOME BOOK INDEX | NEXT UnsupportedFlavorException

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

16.7 Reading and Writing the Clipboard

Now that you know about the different `java.awt.datatransfer` classes required to use the clipboard, let's put them all together in an example. Example 16.1 creates a `TextField` for input (copying), a read-only `TextArea` for output (pasting), and a couple of buttons to control its operation. Figure 16.1 shows the program's user interface. When the user clicks on the Copy button or presses Return in the `TextField`, the text in the `TextField` is copied to the `Clipboard`. When the user clicks on the Paste button, the contents of the clipboard are drawn in the `TextArea`. Since the clipboard is not private, you can copy or paste from anywhere on your desktop, not just this program.

Example 16.1: Using the System Clipboard

```
// Java 1.1 only
import java.io.*;
import java.awt.*;
import java.awt.datatransfer.*;
public class ClipMe extends Frame {
    TextField tf;
    TextArea ta;
    Button copy, paste;
    Clipboard clipboard = null;
    ClipMe() {
        super ("Clipping Example");
        add (tf = new TextField("Welcome"), "North");
        add (ta = new TextArea(), "Center");
        ta.setEditable(false);
        Panel p = new Panel();
        p.add (copy = new Button ("Copy"));
        p.add (paste = new Button ("Paste"));
        add (p, "South");
        setSize (250, 250);
    }
    public static void main (String args[]) {
        new ClipMe().show();
    }
    public boolean handleEvent (Event e) {
        if (e.id == Event.WINDOW_DESTROY) {
            System.exit(0);
            return true;  // never gets here
        }
        return super.handleEvent (e);
    }
    public boolean action (Event e, Object o) {
        if (clipboard == null)
            clipboard = getToolkit().getSystemClipboard();
        if ((e.target == tf) || (e.target == copy)) {
            StringSelection data;
            data = new StringSelection (tf.getText());
            clipboard.setContents (data, data);
        } else if (e.target == paste) {
            Transferable clipData = clipboard.getContents(this);
            String s;
            try {
                s = (String)(clipData.getTransferData(
                       DataFlavor.stringFlavor));
            } catch (Exception ee) {
                s = ee.toString();
            }
            ta.setText(s);
        }
        return true;
    }
}
```

Figure 16.1: Using the system clipboard

[Graphic: Figure 16-1]

We won't say anything about how the display is set up; that should be familiar. All the interesting stuff happens in the `action` method, which is called in response to a button click. We check which button the user clicked; if the user clicked the Copy button, we read the text field `tf` and use it to create a new `StringSelection` named `data`. If the user clicked the Paste button, we retrieve the data from the clipboard by calling `getContents()`. This gives us an object about which (strictly speaking) we know nothing, except that it implements `Transferable`. In this case, we're pretty sure that we're getting text from the clipboard, so we call `getTransferData()` and ask for the data in the `stringFlavor` form. We catch the exception that might occur if we're wrong about the data flavor. This program has no way of placing anything but text on the clipboard, but there's no guarantee that the user didn't cut some other kind of object from a native application.

Once we have our `String`, we call the `setText()` method of the `TextArea` to tell it about the new string, and we are finished.

PREVIOUS | HOME | NEXT
UnsupportedFlavorException | BOOK INDEX | Printing

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

18.2 Package diagrams

The following figures provide a visual representation of the relationships between the classes in the AWT packages.

`java.awt`, as the mother of all AWT packages, is better represented by two diagrams, one for the graphics classes and one for the component and layout classes.

Figure 18.1: Component and Layout classes of the java.awt package.

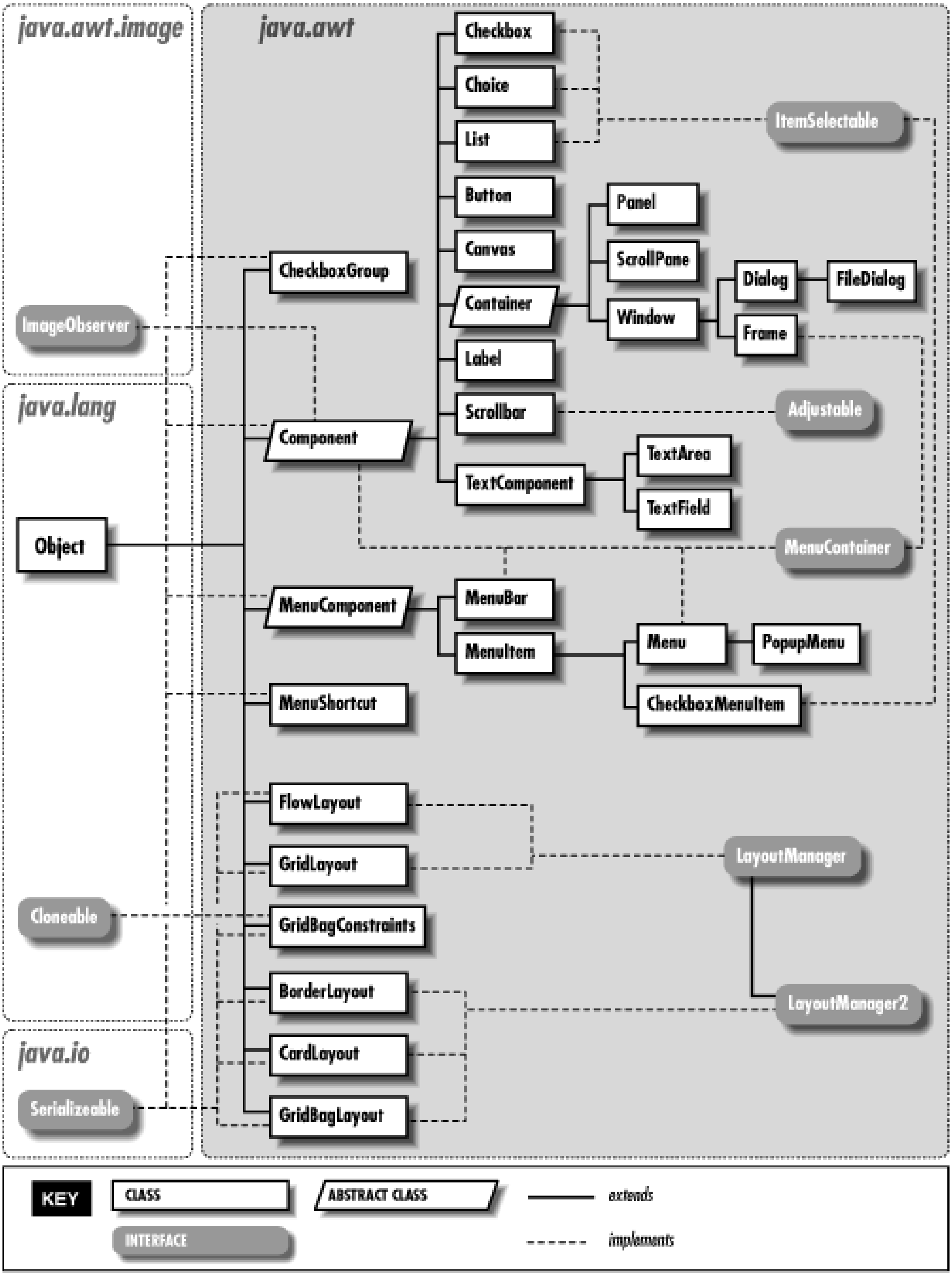


Figure 18.2: Graphics classes of java.awt package

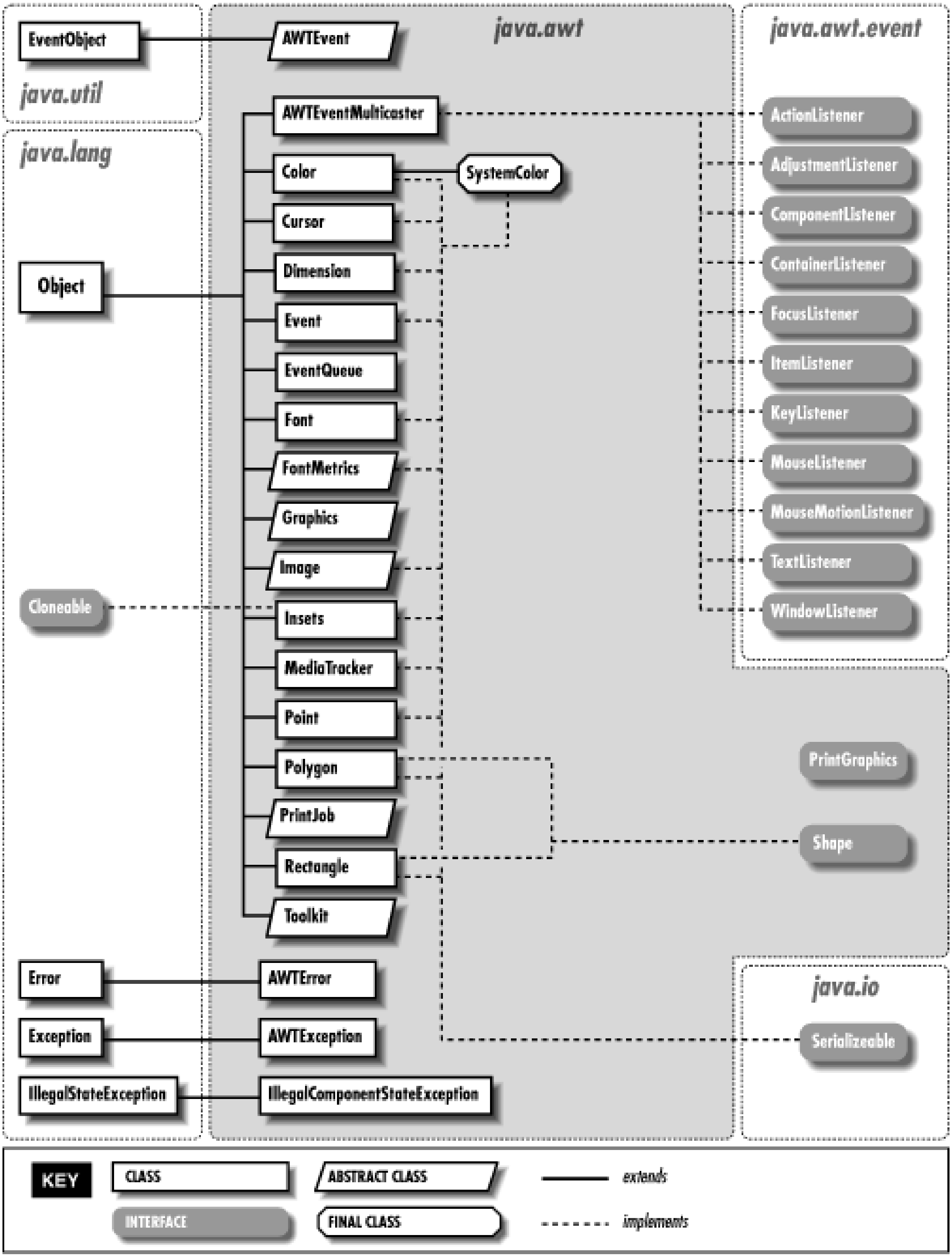


Figure 18.3: The java.awt.image package

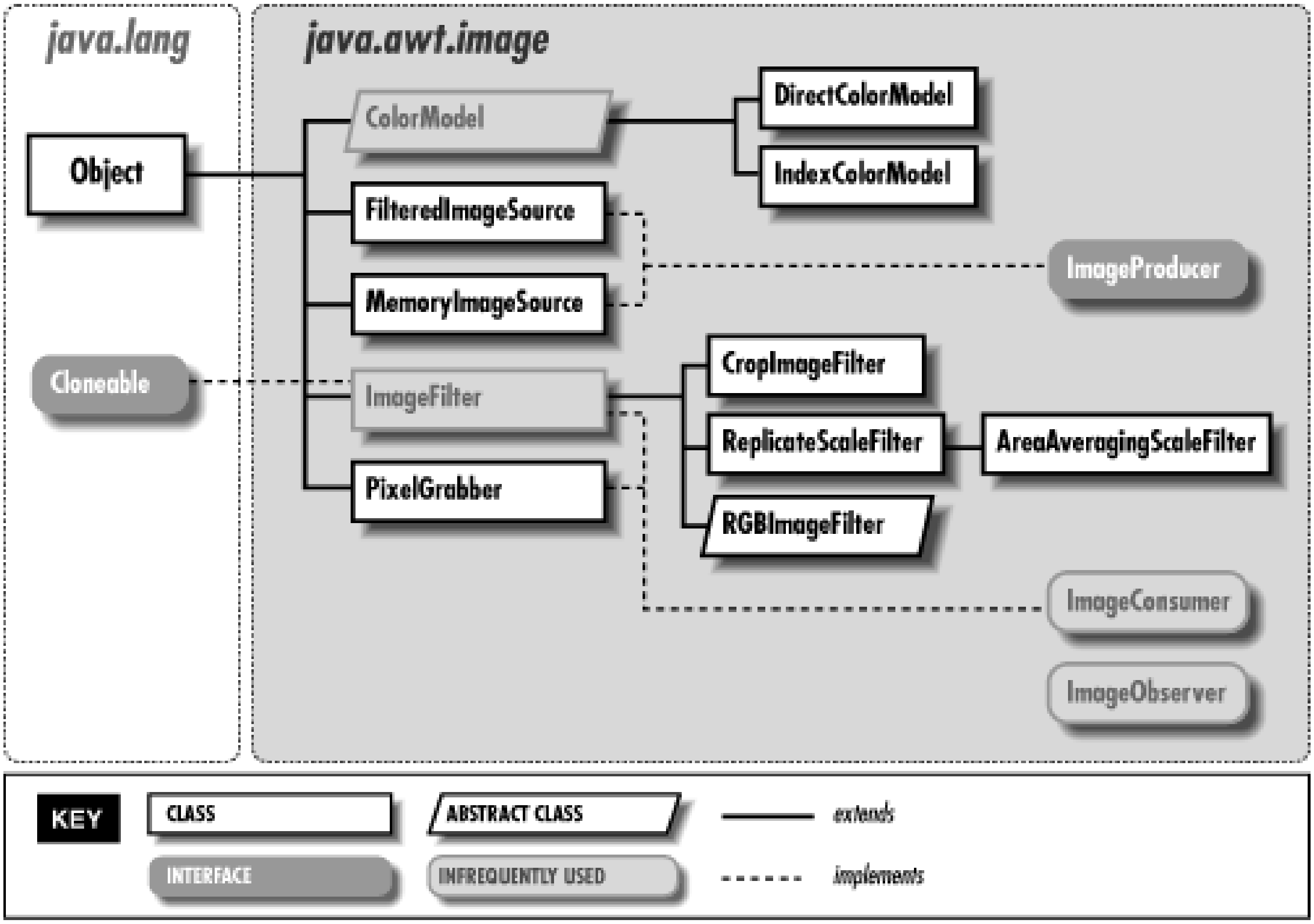


Figure 18.4: The java.awt.datatransfer package

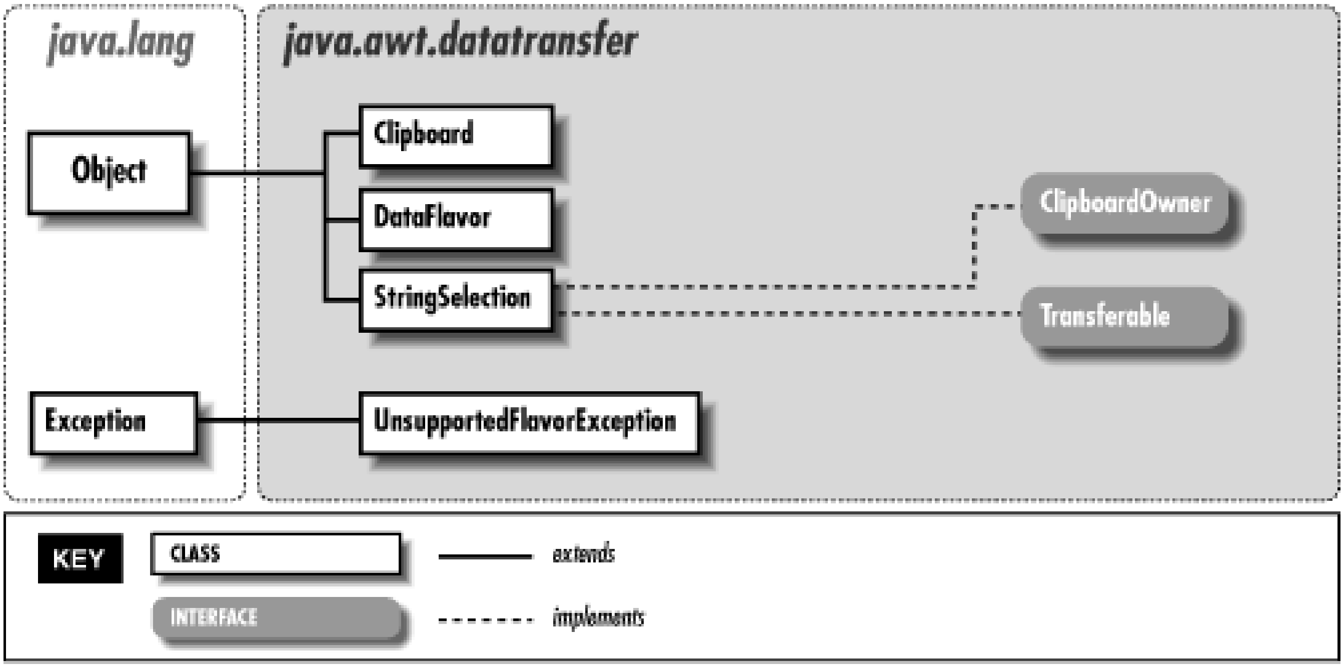


Figure 18.5: The java.awt.event package

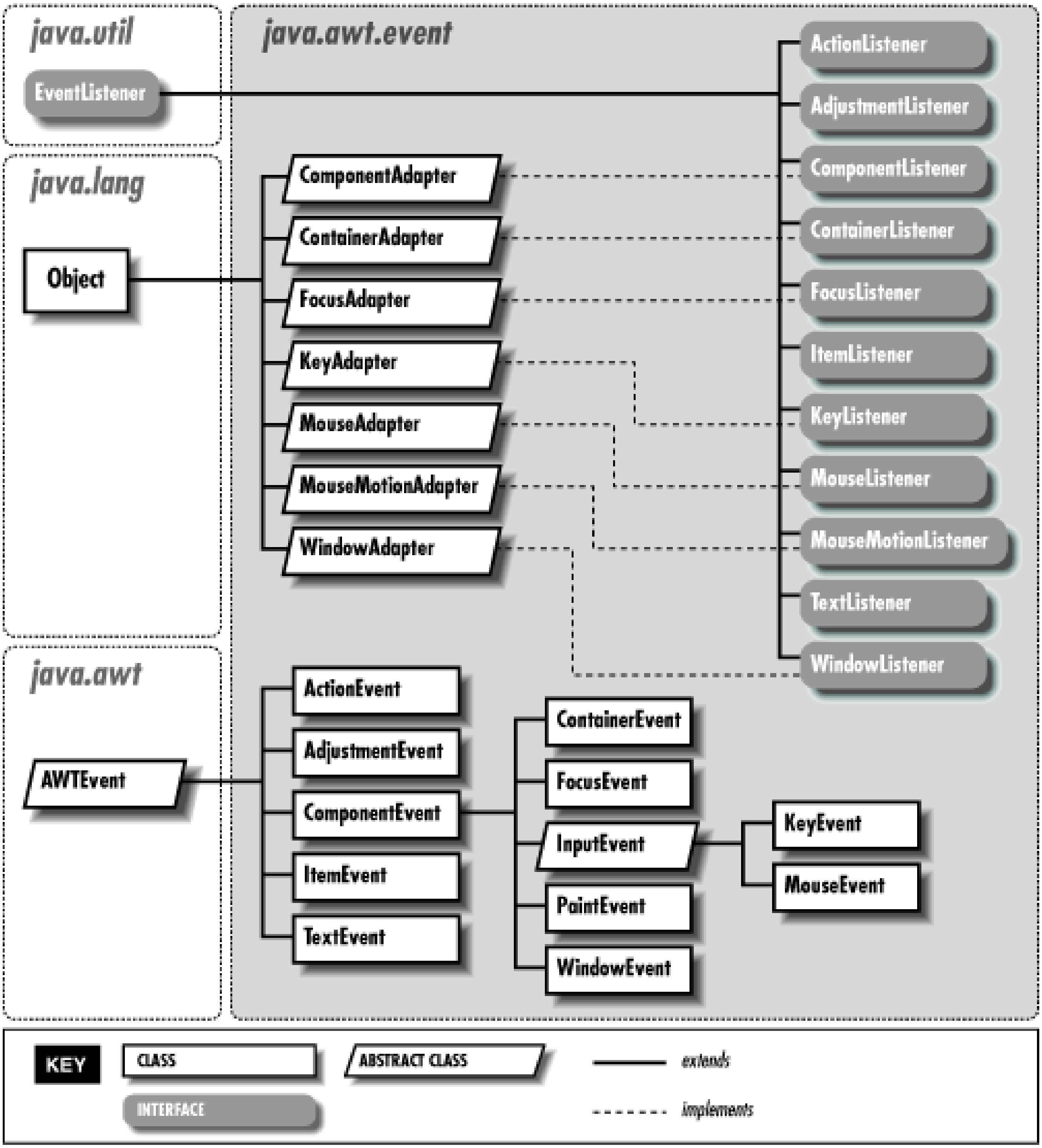


Figure 18.6: The java.awt.peer package

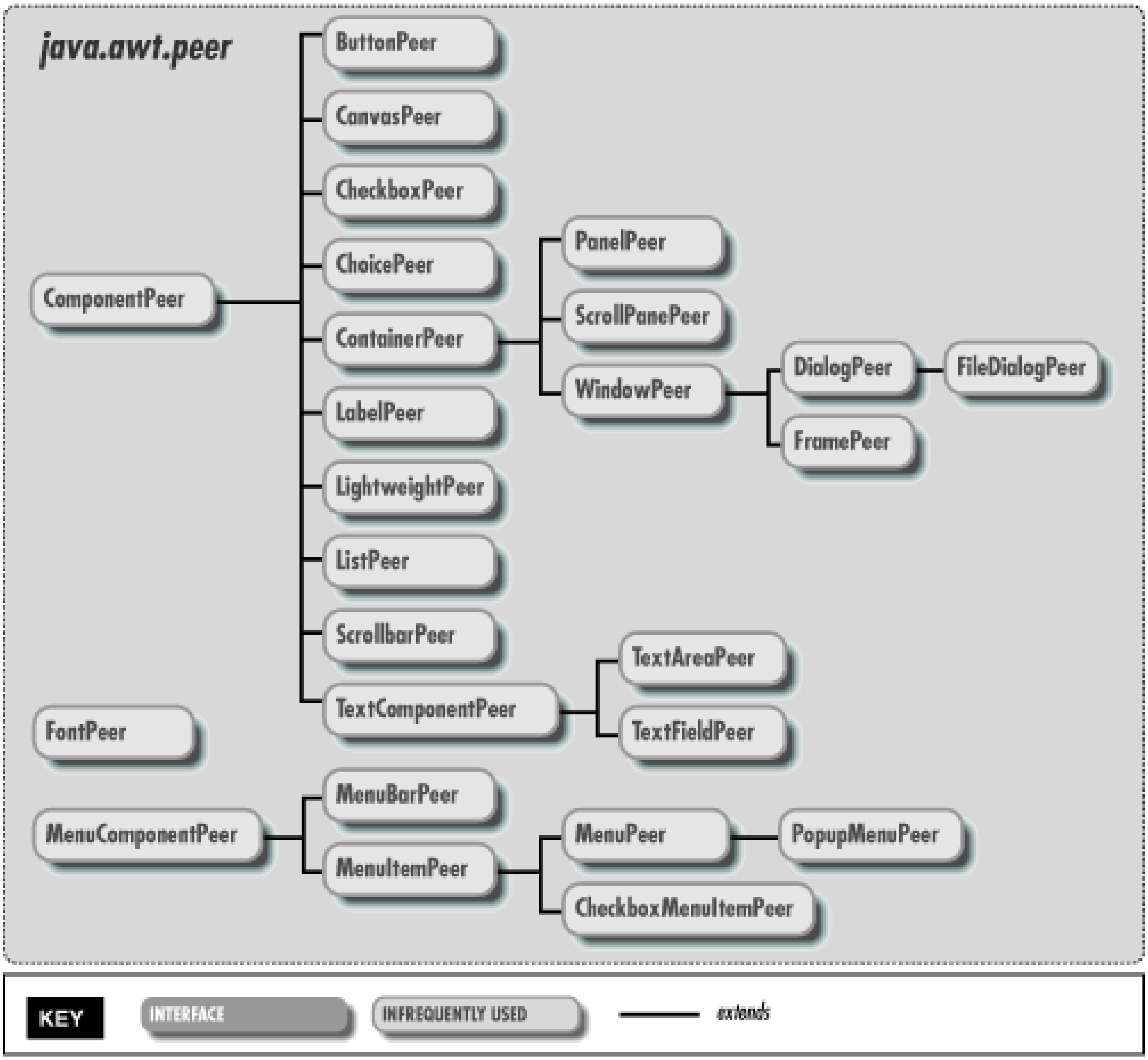
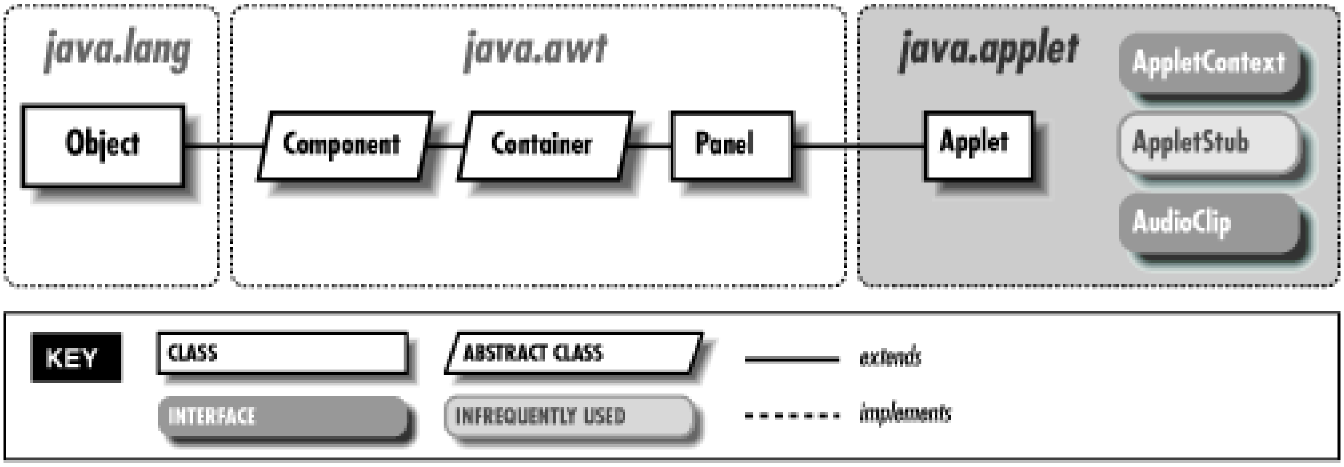


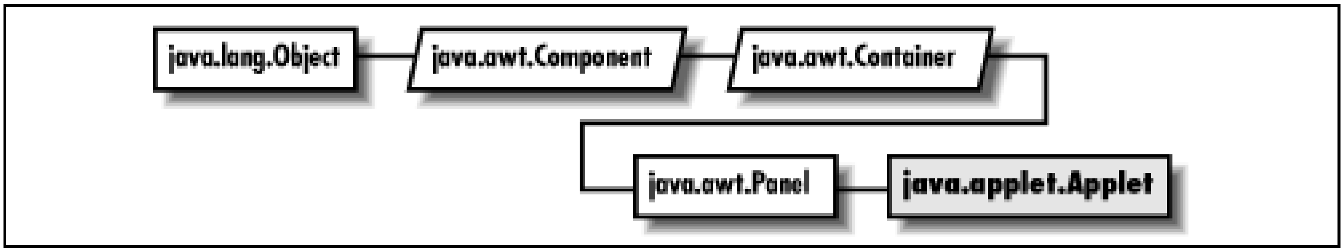
Figure 18.7: The java.applet package



Applet

Name

Applet



Description

The `Applet` class provides the framework for delivering Java programs within web pages.

Class Definition

```
public class java.applet.Applet
    extends java.awt.Panel {
  // Constructors
  public Applet();
  // Instance Methods
  public void destroy();
  public AppletContext getAppletContext();
  public String getAppletInfo();
  public AudioClip getAudioClip (URL url);
  public AudioClip getAudioClip (URL url, String filename);
  public URL getCodeBase();
  public URL getDocumentBase();
  public Image getImage (URL url);
  public Image getImage (URL url, String filename);
  public Locale getLocale(); ★
  public String getParameter (String name);
  public String[][] getParameterInfo();
  public void init();
  public boolean isActive();
  public void play (URL url);
  public void play (URL url, String filename);
  public void resize (int width, int height);
  public void resize (Dimension dim);
  public final void setStub (AppletStub stub);
  public void showStatus (String message);
  public void start();
  public void stop();
}
```

Constructors

Applet

public Applet()

Description

> Constructs an `Applet` object.

Instance Methods

destroy

public void destroy()

Description

> Called when the browser determines that it doesn't need to keep the applet around anymore.

getAppletContext

public AppletContext getAppletContext()

Returns

The current `AppletContext` of the applet.

getAppletInfo

public String getAppletInfo()

Returns

A short information string about the applet to be shown to the user.

getAudioClip

public AudioClip getAudioClip (URL url)

Parameters

url

`URL` of an audio file.

Returns

Object that implements the `AudioClip` interface for playing audio files.

Description

Fetches an audio file to play with the `AudioClip` interface.

public AudioClip getAudioClip (URL url , String filename)

Parameters

url

Base `URL` of an audio file.

filename

Specific file, relative to `url`, that contains an audio file.

Returns

Object that implements `AudioClip` interface for playing audio file.

Description

Fetches an audio file to play with the `AudioClip` interface.

getCodeBase

public URL getCodeBase()

Returns

The complete `URL` of the *.class* file that contains the applet.

getDocumentBase

public URL getDocumentBase()

Returns

The complete `URL` of the *.html* file that loaded the applet.

getImage

public Image getImage (URL url)

Parameters

url

`URL` of an image file.

Returns

`Image` to be displayed.

Description

Initiates the image loading process for the file located at the specified location.

public Image getImage (URL url, String filename)

Parameters

url

Base `URL` of an image file.

filename

Specific file, relative to `url`, that contains an image file.

Returns

Image to be displayed.

Description

Initiates the image loading process for the file located at the specified location.

getLocale

public Locale getLocale() ★

Returns

`Applet`'s locale.

Overrides

`Component.getLocale()`

Description

Used for internationalization support.

getParameter

public String getParameter (String name)

Parameters

name

Name of parameter to get.

Returns

The value associated with the given parameter in the HTML file, or `null`.

Description

Allows you to get parameters from within the `<APPLET>` tag of the *.html* file that loaded the applet.

getParameterInfo

public String[][] getParameterInfo()

Returns

Overridden to provide a series of three-string arrays that describes the parameters this applet reads.

init

public void init()

Description

Called by the system when the applet is first loaded.

isActive

public boolean isActive()

Returns

> `true` if the applet is active, `false` otherwise.

play

public void play (URL url)

Parameters

> *url*
>
> > `URL` of an audio file .

Description

> Plays an audio file once.

public void play (URL url, String filename)

Parameters

> *url*
>
> > Base `URL` of an audio file .
>
> *filename*
>
> > Specific file, relative to `url`, that contains an audio file.

Description

> Plays an audio file once.

resize

public void resize(int width, int height)

Parameters

width

New width for the `Applet`.

height

New height for the `Applet`.

Description

Changes the size of the applet.

public void resize (Dimension dim)

Parameters

dim

New dimensions for the applet.

Description

Changes the size of the applet.

setStub

public final void setStub (AppletStub stub)

Parameters

stub

Platform specific stubfor environment.

Description

Called by the system to setup `AppletStub`.

showStatus

public void showStatus (String message)

Parameters

message

Message to display to user.

Description

Displays a message on the status line of the browser.

start

public void start()

Description

Called by the system every time the applet is displayed.

stop

public void stop()

Description

Called by the system when it wants the applet to stop execution; typically, every time the user leaves the page that includes the applet.

See Also

`AppletContext`, `AppletStub`, `AudioClip`, `Container`, `Dimension`, `Image`, `Locale`, `Panel`, `String`, `URL`

AppletContext

Name

AppletContext

Description

`AppletContext` is an interface that provides the means to control the browser environment in which the applet is running.

Interface Definition

```
public abstract interface java.applet.AppletContext {
  // Interface Methods
  public abstract Applet getApplet (String name);
  public abstract Enumeration getApplets();
  public abstract AudioClip getAudioClip (URL url);
  public abstract Image getImage (URL url);
  public abstract void showDocument (URL url);
  public abstract void showDocument (URL url, String frame);
  public abstract void showStatus (String message);
}
```

Interface Methods

getApplet

public abstract Applet getApplet (String name)

Parameters

name

Name of applet to locate.

Returns

`Applet` fetched.

Description

Gets a reference to another executing applet.

getApplets

public abstract Enumeration getApplets()

Returns

List of applets executing.

Description

Gets references to all executing applets.

getAudioClip

public abstract AudioClip getAudioClip (URL url)

Parameters

url

Location of an audio file.

Returns

`AudioClip` fetched.

Description

Loads an audio file.

getImage

public abstract Image getImage (URL url)

Parameters

url

Location of an image file.

Returns

Image fetched.

Description

Loads an image file.

showDocument

public abstract void showDocument (URL url)

Parameters

url

New web page to display.

Description

Changes the displayed web page.

public abstract void showDocument (URL url, String frame)

Parameters

url

New web page to display.

frame

Name of the frame in which to display the new page.

Description

Displays a web page in another frame.

showStatus

public abstract void showStatus (String message)

Parameters

message

Message to display.

Description

Displays a message on the status line of the browser.

See Also

`Applet`, `AudioClip`, `Enumeration`, `Image`, `Object`, `String`, `URL`

AppletStub

Name

AppletStub

[Graphic: Figure from the text]

Description

`AppletStub` is an interface that provides the means to get information from the run-time browser environment.

Interface Definition

```
public abstract interface java.applet.AppletStub {
  // Interface Methods
  public abstract void appletResize (int width, int height);
  public abstract AppletContext getAppletContext();
  public abstract URL getCodeBase();
  public abstract URL getDocumentBase();
  public abstract String getParameter (String name);
  public abstract boolean isActive();
}
```

Interface Methods

appletResize

public abstract void appletResize (int width, int height)

Parameters

width

Requested new width for applet.

height

Requested new height for applet.

Description

Changes the size of the applet.

getAppletContext

public abstract AppletContext getAppletContext()

Returns

Current `AppletContext` of the applet.

getCodeBase

public abstract URL getCodeBase()

Returns

Complete `URL` for the applet's .*class* file.

getDocumentBase

public abstract URL getDocumentBase()

Returns

Complete `URL` for the applet's *.html* file.

getParameter

public abstract String getParameter (String name)

Parameters

name

Name of a `<PARAM>` tag.

Returns

Value associated with the parameter.

Description

Gets a parameter value from the `<PARAM>` tag(s) of the applet.

isActive

public abstract boolean isActive()

Returns

`true` if the applet is active, `false` otherwise

Description

Returns current state of the applet.

See Also

`AppletContext`, `Object`, `String`, `URL`

AudioClip

Name

AudioClip

Description

`AudioClip` is an interface for playing audio files.

Interface Definition

```
public abstract interface java.applet.AudioClip {
  // Interface Methods
  public abstract void loop();
  public abstract void play();
  public abstract void stop();
}
```

Interface Methods

loop

public abstract void loop()

Description

Plays an audio clip continuously.

play

public abstract void play()

Description

Plays an audio clip once from the beginning.

stop

public abstract void stop()

Description

Stops playing an audio clip.

See Also

`Object`

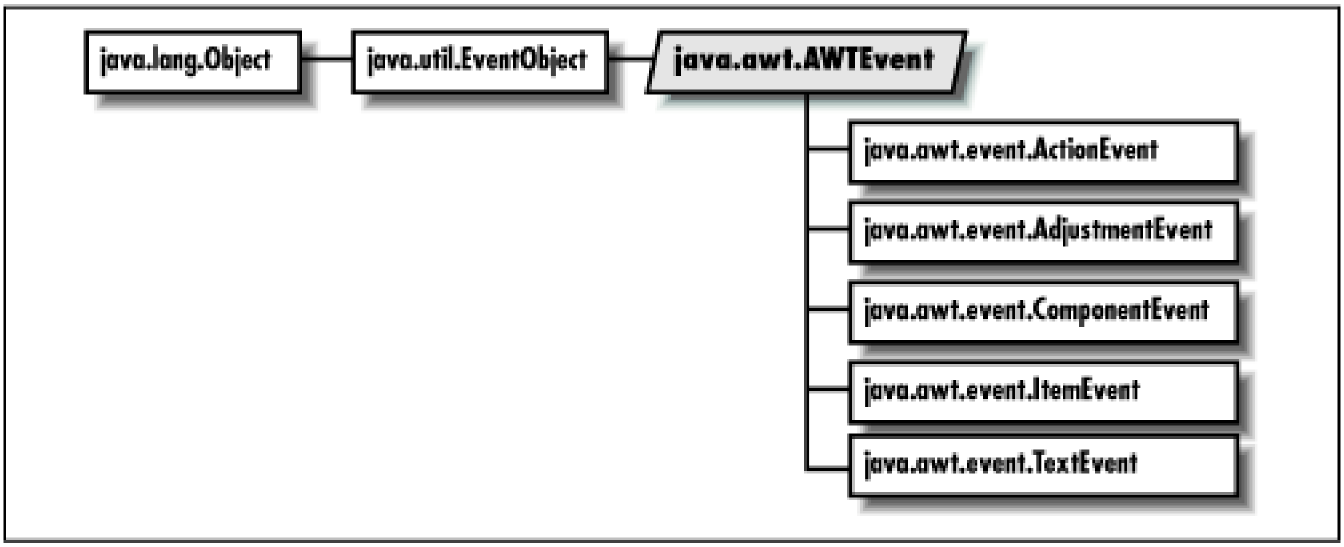
PREVIOUS | HOME | NEXT
AppletStub | BOOK INDEX | AWTError

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

AWTEvent ★

Name

AWTEvent ★



Description

The root class of all AWT events. Subclasses of this class are the replacement for `java.awt.Event`, which is only used for the Java 1.0.2 event model. In Java 1.1, event objects are passed from event source components to objects implementing a corresponding listener interface. Some event sources have a corresponding interface, too. For example, `AdjustmentEvent`s are passed from `Adjustable` objects to `AdjustmentListener`s. Some event types do not have corresponding interfaces; for example, `ActionEvent`s are passed from `Button`s to `ActionListener`s, but there is no "Actionable" interface that `Button` implements.

Class Definition

```
public abstract class java.awt.AWTEvent
    extends java.util.EventObject {
  // Constants
  public final static long ACTION_EVENT_MASK;
  public final static long ADJUSTMENT_EVENT_MASK;
  public final static long COMPONENT_EVENT_MASK;
  public final static long CONTAINER_EVENT_MASK;
  public final static long FOCUS_EVENT_MASK;
  public final static long ITEM_EVENT_MASK;
  public final static long KEY_EVENT_MASK;
  public final static long MOUSE_EVENT_MASK;
  public final static long MOUSE_MOTION_EVENT_MASK;
  public final static long RESERVED_ID_MAX;
  public final static long TEXT_EVENT_MASK;
  public final static long WINDOW_EVENT_MASK;
  // Variables
  protected boolean consumed;
  protected int id;
  // Constructors
  public AWTEvent (Event event);
  public AWTEvent (Object source, int id);
  // Instance Methods
  public int getID();
  public String paramString();
  public String toString();
  // Protected Instance Methods
  protected void consume();
  protected boolean isConsumed();
}
```

Constants

ACTION_EVENT_MASK

public static final long ACTION_EVENT_MASK

The mask for action events.

ADJUSTMENT_EVENT_MASK

public static final long ADJUSTMENT_EVENT_MASK

The mask for adjustment events.

COMPONENT_EVENT_MASK

public static final long COMPONENT_EVENT_MASK

The mask for component events.

CONTAINER_EVENT_MASK

public static final long CONTAINER_EVENT_MASK

The mask for container events.

FOCUS_EVENT_MASK

public static final long FOCUS_EVENT_MASK

The mask for focus events.

ITEM_EVENT_MASK

public static final long ITEM_EVENT_MASK

The mask for item events.

KEY_EVENT_MASK

public static final long KEY_EVENT_MASK

The mask for key events.

MOUSE_EVENT_MASK

public static final long MOUSE_EVENT_MASK

The mask for mouse events.

MOUSE_MOTION_EVENT_MASK

public static final long MOUSE_MOTION_EVENT_MASK

The mask for mouse motion events.

RESERVED_ID_MAX

public static final int

The maximum reserved event id.

TEXT_EVENT_MASK

public static final long TEXT_EVENT_MASK

The mask for text events.

WINDOW_EVENT_MASK

public static final long WINDOW_EVENT_MASK

The mask for window events.

Variables

consumed

protected boolean consumed

If `consumed` is `true`, the event will not be sent back to the peer. Semantic events will never be sent back to a peer; thus `consumed` is always `true` for semantic events.

id

protected int id

The type ID of this event.

Constructors

AWTEvent

public AWTEvent (Event event)

Parameters

> *event*
>
> > A version 1.0.2 `java.awt.Event` object.

Description

> Constructs a 1.1 `java.awt.AWTEvent` derived from a 1.0.2 `java.awt.Event` object.

public AWTEvent (Object source, int id)

Parameters

> *source*
>
> > The object that the event originated from.
>
> *id*
>
> > An event type ID.

Description

> Constructs an `AWTEvent` object.

Instance Methods

getID

public int getID()

Returns

> The type ID of the event.

paramString

public String paramString()

Returns

> A string with the current settings of `AWTEvent`.

Description

> Helper method for `toString()` that generates a string of current settings.

toString

public String toString()

Returns

> A string representation of the `AWTEvent` object.

Overrides

> `Object.toString()`

Protected Instance Methods

consume

protected void consume()

Description

Consumes the event so it is not sent back to its source.

isConsumed

public boolean isConsumed()

Returns

A flag indicating whether this event has been consumed.

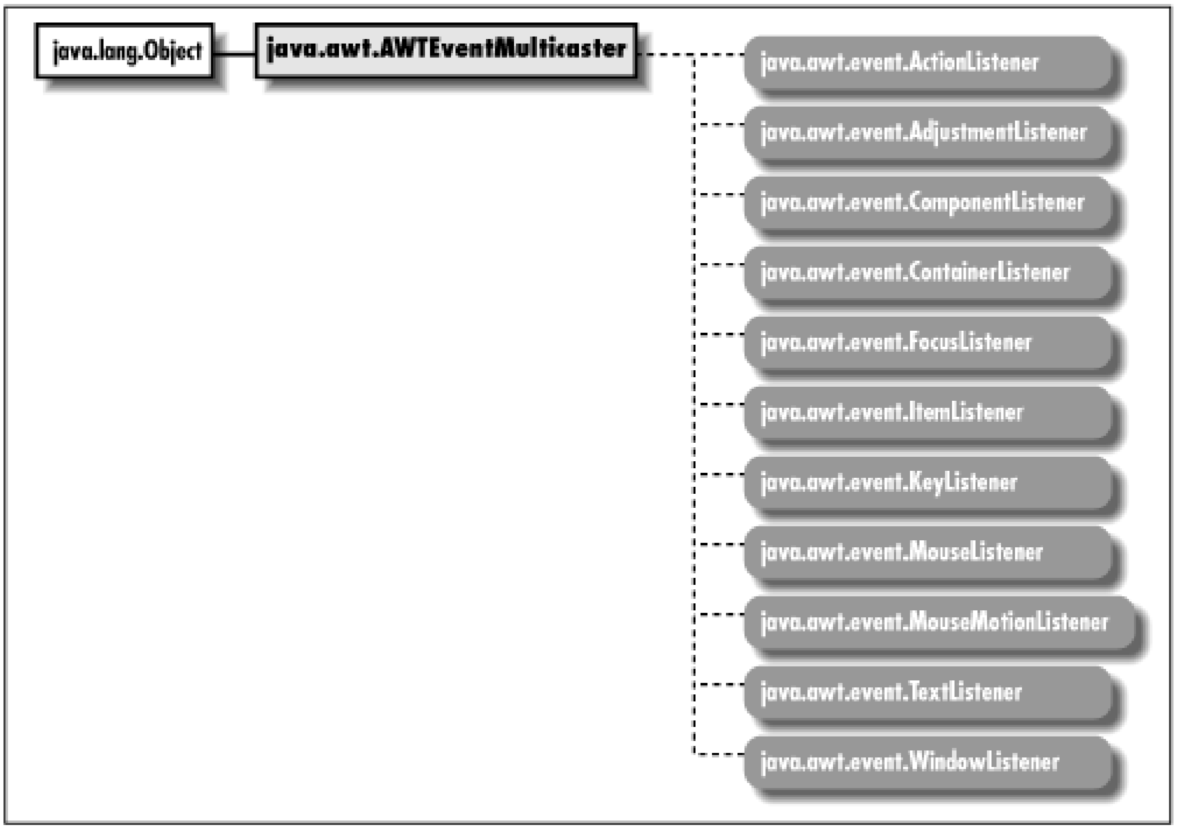
See Also

`ActionEvent`, `AdjustmentEvent`, `ComponentEvent`, `Event`, `EventObject`, `FocusEvent`, `ItemEvent`, `KeyEvent`, `MouseEvent`, `WindowEvent`

AWTEventMulticaster ★

Name

AWTEventMulticaster ★


java.lang.Object
java.awt.AWTEventMulticaster
java.awt.event.ActionListener
java.awt.event.AdjustmentListener
java.awt.event.ComponentListener
java.awt.event.ContainerListener
java.awt.event.FocusListener
java.awt.event.ItemListener
java.awt.event.KeyListener
java.awt.event.MouseListener
java.awt.event.MouseMotionListener
java.awt.event.TextListener
java.awt.event.WindowListener

Description

This class multicasts events to event listeners. Each multicaster has two listeners, cunningly named `a` and `b`. When an event source calls one of the listener methods of the multicaster, the multicaster calls the same listener method on both `a` and `b`. Multicasters are built into trees using the static `add()` and `remove()` methods. In this way a single event can be sent to many listeners.

Static methods make it easy to implement event multicasting in component subclasses. Each time an `add<type>Listener()` function is called in the component subclass, call the corresponding `AWTEventMulticaster.add()` method to chain together (or "tree up") listeners. Similarly, when a `remove<type>Listener()` function is called, `AWTEventMulticaster.remove()` can be called to remove a chained listener.

Class Definition

```
public class java.awt.AWTEventMulticaster
  extends java.lang.Object
  implements java.awt.event.ActionListener, java.awt.event.AdjustmentListener,
             java.awt.event.ComponentListener, java.awt.event.ContainerListener,
             java.awt.event.FocusListener, java.awt.event.ItemListener,
             java.awt.event.KeyListener, java.awt.event.MouseListener,
             java.awt.event.MouseMotionListener, java.awt.event.TextListener,
             java.awt.event.WindowListener {
  // Variables
  protected EventListener a;
  protected EventListener b;
  // Constructors
  protected AWTEventMulticaster(EventListener a, EventListener b);
  // Class Methods
  public static ActionListener add(ActionListener a, ActionListener b);
  public static AdjustmentListener add(AdjustmentListener a,
      AdjustmentListener b);
  public static ComponentListener add(ComponentListener a,
      ComponentListener b);
  public static ContainerListener add(ContainerListener a,
      ContainerListener b);
  public static FocusListener add(FocusListener a, FocusListener b);
  public static ItemListener add(ItemListener a, ItemListener b);
  public static KeyListener add(KeyListener a, KeyListener b);
  public static MouseListener add(MouseListener a, MouseListener b);
  public static MouseMotionListener add(MouseMotionListener a,
      MouseMotionListener b);
  public static TextListener add(TextListener a, TextListener b);
  public static WindowListener add(WindowListener a, WindowListener b);
  protected static EventListener addInternal(EventListener a, EventListener b);
  public static ActionListener remove(ActionListener l, ActionListener oldl);
  public static AdjustmentListener remove(AdjustmentListener l,
      AdjustmentListener oldl);
  public static ComponentListener remove(ComponentListener l,
      ComponentListener oldl);
  public static ContainerListener remove(ContainerListener l,
      ContainerListener oldl);
  public static FocusListener remove(FocusListener l, FocusListener oldl);
  public static ItemListener remove(ItemListener l, ItemListener oldl);
  public static KeyListener remove(KeyListener l, KeyListener oldl);
  public static MouseListener remove(MouseListener l, MouseListener oldl);
  public static MouseMotionListener remove(MouseMotionListener l,
      MouseMotionListener oldl);
  public static TextListener remove(TextListener l, TextListener oldl);
  public static WindowListener remove(WindowListener l, WindowListener;
  protected static EventListener removeInternal(EventListener l,
      EventListener oldl);
  // Instance Methods
  public void actionPerformed(ActionEvent e);
  public void adjustmentValueChanged(AdjustmentEvent e);
  public void componentAdded(ContainerEvent e);
  public void componentHidden(ComponentEvent e);
  public void componentMoved(ComponentEvent e);
  public void componentRemoved(ContainerEvent e);
  public void componentResized(ComponentEvent e);
  public void componentShown(ComponentEvent e);
  public void focusGained(FocusEvent e);
  public void focusLost(FocusEvent e);
  public void itemStateChanged(ItemEvent e);
  public void keyPressed(KeyEvent e);
  public void keyReleased(KeyEvent e);
  public void keyTyped(KeyEvent e);
  public void mouseClicked(MouseEvent e);
  public void mouseDragged(MouseEvent e);
  public void mouseEntered(MouseEvent e);
  public void mouseExited(MouseEvent e);
  public void mouseMoved(MouseEvent e);
  public void mousePressed(MouseEvent e);
  public void mouseReleased(MouseEvent e);
  public void textValueChanged(TextEvent e);
  public void windowActivated(WindowEvent e);
  public void windowClosed(WindowEvent e);
  public void windowClosing(WindowEvent e);
  public void windowDeactivated(WindowEvent e);
  public void windowDeiconified(WindowEvent e);
  public void windowIconified(WindowEvent e);
  public void windowOpened(WindowEvent e);
  // Protected Instance Methods
  protected EventListener remove(EventListener oldl);
  protected void saveInternal(ObjectOutputStream s, String k) throws IOException;
}
```

Variables

a

protected EventListener a

One of the `EventListener`s this `AWTEventMulticaster` sends events to.

b

protected EventListener b

One of the `EventListener`s this `AWTEventMulticaster` sends events to.

Constructors

AWTEventMulticaster

protected AWTEventMulticaster (EventListener a, EventListener b)

Parameters

a

A listener that receives events.

b

A listener that receives events.

Description

Constructs an `AWTEventMulticaster` that sends events it receives to the supplied listeners. The constructor is protected because it is only the class methods of `AWTEventMulticaster` that ever instantiate this class.

Class Methods

add

public static ActionListener add (ActionListener a, ActionListener b)

Parameters

a

An event listener.

b

An event listener.

Returns

A listener object that passes events to a and b.

public static AdjustmentListener add (AdjustmentListener a, AdjustmentListener b)

Parameters

a

An event listener.

b

An event listener.

Returns

A listener object that passes events to a and b.

public static ComponentListener add (ComponentListener a, ComponentListener b)

Parameters

a

An event listener.

b

An event listener.

Returns

A listener object that passes events to a and b.

public static ContainerListener add (ContainerListener a, ContainerListener b)

Parameters

a

An event listener.

b

An event listener.

Returns

A listener object that passes events to a and b.

public static FocusListener add (FocusListener a, FocusListener b)

Parameters

a

An event listener.

b

An event listener.

Returns

A listener object that passes events to a and b.

public static ItemListener add (ItemListener a, ItemListener b)

Parameters

a

An event listener.

b

An event listener.

Returns

A listener object that passes events to a and b.

public static KeyListener add (KeyListener a, KeyListener b)

Parameters

a

An event listener.

b

An event listener.

Returns

A listener object that passes events to `a` and `b`.

public static MouseListener add (MouseListener a, MouseListener b)

Parameters

a

An event listener.

b

An event listener.

Returns

A listener object that passes events to `a` and `b`.

public static MouseMotionListener add (MouseMotionListener a, MouseMotionListener b)

Parameters

a

An event listener.

b

An event listener.

Returns

A listener object that passes events to `a` and `b`.

public static TextListener add (TextListener a, TextListener b)

Parameters

a

An event listener.

b

An event listener.

Returns

A listener object that passes events to `a` and `b`.

public static WindowListener add (WindowListener a, WindowListener b)

Parameters

a

An event listener.

b

An event listener.

Returns

A listener object that passes events to `a` and `b`.

addInternal

public static EventListener addInternal (EventListener a, EventListener b)

Parameters

a

An event listener.

b

An event listener.

Returns

A listener object that passes events to `a` and `b`.

Description

This method is a helper for the `add()` methods.

remove

public static ActionListener remove (ActionListener l, ActionListener oldl)

Parameters

l

An event listener.

oldl

An event listener.

Returns

A listener object that multicasts to `l` but not `oldl`.

public static AdjustmentListener remove (AdjustmentListener l, AdjustmentListener oldl)

Parameters

l

An event listener.

oldl

An event listener.

Returns

A listener object that multicasts to `l` but not `oldl`.

public static ComponentListener remove (ComponentListener l, ComponentListener oldl)

Parameters

l

An event listener.

oldl

An event listener.

Returns

A listener object that multicasts to `l` but not `oldl`.

public static ContainerListener remove (ContainerListener l, ContainerListener oldl)

Parameters

l

An event listener.

oldl

An event listener.

Returns

A listener object that multicasts to `l` but not `oldl`.

public static FocusListener remove (FocusListener l, FocusListener oldl)

Parameters

l

An event listener.

oldl

An event listener.

Returns

A listener object that multicasts to `l` but not `oldl`.

public static ItemListener remove (ItemListener l, ItemListener oldl)

Parameters

l

An event listener.

oldl

An event listener.

Returns

A listener object that multicasts to `l` but not `oldl`.

public static KeyListener remove (KeyListener l, KeyListener oldl)

Parameters

l

An event listener.

oldl

An event listener.

Returns

A listener object that multicasts to `l` but not `oldl`.

public static MouseListener remove (MouseListener l, MouseListener oldl)

Parameters

l

An event listener.

oldl

An event listener.

Returns

A listener object that multicasts to `l` but not `oldl`.

public static MouseMotionListener remove (MouseMotionListener l, MouseMotionListener oldl)

Parameters

l

An event listener.

oldl

An event listener.

Returns

A listener object that multicasts to `l` but not `oldl`.

public static TextListener remove (TextListener l, TextListener oldl)

Parameters

l

An event listener.

oldl

An event listener.

Returns

A listener object that multicasts to `l` but not `oldl`.

public static WindowListener remove (WindowListener l, WindowListener oldl)

Parameters

l

An event listener.

oldl

An event listener.

Returns

A listener object that multicasts to `l` but not `oldl`.

public static WindowListener remove (WindowListener l, WindowListener oldl)

Parameters

l

An event listener.

oldl

An event listener.

Returns

A listener object that multicasts to `l` but not `oldl`.

removeInternal

public static EventListener removeInternal (EventListener l, EventListener oldl)

Parameters

l

An event listener.

oldl

An event listener.

Returns

A listener object that multicasts to `l` but not `oldl`.

Description

This method is a helper for the `remove()` methods.

Instance Methods

actionPerformed

public void actionPerformed (ActionEvent e)

Parameters

e

The action event that occurred.

Description

Handles the event by passing it on to listeners `a` and `b`.

adjustmentValueChanged

public void adjustmentValueChanged (AdjustmentEvent e)

Parameters

e

The adjustment event that occurred.

Description

Handles the event by passing it on to listeners a and b.

componentAdded

public void componentAdded (ContainerEvent e)

Parameters

e

The container event that occurred.

Description

Handles the event by passing it on to listeners a and b.

componentHidden

public void componentHidden (ComponentEvent e)

Parameters

e

The component event that occurred.

Description

Handles the event by passing it on to listeners a and b.

componentMoved

public void componentMoved (ComponentEvent e)

Parameters

e

The component event that occurred.

Description

Handles the event by passing it on to listeners a and b.

componentRemoved

public void componentRemoved (ContainerEvent e)

Parameters

e

The container event that occurred.

Description

Handles the event by passing it on to listeners a and b.

componentResized

public void componentResized (ComponentEvent e)

Parameters

e

The component event that occurred.

Description

Handles the event by passing it on to listeners a and b.

componentShown

public void componentShown (ComponentEvent e)

Parameters

e

The component event that occurred.

Description

Handles the event by passing it on to listeners a and b.

focusGained

public void focusGained (FocusEvent e)

Parameters

e

The focus event that occurred.

Description

Handles the event by passing it on to listeners a and b.

focusLost

public void focusLost (FocusEvent e)

Parameters

e

The focus event that occurred.

Description

Handles the event by passing it on to listeners a and b.

itemStateChanged

public void itemStateChanged (ItemEvent e)

Parameters

e

The item event that occurred.

Description

Handles the event by passing it on to listeners a and b.

keyPressed

public void keyPressed (KeyEvent e)

Parameters

e

The key event that occurred.

Description

Handles the event by passing it on to listeners a and b.

keyReleased

public void keyReleased (KeyEvent e)

Parameters

e

The key event that occurred.

Description

Handles the event by passing it on to listeners a and b.

keyTyped

public void keyTyped (KeyEvent e)

Parameters

e

The key event that occurred.

Description

Handles the event by passing it on to listeners a and b.

mouseClicked

public void mouseClicked (MouseEvent e)

Parameters

e

The mouse event that occurred.

Description

Handles the event by passing it on to listeners a and b.

mouseDragged

public void mouseDragged (MouseEvent e)

Parameters

e

The mouse event that occurred.

Description

Handles the event by passing it on to listeners a and b.

mouseEntered

public void mouseEntered (MouseEvent e)

Parameters

e

The mouse event that occurred.

Description

Handles the event by passing it on to listeners a and b.

mouseExited

public void mouseExited (MouseEvent e)

Parameters

e

The mouse event that occurred.

Description

Handles the event by passing it on to listeners a and b.

mouseMoved

public void mouseMoved (MouseEvent e)

Parameters

e

The mouse event that occurred.

Description

Handles the event by passing it on to listeners a and b.

mousePressed

public void mousePressed (MouseEvent e)

Parameters

e

The mouse event that occurred.

Description

Handles the event by passing it on to listeners a and b.

mouseReleased

public void mouseReleased (MouseEvent e)

Parameters

e

The mouse event that occurred.

Description

Handles the event by passing it on to listeners a and b.

textValueChanged

public void textValueChanged (TextEvent e)

Parameters

e

The text event that occurred.

Description

Handles the event by passing it on to listeners a and b.

windowActivated

public void windowActivated (WindowEvent e)

Parameters

e

The window event that occurred.

Description

Handles the event by passing it on to listeners a and b.

windowClosed

public void windowClosed (WindowEvent e)

Parameters

e

The window event that occurred.

Description

Handles the event by passing it on to listeners a and b.

windowClosing

public void windowClosing (WindowEvent e)

Parameters

e

The window event that occurred.

Description

Handles the event by passing it on to listeners a and b.

windowDeactivated

public void windowDeactivated (WindowEvent e)

Parameters

e

The window event that occurred.

Description

Handles the event by passing it on to listeners a and b.

windowDeiconified

public void windowDeiconified (WindowEvent e)

Parameters

e

The window event that occurred.

Description

Handles the event by passing it on to listeners a and b.

windowIconified

public void windowIconified (WindowEvent e)

Parameters

e

The window event that occurred.

Description

Handles the event by passing it on to listeners `a` and `b`.

windowOpened

public void windowOpened (WindowEvent e)

Parameters

e

The window event that occurred.

Description

Handles the event by passing it on to listeners `a` and `b`.

Protected Instance Methods

remove

protected EventListener remove(EventListener oldl)

Parameters

oldl

The listener to remove.

Returns

The resulting `EventListener`.

Description

This method removes `oldl` from the `AWTEventMulticaster` and returns the resulting listener.

See Also

`ActionEvent`, `AdjustmentEvent`, `ComponentEvent`, `Event`, `EventListener`, `EventObject`, `FocusEvent`, `ItemEvent`, `KeyEvent`, `MouseEvent`, `WindowEvent`

PREVIOUS HOME NEXT

AWTException

Name

AWTException

[Graphic: Figure from the text]

Description

An `AWTException`; thrown to indicate an exceptional condition; must be caught or declared in a `throws` clause.

Class Definition

```
public class java.awt.AWTException
    extends java.lang.Exception {

  // Constructors
  public AWTException (String message);
}
```

Constructors

AWTException

public AWTException (String message)

Parameters

message

Detailed message.

See Also

`Exception`, `String`

PREVIOUS | HOME | NEXT
AWTEventMulticaster ★ | BOOK INDEX | Adjustable ★

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Adjustable ★

Name

Adjustable ★

[Graphic: Figure from the text]

Description

The `Adjustable` interface is useful for scrollbars, sliders, dials, and other components that have an adjustable numeric value. Classes that implement the `Adjustable` interface should send `AdjustmentEvent` objects to listeners that have registered via `addAdjustmentListener(AdjustmentListener)`.

Interface Definition

```
public abstract interface java.awt.Adjustable {
  // Constants
  public final static int HORIZONTAL = 0;
  public final static int VERTICAL = 1;

  // Interface Methods
  public abstract void addAdjustmentListener (AdjustmentListener l);
  public abstract int getBlockIncrement();
  public abstract int getMaximum();
  public abstract int getMinimum();
  public abstract int getOrientation();
  public abstract int getUnitIncrement();
  public abstract int getValue();
  public abstract int getVisibleAmount();
  public abstract void removeAdjustmentListener (AdjustmentListener l);
  public abstract void setBlockIncrement (int b);
  public abstract void setMaximum (int max);
  public abstract void setMinimum (int min);
  public abstract void setUnitIncrement (int u);
  public abstract void setValue (int v);
  public abstract void setVisibleAmount (int v);
}
```

Constants

HORIZONTAL

public static final int HORIZONTAL

A constant representing horizontal orientation.

VERTICAL

public static final int VERTICAL

A constant representing vertical orientation.

Interface Methods

addAdjustmentListener

public abstract void addAdjustmentListener (ActionListener l)

Parameters

l

An object that implements the `AdjustmentListener` interface.

Description

Add a listener for adjustment event.

getBlockIncrement

public abstract int getBlockIncrement()

Returns

The amount to scroll when a paging area is selected.

getMaximum

public abstract int getMaximum()

Returns

The maximum value that the `Adjustable` object can take.

getMinimum

public abstract int getMinimum()

Returns

The minimum value that the `Adjustable` object can take.

getOrientation

public abstract int getOrientation()

Returns

A value representing the direction of the `Adjustable` object.

getUnitIncrement

public abstract int getUnitIncrement()

Returns

The unit amount to scroll.

getValue

public abstract int getValue()

Returns

The current setting for the `Adjustable` object.

getVisibleAmount

public abstract int getVisibleAmount()

Returns

The current visible setting (i.e., size) for the `Adjustable` object.

removeAdjustmentListener

public abstract void removeAdjustmentListener (AdjustmentListener l)

Parameters

l

One of the object's `AdjustmentListener`s.

Description

Remove an adjustment event listener.

setBlockIncrement

public abstract void setBlockIncrement (int b)

Parameters

b

New block increment amount.

Description

Changes the block increment amount for the `Adjustable` object.

setMaximum

public abstract void setMaximum (int max)

Parameters

max

New maximum value.

Description

Changes the maximum value for the `Adjustable` object.

setMinimum

public abstract void setMinimum (int min)

Parameters

min

New minimum value.

Description

Changes the minimum value for the `Adjustable` object.

setUnitIncrement

public abstract void setUnitIncrement (int u)

Parameters

u

New unit increment amount.

Description

Changes the unit increment amount for the `Adjustable` object.

setValue

public abstract void setValue (int v)

Parameters

v

New value.

Description

Changes the current value of the `Adjustable` object.

setVisibleAmount

public abstract void setVisibleAmount (int v)

Parameters

v

New amount visible.

Description

Changes the current visible amount of the `Adjustable` object.

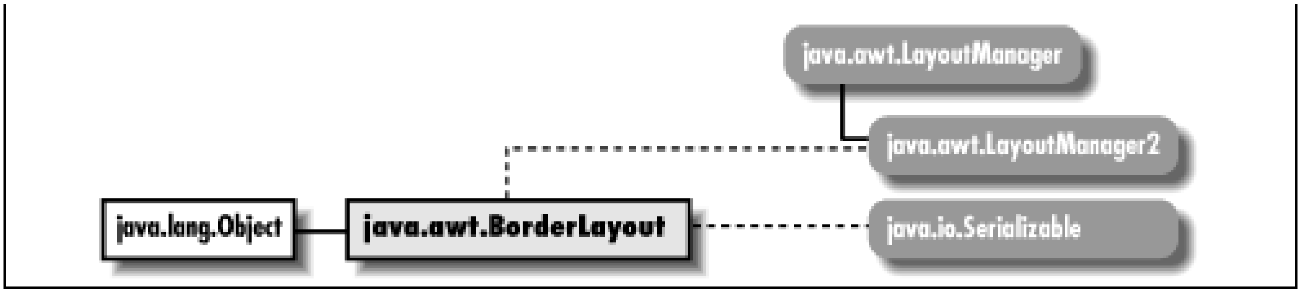
See Also

`AdjustmentEvent`, `AdjustmentListener`, `Scrollbar`

BorderLayout

Name

BorderLayout



Description

`BorderLayout` is a `LayoutManager` that provides the means to lay out components along the edges of a container. It divides the container into five regions, named North, East, South, West, and Center. Normally you won't call the `LayoutManager`'s methods yourself. When you `add()` a `Component` to a `Container`, the `Container` calls the `addLayoutComponent()` method of its `LayoutManager`.

Class Definition

```
public class java.awt.BorderLayout
    extends java.lang.Object
    implements java.awt.LayoutManager2, java.io.Serializable {
  // Constants
  public final static String CENTER; ★
  public final static String EAST; ★
  public final static String NORTH; ★
  public final static String SOUTH; ★
  public final static String WEST; ★

  // Constructors
  public BorderLayout();
  public BorderLayout (int hgap, int vgap);

  // Instance Methods
  public void addLayoutComponent (Component comp, Object constraints); ★
  public void addLayoutComponent (String name, Component component); ☆
  public int getHgap(); ★
  public abstract float getLayoutAlignmentX(Container target); ★
  public abstract float getLayoutAlignmentY(Container target); ★
  public int getVgap(); ★
  public abstract void invalidateLayout(Container target); ★

  public void layoutContainer (Container target);
  public abstract Dimension maximumLayoutSize(Container target); ★
  public Dimension minimumLayoutSize (Container target);
  public Dimension preferredLayoutSize (Container target);
  public void removeLayoutComponent (Component component);
  public void setHgap (int hgap); ★
  public void setVgap (int vgap); ★
  public String toString();
}
```

Constants

CENTER

public final static String CENTER

A constant representing center orientation.

EAST

public final static String EAST

A constant representing east orientation.

NORTH

public final static String NORTH

A constant representing north orientation.

SOUTH

public final static String SOUTH

A constant representing south orientation.

WEST

public final static String WEST

A constant representing west orientation.

Constructors

BorderLayout

public BorderLayout()

Description

> Constructs a `BorderLayout` object.

public BorderLayout (int hgap, int vgap)

Parameters

> *hgap*
>
> > Horizontal space between each component in the container.
>
> *vgap*
>
> > Vertical space between each component in the container.

Description

Constructs a `BorderLayout` object with the values specified as the gaps between each component in the container managed by this instance of `BorderLayout`.

Instance Methods

addLayoutComponent

public void addLayoutComponent (Component comp, Object constraints) ★

Parameters

comp

The component being added.

constraints

An object describing the constraints on this component.

Implements

`LayoutManager2.addLayoutComponent()`

Description

Adds the component `comp` to a container subject to the given `constraints`. This is a more general version of `addLayoutComponent(String, Component)` method. It corresponds to `java.awt.Container`'s `add(Component, Object)` method. In practice, it is used the same in version 1.1 as in Java 1.0.2, except with the parameters swapped:

```
Panel p = new Panel(new BorderLayout());
p.add(new Button("OK"), BorderLayout.SOUTH);
```

addLayoutComponent

public void addLayoutComponent (String name, Component component) ☆

Parameters

name

Name of region to add component to.

component

Actual component being added.

Implements

`LayoutManager.addLayoutComponent()`

Description

Adds a `component` to a container in region `name`. This has been replaced in version 1.1 with the more general `addLayoutComponent(Component, Object)`.

getHgap

public int getHgap() ★

Returns

The horizontal gap for this `BorderLayout` instance.

getLayoutAlignmentX

public abstract float getLayoutAlignmentX (Container target) ★

Parameters

target

The container to inspect.

Returns

The value .5 for all containers.

Description

This method returns the preferred alignment of the given container `target`. A return value of 0 is left aligned, .5 is centered, and 1 is right aligned.

getLayoutAlignmentY

public abstract float getLayoutAlignmentY (Container target) ★

Parameters

target

The container to inspect.

Returns

The value .5 for all containers.

Description

This method returns the preferred alignment of the given container `target`. A return value of 0 is top aligned, .5 is centered, and 1 is bottom aligned.

getVgap

public int getVgap() ★

Returns

The vertical gap for this `BorderLayout` instance.

invalidateLayout

public abstract void invalidateLayout (Container target) ★

Parameters

target

The container to invalidate.

Description

Does nothing.

layoutContainer

public void layoutContainer (Container target)

Parameters

target

The container that needs to be redrawn.

Implements

`LayoutManager.layoutContainer()`

Description

Draws components contained within `target`.

maximumLayoutSize

public abstract Dimension maximumLayoutSize (Container target) ★

Parameters

target

The container to inspect.

Returns

A `Dimension` whose horizontal and vertical components are `Integer.MAX_VALUE`.

Description

For `BorderLayout`, a maximal `Dimension` is always returned.

minimumLayoutSize

public Dimension minimumLayoutSize (Container target)

Parameters

target

The container whose size needs to be calculated.

Returns

Minimum `Dimension` of the container `target`.

Implements

`LayoutManager.minimumLayoutSize()`

Description

Calculates minimum size of `target`. container.

preferredLayoutSize

public Dimension preferredLayoutSize (Container target)

Parameters

target

The container whose size needs to be calculated.

Returns

Preferred `Dimension` of the container `target`.

Implements

`LayoutManager.preferredLayoutSize()`

Description

Calculates preferred size of `target` container.

removeLayoutComponent

public void removeLayoutComponent (Component component)

Parameters

component

Component to stop tracking.

Implements

`LayoutManager.removeLayoutComponent()`

Description

Removes `component` from any internal tracking systems.

setHgap

public void setHgap (int hgap) ★

Parameters

hgap

The horizontal gap value.

Description

Sets the horizontal gap between components.

setVgap

public void setVgap (int vgap) ★

Parameters

vgap

The vertical gap value.

Description

Sets the vertical gap between components.

toString

public String toString()

Returns

A string representation of the `BorderLayout` object.

Overrides

`Object.toString()`

See Also

`Component`, `Container`, `Dimension`, `LayoutManager`, `LayoutManager2`, `Object`, `String`

PREVIOUS HOME NEXT
Adjustable ★ BOOK INDEX Button

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Button

Name

Button

[Graphic: Figure from the text]

Description

The `Button` is the familiar labeled button object. It inherits most of its functionality from `Component`. For example, to change the font of the `Button`, you would use `Component`'s `setFont()` method. The `Button` sends `java.awt.event.ActionEvent` objects to its listeners when it is pressed.

Class Definition

```
public class java.awt.Button
    extends java.awt.Component {

  // Constructors
  public Button();
  public Button (String label);

  // Instance Methods
  public void addActionListener (ActionListener l); ★
  public void addNotify();
  public String getActionCommand(); ★
  public String getLabel();
  public void removeActionListener (ActionListener l); ★
  public void setActionCommand (String command); ★
  public synchronized void setLabel (String label);

  // Protected Instance Methods
  protected String paramString();
  protected void processActionEvent (ActionEvent e); ★
  protected void processEvent (AWTEvent e); ★
}
```

Constructors

Button

public Button()

Description

> Constructs a `Button` object with no label.

public Button (String label)

Parameters

> *label*
>
> > The text for the label on the button

Description

> Constructs a `Button` object with text of `label`.

Instance Methods

addActionListener

public void addActionListener (ActionListener l) ★

Parameters

l

An object that implements the `ActionListener` interface.

Description

Add a listener for the action event.

addNotify

public void addNotify()

Overrides

`Component.addNotify()`

Description

Creates `Button`'s peer.

getActionCommand

public String getActionCommand() ★

Returns

Current action command string.

Description

Returns the string used for the action command.

getLabel

public String getLabel()

Returns

Text of the `Button`'s label.

removeActionListener

public void removeActionListener (ActionListener l) ★

Parameters

l

One of this `Button`'s `ActionListener`s.

Description

Remove an action event listener.

setActionCommand

public void setActionCommand (String command) ★

Parameters

command

New action command string.

Description

Specify the string used for the action command.

setLabel

public synchronized void setLabel (String label)

Parameters

label

New text for label of `Button`.

Description

Changes the `Button`'s label to `label`.

Protected Instance Methods

paramString

protected String paramString()

Returns

String with current settings of `Button`.

Overrides

`Component.paramString()`

Description

Helper method for `toString()` used to generate a string of current settings.

processActionEvent

protected void processActionEvent (ActionEvent e) ★

Parameters

e

The action event to process.

Description

Action events are passed to this method for processing. Normally, this method is called by `processEvent()`.

processEvent

protected void processEvent (AWTEvent e) ★

Parameters

e

The event to process.

Description

Low level `AWTEvent`s are passed to this method for processing.

See Also

`ActionListener`, `Component`, `String`

PREVIOUS | HOME | NEXT
BorderLayout | BOOK INDEX | Canvas

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Canvas

Name

Canvas

[Graphic: Figure from the text]

Description

`Canvas` is a `Component` that provides a drawing area and is often used as a base class for new components.

Class Definition

```
public class java.awt.Canvas
    extends java.awt.Component {

  // Constructors
  public Canvas();

  // Instance Methods
  public void addNotify();
  public void paint (Graphics g);
}
```

Constructors

Canvas

public Canvas()

Description

Constructs a `Canvas` object.

Instance Methods

addNotify

public void addNotify()

Overrides

`Component.addNotify()`

Description

Creates `Canvas`'s peer.

paint

public void paint (Graphics g)

Parameters

g

Graphics context of component.

Description

Empty method to be overridden in order to draw something in graphics context.

See Also

`Component`, `Graphics`

PREVIOUS | HOME | NEXT
Button | BOOK INDEX | CardLayout

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

CardLayout

Name

CardLayout

[Graphic: Figure from the text]

Description

The `CardLayout LayoutManager` provides the means to manage multiple components, displaying one at a time. Components are displayed in the order in which they are added to the layout, or in an arbitrary order by using an assignable name.

Class Definition

```
public class java.awt.CardLayout
    extends java.lang.Object
    implements java.awt.LayoutManager2, java.io.Serializable {

  // Constructors
  public CardLayout();
  public CardLayout (int hgap, int vgap);
  // Instance Methods
  public void addLayoutComponent (Component comp,
      Object constraints); ★
  public void addLayoutComponent (String name, Component component); ☆
  public void first (Container parent);
  public int getHgap(); ★
  public abstract float getLayoutAlignmentX(Container target); ★
  public abstract float getLayoutAlignmentY(Container target); ★
  public int getVgap(); ★
  public abstract void invalidateLayout(Container target); ★
  public void last (Container parent);
  public void layoutContainer (Container target);
  public abstract Dimension maximumLayoutSize(Container target); ★
  public Dimension minimumLayoutSize (Container target);
  public void next (Container parent);
  public Dimension preferredLayoutSize (Container target);
  public void previous (Container parent);
  public void removeLayoutComponent (Component component);
  public void setHgap (int hgap); ★
  public void setVgap (int vgap); ★
  public void show (Container parent, String name);
  public String toString();
}
```

Constructors

CardLayout

public CardLayout()

Description

Constructs a `CardLayout` object.

public CardLayout (int hgap, int vgap)

Parameters

hgap

Horizontal space around left and right of container

vgap

Vertical space around top and bottom of container

Description

Constructs a `CardLayout` object with the values specified as the gaps around the container managed by this instance of `CardLayout`.

Instance Methods

addLayoutComponent

public void addLayoutComponent (Component comp, Object constraints) ★

Parameters

comp

The component being added.

constraints

An object describing the constraints on this component.

Implements

`LayoutManager2.addLayoutComponent()`

Description

Adds the component `comp` to a container subject to the given `constraints`. This is a more generalized version of `addLayoutComponent(String, Component)`. It corresponds to `java.awt.Container's add(Component, Object)`. In practice, it is used the same in Java 1.1 as in Java 1.0.2, except with the parameters swapped:

```
Panel p = new Panel();
p.setLayoutManager(new CardLayout());
p.add(new Button("OK"), "Don Julio");
```

addLayoutComponent

public void addLayoutComponent (String name, Component component) ☆

Parameters

name

Name of the component to add.

component

The actual component being added.

Implements

```
LayoutManager.addLayoutComponent()
```

Description

Places `component` under the layout's management, assigning it the given `name`. This has been replaced in version 1.1 with the more general `addLayoutComponent(Component, Object)`.

first

public void first (Container parent)

Parameters

parent

The container whose displayed component is changing.

Throws

`IllegalArgumentException`

If the `LayoutManager` of `parent` is not `CardLayout`.

Description

Sets the container to display the first component in parent.

getHgap

public int getHgap() ★

Returns

The horizontal gap for this `CardLayout` instance.

getLayoutAlignmentX

public abstract float getLayoutAlignmentX (Container target) ★

Parameters

target

The container to inspect.

Returns

The value .5 for all containers.

Description

This method returns the preferred alignment of the given container `target`. A return value of 0 is left aligned, .5 is centered, and 1 is right aligned.

getLayoutAlignmentY

public abstract float getLayoutAlignmentY (Container target) ★

Parameters

target

The container to inspect.

Returns

The value .5 for all containers.

Description

This method returns the preferred alignment of the given container `target`. A return value of 0 is top aligned, .5 is centered, and 1 is bottom aligned.

getVgap

public int getVgap() ★

Returns

The vertical gap for this `CardLayout` instance.

invalidateLayout

public abstract void invalidateLayout (Container target) ★

Parameters

target

The container to invalidate.

Description

Does nothing.

last

public void last (Container parent)

Parameters

parent

The container whose displayed component is changing.

Throws

`IllegalArgumentException`

If the `LayoutManager` of `parent` is not `CardLayout`.

Description

Sets the container to display the final component in parent.

layoutContainer

public void layoutContainer (Container target)

Parameters

target

The container that needs to be redrawn.

Implements

`LayoutManager.layoutContainer()`

Description

Displays the currently selected component contained within `target`.

maximumLayoutSize

public abstract Dimension maximumLayoutSize (Container target) ★

Parameters

target

The container to inspect.

Returns

A `Dimension` whose horizontal and vertical components are `Integer.MAX_VALUE`.

Description

For `CardLayout`, a maximal `Dimension` is always returned.

minimumLayoutSize

public Dimension minimumLayoutSize (Container target)

Parameters

target

The container whose size needs to be calculated.

Returns

Minimum `Dimension` of the container `target`.

Implements

`LayoutManager.minimumLayoutSize()`

Description

Calculates minimum size of the `target` container.

next

public void next (Container parent)

Parameters

parent

The container whose displayed component is changing.

Throws

`IllegalArgumentException`

If the `LayoutManager` of `parent` is not `CardLayout`.

Description

Sets the container to display the following component in the parent.

preferredLayoutSize

public Dimension preferredLayoutSize (Container target)

Parameters

target

The container whose size needs to be calculated.

Returns

Preferred `Dimension of` the container `target`.

Implements

`LayoutManager.preferredLayoutSize()`

Description

Calculates preferred size of the `target` container.

previous

public void previous (Container parent)

Parameters

parent

The container whose displayed component is changing.

Throws

`IllegalArgumentException`

If the `LayoutManager` of `parent` is not `CardLayout`.

Description

Sets the container to display the prior component in `parent`.

removeLayoutComponent

public void removeLayoutComponent (Component component)

Parameters

component

Component to stop tracking.

Implements

`LayoutManager.removeLayoutComponent()`

Description

Removes `component` from the layout manager's internal tables.

setHgap

public void setHgap (int hgap) ★

Parameters

hgap

The horizontal gap value.

Description

Sets the horizontal gap for the left and right of the container.

setVgap

public void setVgap (int vgap) ★

Parameters

vgap

The vertical gap value.

Description

Sets the vertical gap for the top and bottom of the container.

show

public void show (Container parent, String name)

Parameters

parent

The container whose displayed component is changing.

name

Name of component to display.

Throws

`IllegalArgumentException`

If `LayoutManager` of `parent` is not `CardLayout`.

Description

Sets the container to display the component `name` in `parent`.

toString

public String toString()

Returns

A string representation of the `CardLayout` object.

Overrides

`Object.toString()`

See Also

`Component`, `Container`, `Dimension`, `LayoutManager`, `LayoutManager2`, `Object`, `String`

Checkbox

Name

Checkbox

[Graphic: Figure from the text]

Description

The `Checkbox` is a `Component` that provides a true or false toggle switch for user input.

Class Definition

```
public class java.awt.Checkbox
    extends java.awt.Component
    implements java.awt.ItemSelectable {

  // Constructors
  public Checkbox();
  public Checkbox (String label);
  public Checkbox (String label, boolean state); ★
  public Checkbox (String label, boolean state, CheckboxGroup group); ★
  public Checkbox (String label, CheckboxGroup group, boolean state);

  // Instance Methods
  public void addItemListener (ItemListener l); ★
  public void addNotify();
  public CheckboxGroup getCheckboxGroup();
  public String getLabel();
  public Object[] getSelectedObjects(); ★
  public boolean getState();
  public void removeItemListener (ItemListener l); ★
  public void setCheckboxGroup (CheckboxGroup group);
  public synchronized void setLabel (String label);
  public void setState (boolean state);

  // Protected Instance Methods
  protected String paramString();
  protected void processEvent (AWTEvent e); ★
  protected void processItemEvent (ItemEvent e); ★
}
```

Constructors

Checkbox

public Checkbox()

Description

> Constructs a `Checkbox` object with no label that is initially `false`.

public Checkbox (String label)

Parameters

> *label*
>
> > Text to display with the `Checkbox`.

Description

> Constructs a `Checkbox` object with the given `label` that is initially `false`.

public Checkbox (String label, boolean state) ★

Parameters

label

Text to display with the `Checkbox`.

state

Intial value of the `Checkbox`.

Description

Constructs a `Checkbox` with the given `label`, initialized to the given `state`.

public Checkbox (String label, boolean state, CheckboxGroup group) ★

Parameters

label

Text to display with the `Checkbox`.

state

Intial value of the `Checkbox`.

group

The `CheckboxGroup` this `Checkbox` should belong to.

Description

Constructs a `Checkbox` with the given `label`, initialized to the given `state` and belonging to `group`.

public Checkbox (String label, CheckboxGroup group, boolean state)

Parameters

label

Text to display with the `Checkbox`.

group

The `CheckboxGroup` this `Checkbox` should belong to.

state

Intial value of the `Checkbox`.

Description

Constructs a `Checkbox` object with the given settings.

Instance Methods

addItemListener

public void addItemListener (ItemListener l) ★

Parameters

l

The listener to be added.

Implements

`ItemSelectable.addItemListener(ItemListener l)`

Description

Adds a listener for the `ItemEvent` objects this `Checkbox` generates.

addNotify

public void addNotify()

Overrides

`Component.addNotify()`

Description

Creates `Checkbox` peer.

getCheckboxGroup

public CheckboxGroup getCheckboxGroup()

Returns

The current `CheckboxGroup` associated with the `Checkbox`, if any.

getLabel

public String getLabel()

Returns

The text associated with the `Checkbox`.

getSelectedObjects

public Object[] getSelectedObjects() ★

Implements

`ItemSelectable.getSelectedObjects()`

Description

If the `Checkbox` is checked, returns an array with length 1 containing the label of the `Checkbox`; otherwise returns `null`.

getState

public boolean getState()

Returns

The current state of the `Checkbox`.

removeItemListener

public void removeItemListener (ItemListener l) ★

Parameters

l

The listener to be removed.

Implements

`ItemSelectable.removeItemListener (ItemListener l)`

Description

Removes the specified `ItemListener` so it will not receive `ItemEvent` objects from this `Checkbox`.

setCheckboxGroup

public void setCheckboxGroup (CheckboxGroup group)

Parameters

group

New group in which to place the `Checkbox`.

Description

Associates the `Checkbox` with a different `CheckboxGroup`.

setLabel

public synchronized void setLabel (String label)

Parameters

label

New text to associate with `Checkbox`.

Description

Changes the text associated with the `Checkbox`.

setState

public void setState (boolean state)

Parameters

state

New state for the `Checkbox`.

Description

Changes the state of the `Checkbox`.

Protected Instance Methods

paramString

protected String paramString()

Returns

String with current settings of `Checkbox`.

Overrides

`Component.paramString()`

Description

Helper method for `toString()` to generate string of current settings.

processEvent

protected void processEvent(AWTEvent e) ★

Parameters

e

The event to process.

Description

Low level `AWTEvents` are passed to this method for processing.

processItemEvent

protected void processItemEvent(ItemEvent e) ★

Parameters

e

The item event to process.

Description

Item events are passed to this method for processing. Normally, this method is called by `processEvent()`.

See Also

`CheckboxGroup`, `Component`, `ItemEvent`, `ItemSelectable`, `String`

CheckboxGroup

Name

CheckboxGroup

[Graphic: Figure from the text]

Description

The `CheckboxGroup` class provides the means to group multiple `Checkbox` items into a mutual exclusion set, so that only one checkbox in the set has the value `true` at any time. The checkbox with the value `true` is the currently selected checkbox. Mutually exclusive checkboxes usually have a different appearance from regular checkboxes and are also called "radio buttons."

Class Definition

```
public class java.awt.CheckboxGroup
    extends java.lang.Object
    implements java.io.Serializable {

  // Constructors
  public CheckboxGroup();

  // Instance Methods
  public Checkbox getCurrent(); ☆
  public Checkbox getSelectedCheckbox() ★
  public synchronized void setCurrent (Checkbox checkbox); ☆
  public synchronized void setSelectedCheckbox (Checkbox checkbox); ★
  public String toString();
}
```

Constructors

CheckboxGroup

public CheckboxGroup()

Description

> Constructs a `CheckboxGroup` object.

Instance Methods

getCurrent

public Checkbox getCurrent() ☆

Returns

> The currently selected `Checkbox` within the `CheckboxGroup`.

Description

> Replaced by the more aptly named `getSelectedCheckbox()`.

getSelectedCheckbox

public Checkbox getSelectedCheckbox() ★

Returns

> The currently selected `Checkbox` within the `CheckboxGroup`.

setCurrent

public synchronized void setCurrent (Checkbox checkbox) ☆

Parameters

checkbox

The `Checkbox` to select.

Description

Changes the currently selected `Checkbox` within the `CheckboxGroup`.

Description

Replaced by `setSelectedCheckbox(Checkbox)`.

setSelectedCheckbox

public synchronized void setSelectedCheckbox (Checkbox checkbox) ★

Parameters

checkbox

The `Checkbox` to select.

Description

Changes the currently selected `Checkbox` within the `CheckboxGroup`.

toString

public String toString()

Returns

A string representation of the `CheckboxGroup` object.

Overrides

```
    Object.toString()
```

See Also

`Checkbox`, `Object`, `String`

PREVIOUS | HOME | NEXT
Checkbox | BOOK INDEX | CheckboxMenuItem

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

CheckboxMenuItem

Name

CheckboxMenuItem

[Graphic: Figure from the text]

Description

The `CheckboxMenuItem` class represents a menu item with a boolean state.

Class Definition

```
public class java.awt.CheckboxMenuItem
    extends java.awt.MenuItem
    implements java.awt.ItemSelectable {

  // Constructors
  public CheckboxMenuItem(); ★
  public CheckboxMenuItem (String label);
  public CheckboxMenuItem (String label, boolean state); ★

  // Instance Methods
  public void addItemListener (ItemListener l); ★
  public void addNotify();
  public Object[] getSelectedObjects(); ★
  public boolean getState();
  public String paramString();
  public void removeItemListener (ItemListener l); ★
  public synchronized void setState (boolean condition);

  // Protected Instance Methods
  protected void processEvent (AWTEvent e); ★
  protected void processItemEvent (ItemEvent e); ★
}
```

Constructors

CheckboxMenuItem

public CheckboxMenuItem() ★

Description

Constructs a `CheckboxMenuItem` object with no label.

public CheckboxMenuItem (String label)

Parameters

label

Text that appears on `CheckboxMenuItem`.

Description

Constructs a `CheckboxMenuItem` object whose value is initially `false`.

public CheckboxMenuItem (String label, boolean state) ★

Parameters

label

Text that appears on `CheckboxMenuItem`.

state

The initial state of the menu item.

Description

Constructs a `CheckboxMenuItem` object with the specified `label` and `state`.

Instance Methods

addItemListener

public void addItemListener (ItemListener l) ★

Parameters

l

The listener to be added.

Implements

```
ItemSelectable.addItemListener(ItemListener l)
```

Description

Adds a listener for the `ItemEvent` objects this `CheckboxMenuItem` fires off.

addNotify

public void addNotify()

Overrides

```
MenuItem.addNotify()
```

Description

Creates `CheckboxMenuItem`'s peer.

getSelectedObjects

public Object[] getSelectedObjects() ★

Implements

`ItemSelectable.getSelectedObjects()`

Description

If the `CheckboxMenuItem` is checked, returns an array with length `1` containing the label of the `CheckboxMenuItem`; otherwise returns `null`.

getState

public boolean getState()

Returns

The current state of the `CheckboxMenuItem`.

paramString

public String paramString()

Returns

A string with current settings of `CheckboxMenuItem`.

Overrides

`MenuItem.paramString()`

Description

Helper method for `toString()` to generate string of current settings.

removeItemListener

public void removeItemListener (ItemListener l) ★

Parameters

l

The listener to be removed.

Implements

`ItemSelectable.removeItemListener (ItemListener l)`

Description

Removes the specified `ItemListener` so it will not receive `ItemEvent` objects from this `CheckboxMenuItem`.

setState

public synchronized void setState (boolean condition)

Parameters

condition

New state for the `CheckboxMenuItem`.

Description

Changes the state of the `CheckboxMenuItem`.

Protected Instance Methods

processEvent

protected void processEvent(AWTEvent e) ★

Parameters

e

The event to process.

Overrides

`MenuItem.processEvent(AWTEvent)`

Description

Low level `AWTEvents` are passed to this method for processing.

processItemEvent

protected void processItemEvent(ItemEvent e) ★

Parameters

e

The item event to process.

Description

Item events are passed to this method for processing. Normally, this method is called by `processEvent()`.

See Also

`ItemEvent`, `ItemSelectable`, `MenuItem`, `String`

PREVIOUS
CheckboxGroup

HOME
BOOK INDEX

NEXT
Choice

Choice

Name

Choice

[Graphic: Figure from the text]

Description

The `Choice` is a `Component` that provides a drop-down list of choices to choose from.

Class Definition

```
public class java.awt.Choice
    extends java.awt.Component
    implements java.awt.ItemSelectable {

  // Constructors
  public Choice();

  // Instance Methods
  public synchronized void add (String item); ★
  public synchronized void addItem (String item); ☆
  public void addItemListener (ItemListener l); ★
  public void addNotify();
  public int countItems(); ☆
  public String getItem (int index);
  public int getItemCount(); ★
  public int getSelectedIndex();
  public synchronized String getSelectedItem();
  public synchronized Object[] getSelectedObjects(); ★
  public synchronized void insert (String item, int index); ★
  public synchronized void remove (int position); ★
  public synchronized void remove (String item); ★
  public synchronized void removeAll(); ★
  public void removeItemListener (ItemListener l); ★
  public synchronized void select (int pos);
  public synchronized void select (String str);

  // Protected Instance Methods
  protected String paramString();
  protected void processEvent (AWTEvent e); ★
  protected void processItemEvent (ItemEvent e); ★
}
```

Constructors

Choice

public Choice()

Description

> Constructs a `Choice` object.

Instance Methods

add

public synchronized void add (String item) ★

Parameters

item

Text for new entry.

Throws

`NullPointerException`

If `item` is null.

Description

Adds a new entry to the available choices.

addItem

public synchronized void addItem (String item) ☆

Parameters

item

Text for new entry.

Throws

`NullPointerException`

If `item` is null.

Description

Replaced by `add(String)`.

addItemListener

public void addItemListener (ItemListener l) ★

Parameters

l

The listener to be added.

Implements

`ItemSelectable.addItemListener(ItemListener l)`

Description

Adds a listener for the `ItemEvent` objects this `Choice` generates.

addNotify

public void addNotify()

Overrides

`Component.addNotify()`

Description

Creates `Choice`'s peer.

countItems

public int countItems() ☆

Returns

Number of items in the `Choice`.

Description

Replaced by `getItemCount()`.

getItem

public String getItem (int index)

Parameters

index

Position of entry.

Returns

A string for an entry at a given position.

Throws

`ArrayIndexOutOfBoundsException`

If `index` is invalid; indices start at zero.

getItemCount

public int getItemCount() ★

Returns

Number of items in the `Choice`.

getSelectedIndex

public int getSelectedIndex()

Returns

Position of currently selected entry.

getSelectedItem

public synchronized String getSelectedItem()

Returns

Currently selected entry as a String.

getSelectedObjects

public synchronized Object[] getSelectedObjects() ★

Implements

`ItemSelectable.getSelectedObjects()`

Description

A single-item array containing the current selection.

insert

public synchronized void insert (String item, int index) ★

Parameters

item

The string to add.

index

The position for the new string.

Throws

`IllegalArgumentException`

If `index` is less than zero.

Description

Inserts `item` in the given position.

remove

public synchronized void remove (int position) ★

Parameters

position

The index of an entry in the `Choice` component.

Description

Removes the entry in the given position.

public synchronized void remove (String string) ★

Parameters

string

Text of an entry within the `Choice` component.

Throws

`IllegalArgumentException`

If `string` is not in the `Choice`.

Description

Makes the first entry that matches `string` the selected item.

removeAll

public synchronized void removeAll() ★

Description

Removes all the entries from the `Choice`.

removeItemListener

public void removeItemListener (ItemListener l) ★

Parameters

l

The listener to be removed.

Implements

`ItemSelectable.removeItemListener (ItemListener l)`

Description

Removes the specified `ItemListener` so it will not receive `ItemEvent` objects from this `Choice`.

select

public synchronized void select (int pos)

Parameters

pos

The index of an entry in the `Choice` component.

Throws

`IllegalArgumentException`

If the position is not valid.

Description

Makes the entry in the given position.

public synchronized void select (String str)

Parameters

str

Text of an entry within the `Choice` component.

Description

Makes the first entry that matches `str` the selected item for the `Choice`.

Protected Instance Methods

paramString

protected String paramString()

Returns

A string with current settings of `Choice`.

Overrides

`Component.paramString()`

Description

Helper method for `toString()` to generate string of current settings.

processEvent

protected void processEvent (AWTEvent e) ★

Parameters

e

The event to process.

Description

Low level `AWTEvents` are passed to this method for processing.

processItemEvent

protected void processItemEvent (ItemEvent e) ★

Parameters

e

The item event to process.

Description

Item events are passed to this method for processing. Normally, this method is called by `processEvent()`.

See Also

`Component`, `ItemSelectable`, `String`

PREVIOUS
CheckboxMenuItem

HOME
BOOK INDEX

NEXT
Color

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Color

Name

Color

[Graphic: Figure from the text]

Description

The `Color` class represents a specific color to the system.

Class Definition

```
public final class java.awt.Color
    extends java.lang.Object
    implements java.io.Serializable {

  // Constants
  public static final Color black;
  public static final Color blue;
  public static final Color cyan;
  public static final Color darkGray;
  public static final Color gray;
  public static final Color green;
  public static final Color lightGray;
  public static final Color magenta;
  public static final Color orange;
  public static final Color pink;
  public static final Color red;
  public static final Color white;
  public static final Color yellow;

  // Constructors
  public Color (int rgb);
  public Color (int red, int green, int blue);
  public Color (float red, float green, float blue);

  // Class Methods
  public static Color decode (String name); ★
  public static Color getColor (String name);
  public static Color getColor (String name, Color defaultColor);
  public static Color getColor (String name, int defaultColor);
  public static Color getHSBColor (float hue, float saturation,
      float brightness);
  public static int HSBtoRGB (float hue, float saturation, float brightness);
  public static float[] RGBtoHSB (int red, int green, int blue,
      float hsbvalues[]);

  // Instance Methods
  public Color brighter();
  public Color darker();
  public boolean equals (Object object);
  public int getBlue();
  public int getGreen();
  public int getRed();
  public int getRGB();
  public int hashCode();
  public String toString();
}
```

Constants

black

public static final Color black

The color black.

blue

public static final Color blue

The color blue.

cyan

public static final Color cyan

The color cyan.

darkGray

public static final Color darkGray

The color dark gray.

gray

public static final Color gray

The color gray.

green

public static final Color green

The color green.

lightGray

public static final Color lightGray

The color light gray.

magenta

public static final Color magenta

The color magenta.

orange

public static final Color orange

The color orange.

pink

public static final Color pink

The color pink.

red

public static final Color red

The color red.

white

public static final Color white

The color white.

yellow

public static final Color yellow

The color yellow.

Constructors

Color

public Color (int rgb)

Parameters

rgb

Composite color value

Description

Constructs a `Color` object with the given `rgb` value.

public Color (int red, int green, int blue)

Parameters

red

Red component of color in the range[0, 255]

green

Green component of color in the range[0, 255]

blue

Blue component of color in the range[0, 255]

Description

Constructs a `Color` object with the given `red`, `green`, and `blue` values.

public Color (float red, float green, float blue)

Parameters

red

Red component of color in the range[0.0, 1.0]

green

Green component of color in the range[0.0, 1.0]

blue

Blue component of color in the range[0.0, 1.0]

Description

Constructs a `Color` object with the given `red`, `green`, and `blue` values.

Class Methods

decode

public static Color decode (String nm) ★

Parameters

nm

A `String` representing a color as a 24-bit integer.

Returns

The color requested.

Throws

`NumberFormatException`

If `nm` cannot be converted to a number.

Description

Gets color specified by the given string.

getColor

public static Color getColor (String name)

Parameters

name

The name of a system property indicating which color to fetch.

Returns

Color instance of `name` requested, or `null` if the `name` is invalid.

Description

Gets color specified by the system property name.

public static Color getColor (String name, Color defaultColor)

Parameters

name

The `name` of a system property indicating which color to fetch.

defaultColor

Color to return if `name` is not found in properties, or invalid.

Returns

Color instance of `name` requested, or `defaultColor` if the `name` is invalid.

Description

Gets color specified by the system property `name`.

public static Color getColor (String name, int defaultColor)

Parameters

name

The `name` of a system property indicating which color to fetch.

defaultColor

Color to return if `name` is not found in properties, or invalid.

Returns

Color instance of `name` requested, or `defaultColor` if the `name` is invalid.

Description

Gets color specified by the system property `name`. The default color is specified as a 32-bit RGB value.

getHSBColor

public static Color getHSBColor (float hue, float saturation, float brightness)

Parameters

hue

Hue component of `Color` to create, in the range[0.0, 1.0].

saturation

Saturation component of `Color` to create, in the range[0.0, 1.0].

brightness

Brightness component of `Color` to create, in the range[0.0, 1.0].

Returns

Color instance for values provided.

Description

Create an instance of `Color` by using hue, saturation, and brightness instead of red, green, and blue values.

HSBtoRGB

public static int HSBtoRGB (float hue, float saturation, float brightness)

Parameters

hue

Hue component of `Color` to convert, in the range[0.0, 1.0].

saturation

Saturation component of `Color` to convert, in the range[0.0, 1.0].

brightness

Brightness component of `Color` to convert, in the range[0.0, 1.0].

Returns

Color value for hue, saturation, and brightness provided.

Description

Converts a specific hue, saturation, and brightness to a `Color` and returns the red, green, and blue values in a composite integer value.

RGBtoHSB

public static float[] RGBtoHSB (int red, int green, int blue, float[] hsbvalues)

Parameters

red

Red component of `Color` to convert, in the range[0, 255].

green

Green component of `Color` to convert, in the range[0, 255].

blue

Blue component of `Color` to convert, in the range[0, 255].

hsbvalues

Three element array in which to put the result. This array is used as the method's return object. If `null`, a new array is allocated.

Returns

Hue, saturation, and brightness values for `Color` provided, in elements 0, 1, and 2 (respectively) of the returned array.

Description

Allows you to convert specific red, green, blue value to the hue, saturation, and brightness equivalent.

Instance Methods

brighter

public Color brighter()

Returns

Brighter version of current color.

Description

Creates new `Color` that is somewhat brighter than current.

darker

public Color darker()

Returns

Darker version of current color.

Description

Creates new `Color` that is somewhat darker than current.

equals

public boolean equals (Object object)

Parameters

object

The object to compare.

Returns

`true` if `object` represents the same color, `false` otherwise.

Overrides

`Object.equals(Object)`

Description

Compares two different `Color` instances for equivalence.

getBlue

public int getBlue()

Returns

Blue component of current color.

getGreen

public int getGreen()

Returns

Green component of current color.

getRed

public int getRed()

Returns

Red component of current color.

getRGB

public int getRGB()

Returns

Current color as a composite value.

Description

Gets integer value of current color.

hashCode

public int hashCode()

Returns

A hashcode to use when storing `Color` in a `Hashtable`.

Overrides

`Object.hashCode()`

Description

Generates a hashcode for the `Color`.

toString

public String toString()

Returns

> A string representation of the `Color` object.

Overrides

> `Object.toString()`

See Also

`Object`, `Properties`, `Serializable`, `String`

Component

Name

Component

[Graphic: Figure from the text]

Description

The `Component` class is the parent of all non-menu GUI components.

Class Definition

```
public abstract class java.awt.Component
    extends java.lang.Object
    implements java.awt.image.ImageObserver
    implements java.awt.MenuContainer
    implements java.io.Serializable {
  // Constants
  public final static float BOTTOM_ALIGNMENT; ★
  public final static float CENTER_ALIGNMENT; ★
  public final static float LEFT_ALIGNMENT; ★
  public final static float RIGHT_ALIGNMENT; ★
  public final static float TOP_ALIGNMENT; ★
  // Variables
  protected Locale locale; ★
  // Constructors
  protected Component(); ★

  // Instance Methods
  public boolean action (Event e, Object o); ☆
  public synchronized void add (PopupMenu popup); ★
  public synchronized void addComponentListener
      (ComponentListener l); ★
  public synchronized void addFocusListener (FocusListener l); ★
  public synchronized void addKeyListener (KeyListener l); ★
  public synchronized void addMouseListener (MouseListener l); ★
  public synchronized void addMouseMotionListener
      (MouseMotionListener l); ★
  public void addNotify();
  public Rectangle bounds(); ☆
  public int checkImage (Image image, ImageObserver observer);
  public int checkImage (Image image, int width, int height,
      ImageObserver observer);
  public boolean contains (int x, int y); ★
  public boolean contains (Point p); ★
  public Image createImage (ImageProducer producer);
  public Image createImage (int width, int height);
  public void deliverEvent (Event e); ☆
  public void disable(); ☆
  public final void dispatchEvent (AWTEvent e) ★
  public void doLayout(); ★
  public void enable(); ☆
  public void enable (boolean condition); ☆
  public float getAlignmentX(); ★
  public float getAlignmentY(); ★
  public Color getBackground();
  public Rectangle getBounds(); ★
  public synchronized ColorModel getColorModel();
  public Component getComponentAt (int x, int y); ★
  public Component getComponentAt (Point p); ★
  public Cursor getCursor(); ★
  public Font getFont();
  public FontMetrics getFontMetrics (Font f);
  public Color getForeground();
  public Graphics getGraphics();
  public Locale getLocale(); ★
  public Point getLocation(); ★
  public Point getLocationOnScreen(); ★
  public Dimension getMaximumSize(); ★
  public Dimension getMinimumSize(); ★
  public String getName(); ★
  public Container getParent();
  public ComponentPeer getPeer(); ☆
  public Dimension getPreferredSize(); ★
  public Dimension getSize(); ★
  public Toolkit getToolkit();
  public final Object getTreeLock(); ★
  public boolean gotFocus (Event e, Object o); ☆
  public boolean handleEvent (Event e); ☆
  public void hide(); ☆
  public boolean imageUpdate (Image image, int infoflags, int x, int y,
      int width, int height);
  public boolean inside (int x, int y); ☆
  public void invalidate();
  public boolean isEnabled();
  public boolean isFocusTraversable(); ★
  public boolean isShowing();
  public boolean isValid();
  public boolean isVisible();
  public boolean keyDown (Event e, int key); ☆
  public boolean keyUp (Event e, int key); ☆
  public void layout(); ☆
  public void list();
  public void list (PrintStream out);
  public void list (PrintStream out, int indentation);
  public void list (PrintWriter out); ★
  public void list (PrintWriter out, int indentation); ★
  public Component locate (int x, int y); ☆
  public Point location(); ☆
  public boolean lostFocus (Event e, Object o); ☆
  public Dimension minimumSize(); ☆
  public boolean mouseDown (Event e, int x, int y); ☆
  public boolean mouseDrag (Event e, int x, int y); ☆
  public boolean mouseEnter (Event e, int x, int y); ☆
  public boolean mouseExit (Event e, int x, int y); ☆
  public boolean mouseMove (Event e, int x, int y); ☆
  public boolean mouseUp (Event e, int x, int y); ☆
  public void move (int x, int y); ☆
  public void nextFocus(); ☆
  public void paint (Graphics g);
  public void paintAll (Graphics g);
  public boolean postEvent (Event e); ☆
  public Dimension preferredSize(); ☆
  public boolean prepareImage (Image image, ImageObserver observer);
  public boolean prepareImage (Image image, int width, int height,
      ImageObserver observer);
  public void print (Graphics g);
  public void printAll (Graphics g);
  public synchronized void remove (MenuComponent popup); ★
  public synchronized void removeComponentListener
      (ComponentListener l); ★
  public synchronized void removeFocusListener (FocusListener l); ★
  public synchronized void removeKeyListener (KeyListener l); ★
  public synchronized void removeMouseListener (MouseListener l); ★
  public synchronized void removeMouseMotionListener
      (MouseMotionListener l); ★
  public void removeNotify();
  public void repaint();
  public void repaint (long tm);
  public void repaint (int x, int y, int width, int height);
  public void repaint (long tm, int x, int y, int width, int height);
  public void requestFocus();
  public void reshape (int x, int y, int width, int height); ☆
  public void resize (Dimension d); ☆
  public void resize (int width, int height); ☆
  public void setBackground (Color c);
  public void setBounds (int x, int y, int width, int height); ★
  public void setBounds (Rectangle r); ★
  public synchronized void setCursor (Cursor cursor); ★
  public void setEnabled (boolean b); ★
  public synchronized void setFont (Font f);
  public void setForeground (Color c);
  public void setLocale (Locale l); ★
  public void setLocation (int x, int y); ★
  public void setLocation (Point p); ★
  public void setName (String name); ★
  public void setSize (int width, int height); ★
  public void setSize (Dimension d); ★
  public void setVisible (boolean b); ★
  public void show(); ☆
  public void show (boolean condition); ☆
  public Dimension size(); ☆
  public String toString();
  public void transferFocus(); ★
  public void update (Graphics g);
  public void validate();

  // Protected Instance Methods
  protected final void disableEvents (long eventsToDisable); ★
  protected final void enableEvents (long eventsToEnable); ★
  protected String paramString();
```

```
  protected void processComponentEvent (ComponentEvent e); ★
  protected void processEvent (AWTEvent e); ★
  protected void processFocusEvent (FocusEvent e); ★
  protected void processKeyEvent (KeyEvent e); ★
  protected void processMouseEvent (MouseEvent e); ★
  protected void processMouseMotionEvent (MouseEvent e); ★
}
```

Constants

BOTTOM_ALIGNMENT

public final static float BOTTOM_ALIGNMENT ★

Constant representing bottom alignment in `getAlignmentY()`.

CENTER_ALIGNMENT

public final static float CENTER_ALIGNMENT ★

Constant representing center alignment in `getAlignmentX()` and `getAlignmentY()`.

LEFT_ALIGNMENT

public final static float LEFT_ALIGNMENT ★

Constant representing left alignment in `getAlignmentX()`.

RIGHT_ALIGNMENT

public final static float RIGHT_ALIGNMENT ★

Constant representing right alignment in `getAlignmentX()`.

TOP_ALIGNMENT

public final static float TOP_ALIGNMENT ★

Constant representing top alignment in `getAlignmentY()`.

Variables

locale

protected Locale locale ★

Description

> The locale for the component. Used for internationalization support.

Constructors

Component

protected Component() ★

Description

> This constructor creates a "lightweight" component. This constructor allows `Component` to be directly subclassed using code written entirely in Java.

Instance Methods

action

public boolean action (Event e, Object o) ☆

Parameters

> *e*
>
> > `Event` instance identifying what triggered the call to this method.
>
> *o*

Argument specific to the component subclass that generated the event.

Returns

`true` if event handled, `false` to propagate it to parent container.

Description

Method called when user performs some action in `Component`. This method is a relic of the old 1.0.2 event model and is replaced by the `process...Event()` methods.

add

public synchronized void add (PopupMenu popup) ★

Parameters

popup

The menu to add.

Description

After the `PopupMenu` is added to a component, it can be shown in the component's coordinate space.

addComponentListener

public void addComponentListener (ComponentListener l) ★

Description

Adds a listener for the `ComponentEvent` objects this `Component` generates.

addFocusListener

public void addFocusListener (FocusListener l) ★

Description

Adds a listener for the `FocusEvent` objects this `Component` generates.

addKeyListener

public void addKeyListener (KeyListener l) ★

Description

Adds a listener for the `KeyEvent` objects this `Component` generates.

addMouseListener

public void addMouseListener (MouseListener l) ★

Description

Adds a listener for the `MouseEvent` objects this `Component` generates.

addMouseMotionListener

public void addMouseMotionListener (MouseMotionListener l) ★

Description

Adds a listener for the motion `MouseEvent` objects this `Component` generates.

addNotify

public void addNotify()

Description

Creates peer of `Component`'s subclass.

bounds

public Rectangle bounds() ☆

Returns

Gets bounding rectangle of `Component`.

Description

A `Rectangle` that returns the outer limits of the `Component`. Replaced by `getBounds()` in 1.1.

checkImage

public int checkImage (Image image, ImageObserver observer)

Parameters

image

`Image` to check.

observer

The object an image will be rendered onto.

Returns

`ImageObserver` Flags ORed together indicating the image's status.

Description

Checks status of image construction.

public int checkImage (Image image, int width, int height, ImageObserver observer)

Parameters

image

Image to check.

width

Horizontal size image will be scaled to.

height

Vertical size image will be scaled to.

observer

Object image will be rendered onto.

Returns

`ImageObserver` flags ORed together indicating the image's status.

Description

Checks status of image construction.

contains

public boolean contains (int x, int y) ★

Parameters

x

The x coordinate, in this `Component`'s coordinate system.

y

The y coordinate, in this `Component`'s coordinate system.

Returns

`true` if the `Component` contains the point; `false` otherwise.

public boolean contains (Point p) ★

Parameters

p

The point to be tested, in this `Component`'s coordinate system.

Returns

`true` if the `Component` contains the point; `false` otherwise.

createImage

public Image createImage (ImageProducer producer)

Parameters

producer

Class that implements `ImageProducer` interface to create the new image.

Returns

Newly created image instance.

Description

Creates an `Image` based upon an `ImageProducer`.

public Image createImage (int width, int height)

Parameters

width

Horizontal size for in-memory `Image`.

height

Vertical size for in-memory `Image`.

Returns

Newly created image instance.

Description

Creates an empty in-memory `Image` for double buffering; to draw on the image, use its graphics context.

deliverEvent

public void deliverEvent (Event e) ☆

Parameters

e

`Event` instance to deliver.

Description

Delivers event to the component for processing.

disable

public void disable() ☆

Description

Disables component so that it is unresponsive to user interactions. Replaced by `setEnabled(false)`.

dispatchEvent

public final void dispatchEvent (AWTEvent e) ★

Parameters

e

The `AWTEvent` to process.

Description

Tells the component to deal with the `AWTEvent e`.

doLayout

public void doLayout() ★

Description

Lays out component. This method is a replacement for `layout()`.

enable

public void enable() ☆

Description

Enables component so that it is responsive to user interactions. Use `setEnabled(true)` instead.

public void enable (boolean condition) ☆

Parameters

condition

`true` to enable the component; `false` to disable it.

Description

Enables or disables the component based upon `condition`. Use `setEnabled(boolean)` instead.

getAlignmentX

public float getAlignmentX() ★

Returns

A number between 0 and 1 representing the horizontal alignment of this component.

Description

One of the constants `LEFT_ALIGNMENT`, `CENTER_ALIGNMENT`, or `RIGHT_ALIGNMENT` may be returned. `CENTER_ALIGNMENT` is returned by default.

getAlignmentY

public float getAlignmentY() ★

Returns

A number between 0 and 1 representing the vertical alignment of this component.

Description

One of the constants `TOP_ALIGNMENT`, `CENTER_ALIGNMENT`, or `BOTTOM_ALIGNMENT` may be returned. `CENTER_ALIGNMENT` is returned by default.

getBackground

public Color getBackground()

Returns

Background color of the component.

getBounds

public Rectangle getBounds() ★

Returns

Gets bounding rectangle of `Component`.

Description

Returns a `Rectangle` that returns the outer limits of the `Component`.

getColorModel

public synchronized ColorModel getColorModel()

Returns

`ColorModel` used to display the current component.

getComponentAt

public Component getComponentAt (int x, int y) ★

Parameters

x

The x coordinate, in this `Component`'s coordinate system.

y

The y coordinate, in this `Component`'s coordinate system.

Returns

Returns the `Component` containing the given point.

public Component getComponentAt (Point p) ★

Parameters

p

The point to be tested, in this `Component`'s coordinate system.

Returns

Returns the `Component` containing the given point.

getCursor

public Cursor getCursor() ★

Returns

Current cursor of the component.

getFont

public Font getFont()

Returns

Current font of the component.

getFontMetrics

public FontMetrics getFontMetrics (Font f)

Parameters

f

A `Font` object, whose platform specific information is desired.

Returns

Size information for the given `Font`.

getForeground

public Color getForeground()

Returns

Foreground color of component.

getGraphics

public Graphics getGraphics()

Throws

`InternalException`

If acquiring graphics context is unsupported.

Returns

`Component`'s graphics context.

getLocale

public Locale getLocale() ★

Throws

`IllegalComponentStateException`

If the component does not have a locale or it has not been added to a hierarchy that does.

Returns

`Component`'s locale.

getLocation

public Point getLocation() ★

Returns

Position of component.

Description

Gets the current position of this `Component` in its parent's coordinate space.

getLocationOnScreen

public Point getLocationOnScreen() ★

Returns

Position of component.

Description

Gets the current position of this `Component` in the screen's coordinate space.

getMaximumSize

public Dimension getMaximumSize() ★

Returns

The maximum dimensions of the component.

Description

By default, a maximal `Dimension` is returned.

getMinimumSize

public Dimension getMinimumSize() ★

Returns

The minimum dimensions of the component.

getName

public String getName() ★

Returns

This component's name.

getParent

public Container getParent()

Returns

> Parent `Container` of `Component`.

Description

> Gets container that this `Component` is held in.

getPeer

public ComponentPeer getPeer() ☆

Returns

> Peer of `Component`.

getPreferredSize

public Dimension getPreferredSize() ★

Returns

> The preferred dimensions of the component.

getSize

public Dimension getSize() ★

Returns

> Dimensions of component.

Description

> Gets width and height of component.

getToolkit

public Toolkit getToolkit()

Returns

`Toolkit` of `Component`.

getTreeLock

public final Object getTreeLock() ★

Returns

The AWT tree locking object.

Description

Returns the object used for tree locking and layout operations.

gotFocus

public boolean gotFocus (Event e, Object o) ☆

Parameters

e

`Event` instance identifying what triggered the call to this method.

o

Argument specific to the component subclass that generated the event.

Returns

`true` if event handled, `false` to propagate it to parent container.

Description

Called when `Component` gets input focus. This method is not used in the 1.1 event model.

handleEvent

public boolean handleEvent (Event e) ☆

Parameters

e

`Event` instance identifying what triggered the call to this method.

Returns

`true` if event handled, `false` to propagate it to parent container.

Description

High-level event handling routine that calls helper routines. Replaced by `processEvent(AWTEvent)`.

hide

public void hide() ☆

Description

Hides component from view. Replaced by `setVisible(false)`.

imageUpdate

public boolean imageUpdate (Image image, int infoflags, int x, int y, int width, int height)

Parameters

image

`Image` being loaded.

infoflags

`ImageObserver` flags ORed together of available information.

x

x coordinate of upper-left corner of `Image`.

y

y coordinate of upper-left corner of `Image`.

width

Horizontal dimension of `Image`.

height

Vertical dimension of `Image`.

Returns

`true` if `Image` fully loaded, `false` otherwise.

Implements

`ImageObserver.imageUpdate()`

Description

An asynchronous update interface for receiving notifications about `Image` information as it is loaded. Meaning of parameters changes with values of flags.

inside

public boolean inside (int x, int y) ☆

Parameters

x

Horizontal position.

y

Vertical position.

Returns

`true` if the point (`x`, `y`) falls within the component's bounds, `false` otherwise.

Description

Checks if coordinates are within bounding box of `Component`. Replaced by `contains(int, int)`.

invalidate

public void invalidate()

Description

Sets the component's valid state to `false`.

isEnabled

public boolean isEnabled()

Returns

`true` if enabled, `false` otherwise.

Description

Checks to see if the `Component` is currently enabled.

isFocusTraversable

public boolean isFocusTraversable() ★

Returns

`true` if this `Component` can be traversed using Tab and Shift-Tab, `false` otherwise.

Description

Checks to see if the `Component` is navigable using the keyboard.

isShowing

public boolean isShowing()

Returns

`true` if showing, `false` otherwise.

Description

Checks to see if the `Component` is currently showing.

isValid

public boolean isValid()

Returns

`true` if valid, `false` otherwise.

Description

Checks to see if the `Component` is currently valid.

isVisible

public boolean isVisible()

Returns

`true` if visible, `false` otherwise.

Description

Checks to see if the `Component` is currently visible.

keyDown

public boolean keyDown (Event e, int key) ☆

Parameters

e

`Event` instance identifying what triggered the call to this method.

key

Integer representation of key pressed.

Returns

`true` if event handled, `false` to propagate it to parent container.

Description

Method called whenever the user presses a key. Replaced by `processKeyEvent(KeyEvent)`.

keyUp

public boolean keyUp (Event e, int key) ☆

Parameters

e

`Event` instance identifying what triggered the call to this method.

key

Integer representation of key released.

Returns

`true` if event handled, `false` to propagate it to parent container.

Description

Method called whenever the user releases a key. Replaced by `processKeyEvent(KeyEvent)`.

layout

public void layout() ☆

Description

Lays out component. Replaced by `doLayout()`.

list

public void list()

Description

Prints the contents of the `Component` to `System.out`.

public void list (PrintStream out)

Parameters

out

Output stream to send results to.

Description

Prints the contents of the `Component` to a `PrintStream`.

public void list (PrintStream out, int indentation)

Parameters

out

Output stream to send results to.

indentation

Indentation to use when printing.

Description

Prints the contents of the `Component` indented to a `PrintStream`.

public void list (PrintWriter out)

Parameters

out

Output stream to send results to.

Description

Prints the contents of the `Component` to a `PrintWriter`.

public void list (PrintWriter out, int indentation)

Parameters

out

Output stream to send results to.

indentation

Indentation to use when printing.

Description

Prints the contents of the `Component` indented to a `PrintWriter`.

locate

public Component locate (int x, int y) ☆

Parameters

x

Horizontal position.

y

Vertical position.

Returns

`Component` if the point (`x`, `y`) falls within the component, `null` otherwise.

Description

Replaced by `getComponentAt(int, int)`.

location

public Point location() ☆

Returns

Position of component.

Description

Gets the current position of this `Component` in its parent's coordinate space. Replaced by `getLocation()`.

lostFocus

public boolean lostFocus (Event e, Object o) ☆

Parameters

e

`Event` instance identifying what triggered the call to this method.

o

Argument specific to the component subclass that generated the event.

Returns

`true` if event handled, `false` to propagate it to parent container.

Description

Method called when `Component` loses input focus. Replaced by `processFocusEvent(FocusEvent)`.

minimizeSize

public Dimension minimumSize() ☆

Returns

The minimum dimensions of the component. Replaced by `getMinimumSize()`.

mouseDown

public boolean mouseDown (Event e, int x, int y) ☆

Parameters

e

`Event` instance identifying what triggered the call to this method.

x

Horizontal position of the mouse within `Component` when `Event` initiated

y

Vertical position of the mouse within `Component` when `Event` initiated

Returns

`true` if event handled, `false` to propagate it to parent container.

Description

Method called when the user presses a mouse button over `Component`. Replaced by `processMouseEvent(MouseEvent)`.

mouseDrag

public boolean mouseDrag (Event e, int x, int y) ☆

Parameters

e

`Event` instance identifying what triggered the call to this method.

x

Horizontal position of the mouse within `Component` when `Event` initiated

y

Vertical position of the mouse within `Component` when `Event` initiated

Returns

`true` if event handled, `false` to propagate it to parent container.

Description

Method called when the user is pressing a mouse button and moves the mouse. Replaced by `processMouseMotionEvent(MouseEvent)`.

mouseEnter

public boolean mouseEnter (Event e, int x, int y) ☆

Parameters

e

`Event` instance identifying what triggered the call to this method.

x

Horizontal position of the mouse within `Component` when `Event` initiated

y

Vertical position of the mouse within `Component` when `Event` initiated

Returns

`true` if event handled, `false` to propagate it to parent container.

Description

Method called when the mouse enters `Component`. Replaced by `processMouseEvent(MouseEvent)`.

mouseExit

public boolean mouseExit (Event e, int x, int y) ☆

Parameters

e

`Event` instance identifying what triggered the call to this method.

x

Horizontal position of the mouse within `Component` when `Event` initiated

y

Vertical position of the mouse within `Component` when `Event` initiated

Returns

`true` if event handled, `false` to propagate it to parent container.

Description

Method called when the mouse exits `Component`. Replaced by `processMouseEvent(MouseEvent)`.

mouseMove

public boolean mouseMove (Event e, int x, int y) ☆

Parameters

e

`Event` instance identifying what triggered the call to this method.

x

Horizontal position of the mouse within `Component` when `Event` initiated

y

Vertical position of the mouse within `Component` when `Event` initiated

Returns

`true` if event handled, `false` to propagate it to parent container.

Description

Method called when the user is not pressing a mouse button and moves the mouse. Replaced by `processMouseMotionEvent(MouseEvent)`.

mouseUp

public boolean mouseUp (Event e, int x, int y) ☆

Parameters

e

`Event` instance identifying what triggered the call to this method.

x

Horizontal position of the mouse within `Component` when `Event` initiated

y

Vertical position of the mouse within `Component` when `Event` initiated

Returns

`true` if event is handled, `false` to propagate it to the parent container.

Description

Method called when user releases mouse button over `Component`. Replaced by `processMouseEvent(MouseEvent)`.

move

public void move (int x, int y) ☆

Parameters

x

New horizontal position for component.

y

New vertical position for component.

Description

> Relocates component. Replaced by `setLocation(int, int)`.

nextFocus

public void nextFocus() ☆

Description

> Moves focus from current component to next one in parent container. Replaced by `transferFocus()`.

paint

public void paint (Graphics g)

Parameters

> *g*
>
> > Graphics context of component.

Description

> Empty method to be overridden to draw something in the graphics context.

paintAll

public void paintAll (Graphics g)

Parameters

> *g*
>
> > Graphics context of component.

Description

> Method to validate component and paint its peer if it is visible.

postEvent

public boolean postEvent (Event e) ☆

Parameters

e

`Event` instance to post to component

Returns

If `Event` is handled, `true` is returned. Otherwise, `false` is returned.

Description

Tells `Component` to deal with `Event`.

preferredSize

public Dimension preferredSize() ☆

Returns

The preferred dimensions of the component. Replaced by `getPreferredSize()`.

prepareImage

public boolean prepareImage (Image image, ImageObserver observer)

Parameters

image

`Image` to start loading.

observer

Component on which image will be rendered.

Returns

`true` if `Image` is fully loaded, `false` otherwise.

Description

Forces `Image` to start loading.

public boolean prepareImage (Image image, int width, int height, ImageObserver observer)

Parameters

image

`Image` to start loading.

width

Horizontal size of the `Image` after scaling.

height

Vertical size of the `Image` after scaling.

observer

Component on which image will be rendered.

Returns

`true` if `Image` is fully loaded, `false` otherwise.

Description

Forces `Image` to start loading.

print

public void print (Graphics g)

Parameters

g

Graphics context.

Description

Empty method to be overridden to print something into the graphics context.

printAll

public void printAll (Graphics g)

Parameters

g

Graphics context.

Description

Method to print this component and its children.

remove

public void remove (MenuComponent popup) ★

Parameters

popup

The menu to remove.

Description

After adding a `PopupMenu`, you can use this method to remove it.

removeComponentListener

public void removeComponentListener (ComponentListener l) ★

Description

> Removes the specified `ComponentListener` from this `Component`.

removeFocusListener

public void removeFocusListener (FocusListener l) ★

Description

> Removes the specified `FocusListener` from this `Component`.

removeKeyListener

public void removeKeyListener (KeyListener l) ★

Description

> Removes the specified `KeyListener` from this `Component`.

removeMouseListener

public void removeMouseListener (MouseListener l) ★

Description

> Removes the specified `MouseListener` from this `Component`.

removeMouseMotionListener

public void removeMouseMotionListener (MouseMotionListener l) ★

Description

> Removes the specified `MouseMotionListener` from this `Component`.

removeNotify

public void removeNotify()

Description

Removes peer of `Component`'s subclass.

repaint

public void repaint()

Description

Requests scheduler to redraw the component as soon as possible.

public void repaint (long tm)

Parameters

tm

Millisecond delay allowed before repaint.

Description

Requests scheduler to redraw the component within a time period.

public void repaint (int x, int y, int width, int height)

Parameters

x

Horizontal origin of bounding box to redraw.

y

Vertical origin of bounding box to redraw.

width

Width of bounding box to redraw.

height

Height of bounding box to redraw.

Description

Requests scheduler to redraw a portion of component as soon as possible.

public void repaint (long tm, int x, int y, int width, int height)

Parameters

tm

Millisecond delay allowed before repaint.

x

Horizontal origin of bounding box to redraw.

y

Vertical origin of bounding box to redraw.

width

Width of bounding box to redraw.

height

Height of bounding box to redraw.

Description

Requests scheduler to redraw a portion of component within a time period.

requestFocus

public void requestFocus()

Description

Requests the input focus for this `Component`.

reshape

public void reshape (int x, int y, int width, int height) ☆

Parameters

x

New horizontal position for component.

y

New vertical position for component.

width

New width for component.

height

New height for component.

Description

Relocates and resizes component. Replaced by `setBounds(int, int, int, int)`.

resize

public void resize (Dimension d) ☆

Parameters

d

New dimensions for the component.

Description

Resizes component. Replaced by `setSize(Dimension)`.

public void resize (int width, int height) ☆

Parameters

width

New width for component.

height

New height for component.

Description

Resizes component. Replaced by `setSize(int, int)`.

setBackground

public void setBackground (Color c)

Parameters

c

New background color.

Description

Changes the component's background color.

setBounds

public void setBounds (int x, int y, int width, int height) ★

Parameters

x

New horizontal position for component.

y

New vertical position for component.

width

New width for component.

height

New height for component.

Description

Relocates and resizes the component.

public void setBounds (Rectangle r) ★

Parameters

r

New coordinates for component.

Description

Relocates and resizes component.

setCursor

public synchronized void setCursor (Cursor cursor) ★

Parameters

cursor

The new cursor for the component.

Description

Changes the component's cursor.

setEnabled

public void setEnabled (boolean b) ★

Parameters

b

`true` to enable the component, `false` to disable it.

Description

Enables or disables the component. Replaces `enable()`, `enable(boolean)`, and `disable()`.

setFont

public synchronized void setFont (Font f)

Parameters

f

Font to change component to.

Description

Changes the font of the component.

setForeground

public void setForeground (Color c)

Parameters

c

New foreground color.

Description

Changes the foreground color of component's area.

setLocale

public void setLocale (Locale l) ★

Parameters

l

The locale object for the component.

Description

Sets the `Component`'s locale.

setLocation

public void setLocation (int x, int y) ★

Parameters

x

New horizontal position for component.

y

New vertical position for component.

Description

Relocates the component.

public void setLocation (Point p) ★

Parameters

p

New position for component.

Description

Relocates the component.

setName

public void setName (String name) ★

Parameters

name

New name for component.

Description

Sets the component's name.

setSize

public void setSize (int width, int height) ★

Parameters

width

New width for component.

height

New height for component.

Description

Resizes the component.

public void setSize (Dimension d) ★

Parameters

d

New dimensions for the component.

Description

Resizes the component.

setVisible

public void setVisible (boolean b) ★

Parameters

b

`true` to show component, `false` to hide it.

Description

Shows or hides the component based on the `b` parameter.

show

public void show() ☆

Description

Replaced by `setVisible(true)`.

public void show (boolean condition) ☆

Parameters

condition

`true` to show the component, `false` to hide it.

Description

Replaced by `setVisible(boolean)`.

size

public Dimension size() ☆

Returns

Dimensions of the component.

Description

Gets width and height of the component. Replaced by `getSize()`.

toString

public String toString()

Returns

A string representation of the `Component` object.

Overrides

`Object.toString()`

transferFocus

public void transferFocus() ★

Description

Transfers focus to the next component in the container hierarchy.

update

public void update (Graphics g)

Parameters

g

Graphics context of component.

Description

Called to update the component's display area.

validate

public void validate()

Description

Sets the component's valid state to `true`.

Protected Instance Methods

disableEvents

protected final void disableEvents (long eventsToDisable) ★

Parameters

eventsToDisable

A value representing certain kinds of events. This can be constructed by ORing the event mask constants defined in `java.awt.AWTEvent`.

Description

By default, a component receives events corresponding to the event listeners that have registered. If a component should not receive events of a certain type, even if there is a listener registered for that type of event, this method can be used to disable that event type.

enableEvents

protected final void enableEvents (long eventsToEnable) ★

Parameters

eventsToEnable

A value representing certain kinds of events. This can be constructed by ORing the event mask constants defined in `java.awt.AWTEvent`.

Description

By default, a component receives events corresponding to the event listeners that have registered. If a component should receive other types of events as well, this method can be used to request them.

paramString

protected String paramString()

Returns

A `String` with the current settings of the `Component`.

Description

Helper method for `toString()` to generate a string of current settings.

processComponentEvent

protected void processComponentEvent(ComponentEvent e) ★

Parameters

e

The event to process.

Description

Component events are passed to this method for processing. Normally, this method is called by `processEvent()`.

processEvent

protected void processEvent(AWTEvent e) ★

Parameters

e

The event to process.

Description

Low level `AWTEvent`s are passed to this method for processing.

processFocusEvent

protected void processFocusEvent(FocusEvent e) ★

Parameters

e

The event to process.

Description

Focus events are passed to this method for processing. Normally, this method is called by `processEvent()`.

processKeyEvent

protected void processKeyEvent(KeyEvent e) ★

Parameters

e

The event to process.

Description

Key events are passed to this method for processing. Normally, this method is called by `processEvent()`.

processMouseEvent

protected void processMouseEvent(MouseEvent e) ★

Parameters

e

The event to process.

Description

Mouse events are passed to this method for processing. Normally, this method is called by `processEvent()`.

processMouseMotionEvent

protected void processMouseMotionEvent(MouseEvent e) ★

Parameters

e

The event to process.

Description

> Mouse motion events are passed to this method for processing. Normally, this method is called by `processEvent()`.

See Also

`Button`, `Canvas`, `Checkbox`, `Choice`, `Color`, `ColorModel`, `ComponentPeer`, `Container`, `Dimension`, `Event`, `Font`, `FontMetrics`, `Graphics`, `ImageObserver`, `ImageProducer`, `Label`, `List`, `MenuContainer`, `Object`, `Point`, `PrintStream`, `Rectangle`, `Scrollbar`, `Serializable`, `String`, `TextComponent`, `Toolkit`

Container

Name

Container

[Graphic: Figure from the text]

Description

The `Container` class serves as a general purpose holder of other `Component` objects.

Class Definition

```
public abstract class java.awt.Container
    extends java.awt.Component {
  // Constructors
  protected Container(); ★

  // Instance Methods
  public Component add (Component component);
  public Component add (Component component, int position);
  public void add (Component comp, Object constraints); ★
  public void add (Component comp, Object constraints,
      int position); ★
  public Component add (String name, Component component); ☆
  public void addContainerListener (ContainerListener l); ★
  public void addNotify();
  public int countComponents();
  public void deliverEvent (Event e); ★
  public void doLayout(); ★
  public float getAlignmentX(); ★
  public float getAlignmentY(); ★
  public Component getComponent (int n);
  public Component getComponentAt (int x, int y);  ★
  public Component getComponentAt (Point p);  ★
  public int getComponentCount(); ★
  public Component[] getComponents();
  public Insets getInsets(); ★
  public LayoutManager getLayout();
  public Dimension getMaximumSize(); ★
  public Dimension getMinimumSize();  ★
  public Dimension getPreferredSize();  ★
  public Insets insets();
  public void invalidate(); ★
  public boolean isAncestorOf (Component c); ★
  public void layout(); ☆
  public void list (PrintStream out, int indentation);
  public void list (PrintWriter out, int indentation); ★
  public Component locate (int x, int y); ☆
  public Dimension minimumSize(); ☆
  public void paint (Graphics g); ★
  public void paintComponents (Graphics g);
  public Dimension preferredSize(); ☆
  public void print (Graphics g); ★
  public void printComponents (Graphics g);
  public void remove (int index); ★
  public void remove (Component component);
```

```
  public void removeAll();
  public void removeContainerListener (ContainerListener l); ★
  public void removeNotify();
  public void setLayout (LayoutManager manager);
  public void validate();
  // Protected Instance Methods
  protected void addImpl (Component comp, Object constraints,
      int index); ★
  protected String paramString();
  protected void processContainerEvent (ContainerEvent e); ★
  protected void processEvent (AWTEvent e); ★
  protected void validateTree(); ★
}
```

Constructors

Container

protected Container() ★

Description

> This constructor creates a "lightweight" container. This constructor allows `Container` to be subclassed using code written entirely in Java.

Instance Methods

add

public Component add (Component component)

Parameters

> *component*
>
> > `Component` to add to container.

Returns

`Component` just added.

Throws

`IllegalArgumentException` if you add `component` to itself.

Description

Adds `component` as the last component in the container.

public Component add (Component component, int position)

Parameters

component

`Component` to add to container.

position

Position of component; -1 adds the component as the last in the container.

Returns

`Component just added.`

Throws

`ArrayIndexOutOfBoundsException`

If `position` invalid.

`IllegalArgumentException`

If you add `Component` to itself.

Description

Adds `component` to container at a certain position.

public void add (Component component, Object constraints) ★

Parameters

component

`Component` to add to container.

constraints

An object describing constraints on the component being added.

Description

Adds `component` to container subject to `contraints`.

public void add (Component component, Object constraints, int index) ★

Parameters

component

`Component` to add to container.

constraints

An object describing constraints on the component being added.

index

The position of the component in the container's list.

Description

Adds `component` to container subject to `contraints` at position `index`.

public Component add (String name, Component component) ☆

Parameters

name

Name of component being added. This parameter is often significant to the layout manager of the container (e.g "North", "Center").

component

`Component` to add to container.

Returns

`Component` just added.

Throws

IllegalArgumentException

If you add `component` to itself.

Description

Adds the component to the container with the given `name`. Replaced by the more general `add(Component, Object)`.

addContainerListener

public void addContainerListener (ContainerListener l) ★

Parameters

l

An object that implements the `ContainerListener` interface.

Description

Add a listener for the container events.

addNotify

public void addNotify()

Overrides

`Component.addNotify()`

Description

Creates `Container`'s peer and peers of contained components.

countComponents

public int countComponents()

Returns

Number of components within `Container`.

deliverEvent

public void deliverEvent (Event e) ☆

Parameters

e

`Event` instance to deliver.

Overrides

`Component.deliverEvent(Event)`

Description

Tries to locate the component contained in the container that should receive the event.

doLayout

public void doLayout() ★

Description

Lays out the container. This method is a replacement for `layout()`.

getAlignmentX

public float getAlignmentX() ★

Returns

A number between 0 and 1 representing the horizontal alignment of this component.

Overrides

`Component.getAlignmentX()`

Description

If the container's layout manager implements `LayoutManager2`, this method returns the `getLayoutAlignmentX()` value of the layout manager. Otherwise the `getAlignmentX()` value of `Component` is returned.

getAlignmentY

public float getAlignmentY() ★

Returns

A number between 0 and 1 representing the vertical alignment of this component.

Overrides

`Component.getAlignmentY()`

Description

If the container's layout manager implements `LayoutManager2`, this method returns the `getLayoutAlignmentY()` value of the layout manager. Otherwise the `getAlignmentY()` value of `Component` is returned.

getComponent

public synchronized Component getComponent (int position)

Parameters

position

Position of component to get.

Throws

`ArrayIndexOutOfBoundsException`

If `position` is invalid.

Returns

`Component` at designated `position` within `Container`.

getComponentAt

public Component getComponentAt (int x, int y) ★

Parameters

x

The x coordinate, in this `Container`'s coordinate system.

y

The y coordinate, in this `Container`'s coordinate system.

Returns

Returns the `Component` containing the give point.

public Component getComponentAt (Point p) ★

Parameters

p

The point to be tested, in this `Container`'s coordinate system.

Returns

Returns the `Component` containing the give point.

getComponentCount

public int getComponentCount() ★

Returns

Returns the number of components in the container.

getComponents

public Component[] getComponents()

Returns

Array of components within the container.

getInsets

public Insets getInsets()

Returns

The insets of the container.

getLayout

public LayoutManager getLayout()

Returns

`LayoutManager` of `Container`.

getMaximumSize

public Dimension getMaximumSize() ★

Overrides

`Component.getMaximumSize()`

Returns

The maximum dimensions of the component.

getMinimumSize

public Dimension getMinimumSize() ★

Overrides

`Component.getMinimumSize()`

Returns

The minimum dimensions of the component.

getPreferredSize

public Dimension getPreferredSize() ★

Returns

The preferred dimensions of the component.

insets

public Insets insets() ☆

Returns

Current `Insets` of `Container`. Replaced by `getInsets()`.

invalidate

public void invalidate()

Overrides

`Component.invalidate()`

Description

Sets the container's valid state to `false`.

isAncestorOf

public boolean isAncestorOf (Component c) ★

Parameters

c

The component in question.

Returns

If `c` is contained in the container's hierarchy, returns `true`; otherwise `false`.

layout

public void layout() ☆

Overrides

`Component.layout()`

Description

Replaced by `doLayout()`.

list

public void list (PrintStream out, int indentation)

Parameters

out

Output `Stream` to send results to.

indentation

Indentation to use when printing.

Overrides

`Component.list(PrintStream, int)`

Description

Recursively lists all components in `Container`.

public void list (PrintWriter out, int indentation)

Parameters

out

Output `Writer` to send results to.

indentation

Indentation to use when printing.

Overrides

`Component.list(PrintWriter, int)`

Description

Recursively lists all components in `Container`.

locate

public Component locate (int x, int y) ☆

Parameters

x

Horizontal position to check.

y

Vertical position to check.

Returns

`Component` within `Container` at given coordinates, or `Container`.

Overrides

`Component.locate(int, int)`

Description

Replaced by `getComponentAt(int, int)`.

minimizeSize

public Dimension minimumSize() ☆

Returns

Minimum dimensions of contained objects.

Overrides

`Component.minimumSize()`

Description

Replaced by `getMinimumSize()`.

paint

public void paint (Graphics g)

Parameters

g

Graphics context of container.

Overrides

`Component.paint()`

Description

This method tells any lightweight components that are children of this container to paint themselves.

paintComponents

public void paintComponents (Graphics g)

Parameters

g

Graphics context of `Container`.

Description

Paints the different components in `Container`.

preferredSize

public Dimension preferredSize() ☆

Returns

Preferred dimensions of contained objects.

Overrides

`Component.preferredSize()`

Description

Replaced by `getPreferredSize()`.

print

public void print (Graphics g)

Parameters

g

Graphics context of container.

Overrides

`Component.print()`

Description

This method tells any lightweight components that are children of this container to print themselves.

printComponents

public void printComponents (Graphics g)

Parameters

g

Graphics context of `Container`.

Description

Prints the different components in `Container`.

remove

public void remove (int index) ★

Parameters

index

Index of the component to remove.

Description

Removes the component in position `index` from `Container`.

public void remove (Component component)

Parameters

component

`Component` to remove.

Description

Removes `component` from `Container`.

removeAll

public void removeAll()

Description

Removes all components from `Container`.

removeContainerListener

public void removeContainerListener (ContainerListener l) ★

Parameters

l

One of this `Container`'s `ContainerListener`s.

Description

Remove a container event listener.

removeNotify

public void removeNotify()

Overrides

`Component.removeNotify()`

Description

Removes `Container`'s peer and peers of contained components.

setLayout

public void setLayout (LayoutManager manager)

Parameters

manager

New `LayoutManager` for `Container`.

Description

Changes `LayoutManager` of `Container`.

validate

public void validate()

Overrides

`Component.validate()`

Description

Sets `Container`'s valid state to `true` and recursively validates its children.

Protected Instance Methods

addImpl

protected void addImpl (Component comp, Object constraints, int index) ★

Parameters

comp

The component to add.

constraints

Constraints on the component.

index

Position at which to add this component. Pass -1 to add the component at the end.

Description

This method adds a component subject to the given constraints at a specific position in the container's list of components. It is a helper method for the various overrides of `add()`.

paramString

protected String paramString()

Returns

String with current settings of `Container`.

Overrides

`Component.paramString()`

Description

Helper method for `toString()` to generate string of current settings.

processContainerEvent

protected void processContainerEvent (ContainerEvent e) ★

Parameters

e

The event to process.

Description

Container events are passed to this method for processing. Normally, this method is called by `processEvent()`.

processEvent

protected void processEvent (AWTEvent e) ★

Parameters

e

The event to process.

Overrides

`Component.processEvent()`

Description

Low level `AWTEvent`s are passed to this method for processing.

validateTree

protected void validateTree() ★

Description

Descends recursively into the `Container`'s components and recalculates layout for any subtrees that are marked invalid.

See Also

`Component`, `Dimension`, `Event`, `Graphics`, `Insets`, `LayoutManager`, `Panel`, `PrintStream`, `String`, `Window`

PREVIOUS
Component

HOME
BOOK INDEX

NEXT
Cursor ★

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Cursor ★

Name

Cursor ★

[Graphic: Figure from the text]

Description

The `Cursor` class represents the mouse pointer. It encapsulates information that used to be in `java.awt.Frame` in the 1.0.2 release.

Class Definition

```
public class java.awt.Cursor
    extends java.lang.Object
    implements java.io.Serializable {
  // Constants
  public final static int CROSSHAIR_CURSOR;
  public final static int DEFAULT_CURSOR;
  public final static int E_RESIZE_CURSOR;
  public final static int HAND_CURSOR;
  public final static int MOVE_CURSOR;
  public final static int N_RESIZE_CURSOR;
  public final static int NE_RESIZE_CURSOR;
  public final static int NW_RESIZE_CURSOR;
  public final static int S_RESIZE_CURSOR;
  public final static int SE_RESIZE_CURSOR;
  public final static int SW_RESIZE_CURSOR;
  public final static int TEXT_CURSOR;
  public final static int W_RESIZE_CURSOR;
  public final static int WAIT_CURSOR;
  // Class Variables
  protected static Cursor[] predefined;
  // Class Methods
  public static Cursor getDefaultCursor();
  public static Cursor getPredefinedCursor (int type);
  // Constructors
  public Cursor (int type);
  // Instance Methods
  public int getType();
}
```

Constants

CROSSHAIR_CURSOR

public final static int CROSSHAIR_CURSOR

Constant representing a cursor that looks like a crosshair.

DEFAULT_CURSOR

public final static int DEFAULT_CURSOR

Constant representing the platform's default cursor.

E_RESIZE_CURSOR

public final static int E_RESIZE_CURSOR

Constant representing the cursor for resizing an object on the left.

HAND_CURSOR

public final static int HAND_CURSOR

Constant representing a cursor that looks like a hand.

MOVE_CURSOR

public final static int MOVE_CURSOR

Constant representing a cursor used to move an object.

N_RESIZE_CURSOR

public final static int N_RESIZE_CURSOR

Constant representing a cursor for resizing an object on the top.

NE_RESIZE_CURSOR

public final static int NE_RESIZE_CURSOR

Constant representing a cursor for resizing an object on the top left corner.

NW_RESIZE_CURSOR

public final static int NW_RESIZE_CURSOR

Constant representing a cursor for resizing an object on the top right corner.

S_RESIZE_CURSOR

public final static int S_RESIZE_CURSOR

Constant representing a cursor for resizing an object on the bottom.

SE_RESIZE_CURSOR

public final static int SE_RESIZE_CURSOR

Constant representing a cursor for resizing an object on the bottom left corner.

SW_RESIZE_CURSOR

public final static int SW_RESIZE_CURSOR

Constant representing a cursor for resizing an object on the bottom right corner.

TEXT_CURSOR

public final static int TEXT_CURSOR

Constant representing a cursor used within text.

W_RESIZE_CURSOR

public final static int W_RESIZE_CURSOR

Constant representing a cursor for resizing an object on the right side.

WAIT_CURSOR

public final static int WAIT_CURSOR

Constant representing a cursor that indicates the program is busy.

Class Variables

predefined

protected static Cursor[] predefined

An array of cursor instances corresponding to the predefined cursor types.

Class Methods

getDefaultCursor

public static Cursor getDefaultCursor()

Returns

The default system cursor.

getPredefinedCursor

public static Cursor getPredefinedCursor (int type)

Parameters

type

One of the type constants defined in this class.

Returns

A `Cursor` object with the specified type.

Constructors

Cursor

public Cursor (int type)

Parameters

type

One of the type constants defined in this class.

Description

Constructs a `Cursor` object with the specified type.

Instance Methods

getType

public int getType()

Returns

The type of cursor.

See Also

`Frame`

Graphics

Name

Graphics

[Graphic: Figure from the text]

Description

The `Graphics` class is an abstract class that represents an object on which you can draw. The concrete classes that are actually used to represent graphics objects are platform dependent, but because they extend the `Graphics` class, must implement the methods here.

Class Definition

```
public abstract class java.awt.Graphics
    extends java.lang.Object {

  // Constructors
  protected Graphics();

  // Instance Methods
  public abstract void clearRect (int x, int y, int width, int height);
  public abstract void clipRect (int x, int y, int width, int height);
  public abstract void copyArea (int x, int y, int width, int height,
      int deltax, int deltay);
  public abstract Graphics create();
  public Graphics create (int x, int y, int width, int height);
  public abstract void dispose();
  public void draw3DRect (int x, int y, int width, int height,
      boolean raised);
  public abstract void drawArc (int x, int y, int width, int height,
      int startAngle, int arcAngle);
  public void drawBytes (byte text[], int offset, int length,
      int x, int y);
  public void drawChars (char text[], int offset, int length,
      int x, int y);
  public abstract boolean drawImage (Image image, int x, int y,
      ImageObserver observer);
  public abstract boolean drawImage (Image image, int x, int y,
      int width, int height, ImageObserver observer);
  public abstract boolean drawImage (Image image, int x, int y,
      Color backgroundColor, ImageObserver observer);
  public abstract boolean drawImage (Image image, int x, int y,
      int width, int height, Color backgroundColor, ImageObserver observer);
  public abstract boolean drawImage(Image img, int dx1, int dy1,
      int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, ImageObserver
      observer); ★
  public abstract boolean drawImage(Image img, int dx1, int dy1,
      int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, Color bgcolor,
      ImageObserver observer); ★
  public abstract void drawLine (int x1, int y1, int x2, int y2);
  public abstract void drawOval (int x, int y, int width, int height);
  public abstract void drawPolygon (int xPoints[], int yPoints[],
      int numPoints);
  public void drawPolygon (Polygon p);
  public abstract void drawPolyline(int[ ] xPoints, int[ ] yPoints,
      int nPoints); ★
  public void drawRect (int x, int y, int width, int height);
  public abstract void drawRoundRect (int x, int y, int width,
      int height, int arcWidth, int arcHeight);
  public abstract void drawString (String text, int x, int y);
  public void fill3DRect (int x, int y, int width, int height,
      boolean raised);
  public abstract void fillArc (int x, int y, int width, int height,
      int startAngle, int arcAngle);
  public abstract void fillOval (int x, int y, int width, int height);
  public abstract void fillPolygon (int xPoints[], int yPoints[],
      int numPoints);
  public void fillPolygon (Polygon p);
  public abstract void fillRect (int x, int y, int width, int height);
  public abstract void fillRoundRect (int x, int y, int width,
      int height, int arcWidth, int arcHeight);
  public void finalize();
  public abstract Shape getClip(); ★
```

```
  public abstract Rectangle getClipBounds(); ★
  public abstract Rectangle getClipRect();
  public abstract Color getColor();
  public abstract Font getFont();
  public FontMetrics getFontMetrics();
  public abstract FontMetrics getFontMetrics (Font font);
  public abstract void setClip (int x, int y, int width, int height); ★
  public abstract void setClip (Shape clip); ★
  public abstract void setColor (Color color);
  public abstract void setFont (Font font);
  public abstract void setPaintMode();
  public abstract void setXORMode (Color xorColor);
  public String toString();
  public abstract void translate (int x, int y);
}
```

Constructors

Graphics

protected Graphics()

Description

Called by constructors of platform specific subclasses.

Instance Methods

clearRect

public abstract void clearRect (int x, int y, int width, int height)

Parameters

x

x coordinate of origin of area to clear.

y

y coordinate of origin of area to clear.

width

size in horizontal direction to clear.

height

size in vertical direction to clear.

Description

Resets a rectangular area to the background color.

clipRect

public abstract void clipRect (int x, int y, int width, int height)

Parameters

x

x coordinate of origin of clipped area.

y

y coordinate of origin of clipped area.

width

size in horizontal direction to clip.

height

size in vertical direction to clip.

Description

Reduces the drawing area to the intersection of the current drawing area and the rectangular area defined by `x`, `y`, `width`, and `height`.

copyArea

public abstract void copyArea (int x, int y, int width, int height, int deltax, int deltay)

Parameters

x

x coordinate of origin of area to copy.

y

y coordinate of origin of area to copy.

width

size in horizontal direction to copy.

height

size in vertical direction to copy.

deltax

offset in horizontal direction to copy area to.

deltay

offset in vertical direction to copy area to.

Description

Copies a rectangular area to a new area, whose top left corner is (`x+deltax`, `y+deltay`).

create

public abstract Graphics create()

Returns

New graphics context.

Description

Creates a second reference to the same graphics context.

public Graphics create (int x, int y, int width, int height)

Parameters

x

x coordinate of origin of new graphics context.

y

y coordinate of origin of new graphics context.

width

size in horizontal direction.

height

size in vertical direction.

Returns

New graphics context

Description

Creates a second reference to a subset of the same graphics context.

dispose

public abstract void dispose()

Description

Frees system resources used by graphics context.

draw3DRect

public void draw3DRect (int x, int y, int width, int height, boolean raised)

Parameters

x

x coordinate of the rectangle origin.

y

y coordinate of the rectangle origin

width

Width of the rectangle to draw.

height

Height of the rectangle to draw.

raised

Determines if rectangle drawn is raised or not; `true` for a raised rectangle.

Description

Draws an unfilled 3-D rectangle from (`x`, `y`) of size `width` x `height`.

drawArc

public abstract void drawArc (int x, int y, int width, int height, int startAngle, int arcAngle)

Parameters

x

x coordinate of the bounding rectangle's origin.

y

y coordinate of the bounding rectangle's origin

width

Width of the bounding rectangle for the arc.

height

Height of the bounding rectangle for the arc.

startAngle

Angle at which arc begins, in degrees

arcAngle

length of arc, in degrees

Description

Draws an unfilled arc from `startAngle` to `arcAngle` within bounding rectangle from (`x`, `y`) of size `width` x `height`. Zero degrees is at three o'clock; positive angles are counter clockwise.

drawBytes

public void drawBytes (byte text[], int offset, int length, int x, int y)

Parameters

text

Text to draw, as a byte array.

offset

Starting position within `text` to draw.

length

Number of bytes to draw.

x

x coordinate of baseline origin.

y

y coordinate of baseline origin.

Throws

ArrayIndexOutOfBoundsException

If `offset` or `length` is invalid.

Description

Draws text on screen, starting with `text[offset]` and ending with `text[offset+length-1]`.

drawChars

public void drawChars (char text[], int offset, int length, int x, int y)

Parameters

text

Text to draw, as a char array.

offset

Starting position within `text` to draw.

length

Number of bytes to draw.

x

x coordinate of baseline origin.

y

y coordinate of baseline origin.

Throws

ArrayIndexOutOfBoundsException

If `offset` or `length` is invalid.

Description

Draws text on screen, starting with `text[offset]` and ending with `text[offset+length-1]`.

drawImage

public abstract boolean drawImage (Image image, int x, int y, ImageObserver observer)

Parameters

image

Image to draw.

x

x coordinate of image origin.

y

y coordinate of image origin.

observer

Object that watches for image information; almost always `this`.

Returns

`true` if the image has fully loaded when the method returns, `false` otherwise.

Description

Draws image to screen at (`x`, `y`), at its original size. Drawing may be asynchronous. If `image` is not fully loaded when the method returns, `observer` is notified when additional information made available.

public abstract boolean drawImage (Image image, int x, int y, int width, int height, ImageObserver observer)

Parameters

image

Image to draw.

x

x coordinate of image origin.

y

y coordinate of image origin.

width

New image size in horizontal direction.

height

New image size in vertical direction.

observer

Object that watches for image information; almost always `this`.

Returns

`true` if the image has fully loaded when the method returns, `false` otherwise.

Description

Draws image to screen at (`x`, `y`), scaled to `width` x `height`. Drawing may be asynchronous. If `image` is not fully loaded when the method returns, `observer` is notified when additional information made available.

public abstract boolean drawImage (Image image, int x, int y, Color backgroundColor, ImageObserver observer)

Parameters

image

Image to draw.

x

x coordinate of image origin.

y

y coordinate of image origin.

backgroundColor

Color to show through image where transparent.

observer

Object that watches for image information; almost always `this`.

Returns

`true` if the image has fully loaded when the method returns, `false` otherwise.

Description

Draws image to screen at (`x`, `y`), at its original size. Drawing may be asynchronous. If `image` is not fully loaded when the method returns, `observer` is notified when additional information made available. The background color is visible through any transparent pixels.

public abstract boolean drawImage (Image image, int x, int y, int width, int height, Color backgroundColor, ImageObserver observer)

Parameters

image

Image to draw.

x

x coordinate of image origin.

y

y coordinate of image origin.

width

New image size in horizontal direction.

height

New image size in vertical direction.

backgroundColor

Color to show through image where transparent.

observer

Object that watches for image information; almost always `this`.

Returns

`true` if the image has fully loaded when the method returns, `false` otherwise.

Description

Draws image to screen at (`x`, `y`), scaled to `width` x `height`. Drawing may be asynchronous. If `image` is not fully loaded when the method returns, `observer` is notified when additional information made available. The background color is visible through any transparent pixels.

public abstract boolean drawImage (Image image, int dx1, int dy1, int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, ImageObserver observer) ★

Parameters

image

Image to draw.

dx1

x coordinate of one corner of destination (device) rectangle.

dy1

y coordinate of one corner of destination (device) rectangle.

dx2

x coordinate of the opposite corner of destination (device) rectangle.

dy2

y coordinate of the opposite corner of destination (device) rectangle.

sx1

x coordinate of one corner of source (image) rectangle.

sy1

y coordinate of one corner of source (image) rectangle.

sx2

x coordinate of the opposite corner of source (image) rectangle.

sy2

y coordinate of the opposite corner of source (image) rectangle.

observer

Object that watches for image information; almost always `this`.

Returns

`true` if the image has fully loaded when the method returns, `false` otherwise.

Description

Draws the part of image described by `dx1`, `dy1`, `dx2`, and `dy2` to the screen into the rectangle described by `sx1`, `sy1`, `sx2`, and `sy2`. Drawing may be asynchronous. If `image` is not fully loaded when the method returns, `observer` is notified when additional information is made available.

public abstract boolean drawImage (Image image, int dx1, int dy1, int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, Color backgroundColor, ImageObserver observer) ★

Parameters

image

Image to draw.

dx1

x coordinate of one corner of destination (device) rectangle.

dy1

y coordinate of one corner of destination (device) rectangle.

dx2

x coordinate of the opposite corner of destination (device) rectangle.

dy2

y coordinate of the opposite corner of destination (device) rectangle.

sx1

x coordinate of one corner of source (image) rectangle.

sy1

y coordinate of one corner of source (image) rectangle.

sx2

x coordinate of the opposite corner of source (image) rectangle.

sy2

y coordinate of the opposite corner of source (image) rectangle.

backgroundColor

Color to show through image where transparent.

observer

Object that watches for image information; almost always `this`.

Returns

`true` if the image has fully loaded when the method returns, `false` otherwise.

Description

Draws the part of image described by `dx1`, `dy1`, `dx2`, and `dy2` to the screen into the rectangle described by `sx1`, `sy1`, `sx2`, and `sy2`. Drawing may be asynchronous. If `image` is not fully loaded when the method returns, `observer` is notified when additional information made available. The background color is visible through any transparent pixels.

drawLine

public abstract void drawLine (int x1, int y1, int x2, int y2)

Parameters

x1

x coordinate of one point on line.

y1

y coordinate of one point on line.

x2

x coordinate of the opposite point on line.

y2

y coordinate of the opposite point on line.

Description

Draws a line connecting `(x1, y1)` and `(x2, y2)`.

drawOval

public abstract void drawOval (int x, int y, int width, int height)

Parameters

x

x coordinate of bounding rectangle origin.

y

y coordinate of bounding rectangle origin

width

Width of bounding rectangle to draw in.

height

Height of bounding rectangle to draw in.

Description

Draws an unfilled oval within bounding rectangle from (`x`, `y`) of size `width` x `height`.

drawPolygon

public abstract void drawPolygon (int xPoints[], int yPoints[], int numPoints)

Parameters

xPoints[]

The array of x coordinates for each point.

yPoints[]

The array of y coordinates for each point.

numPoints

The number of elements in both `xPoints` and `yPoints` arrays to use.

Description

Draws an unfilled polygon based on first `numPoints` elements in `xPoints` and `yPoints`.

public void drawPolygon (Polygon p)

Parameters

p

Points of object to draw.

Description

Draws an unfilled polygon based on points within the `Polygon p`.

drawPolyline

public abstract void drawPolyline (int xPoints[], int yPoints[], int nPoints) ★

Parameters

xPoints[]

The array of x coordinates for each point.

yPoints[]

The array of y coordinates for each point.

nPoints

The number of elements in both `xPoints` and `yPoints` arrays to use.

Description

Draws a series of line segments based on first `numPoints` elements in `xPoints` and `yPoints`.

drawRect

public void drawRect (int x, int y, int width, int height)

Parameters

x

x coordinate of rectangle origin.

y

y coordinate of rectangle origin

width

Width of rectangle to draw.

height

Height of rectangle to draw.

Description

Draws an unfilled rectangle from (`x`, `y`) of size `width` x `height`.

drawRoundRect

public abstract void drawRoundRect (int x, int y, int width, int height, int arcWidth, int arcHeight)

Parameters

x

x coordinate of bounding rectangle origin.

y

y coordinate of bounding rectangle origin

width

Width of rectangle to draw.

height

Height of rectangle to draw.

arcWidth

Width of arc of rectangle corner.

arcHeight

Height of arc of rectangle corner.

Description

Draws an unfilled rectangle from (`x`, `y`) of size `width` x `height` with rounded corners.

drawString

public abstract void drawString (String text, int x, int y)

Parameters

text

Text to draw.

x

x coordinate of baseline origin.

y

y coordinate of baseline origin.

Description

Draws text on screen.

fill3DRect

public void fill3DRect (int x, int y, int width, int height, boolean raised)

Parameters

x

x coordinate of rectangle origin.

y

y coordinate of rectangle origin

width

Width of rectangle to draw.

height

Height of rectangle to draw.

raised

`true` to draw a rectangle that appears raised; `false` to draw a rectangle that appears depressed.

Description

Draws a filled 3-D rectangle from (`x`, `y`) of size `width` x `height`.

fillArc

public abstract void fillArc (int x, int y, int width, int height, int startAngle, int arcAngle)

Parameters

x

x coordinate of bounding rectangle origin.

y

y coordinate of bounding rectangle origin

width

Width of bounding rectangle to draw in.

height

Height of bounding rectangle to draw in.

startAngle

Starting angle of arc.

arcAngle

The extent of the arc, measured from startAngle

Description

Draws a filled arc from `startAngle` to `arcAngle` within bounding rectangle from (`x`, `y`) of size `width` x `height`. Zero degrees is at three o'clock; positive angles are counter clockwise.

fillOval

public abstract void fillOval (int x, int y, int width, int height)

Parameters

x

x coordinate of bounding rectangle origin.

y

y coordinate of bounding rectangle origin

width

Width of bounding rectangle to draw in.

height

Height of bounding rectangle to draw in.

Description

Draws filled oval within bounding rectangle from (`x`, `y`) of size `width` x `height`.

fillPolygon

public abstract void fillPolygon (int xPoints[], int yPoints[], int numPoints)

Parameters

xPoints[]

The array of x coordinates for each point.

yPoints[]

The array of y coordinates for each point.

numPoints

The number of elements in both `xPoints` and `yPoints` arrays to use.

Throws

ArrayIndexOutOfBoundsException

If `numPoints` > `xPoints.length` or `numPoints` > `yPoints.length`.

Description

Draws filled polygon based on first `numPoints` elements in `xPoints` and `yPoints`.

public void fillPolygon (Polygon p)

Parameters

p

Points of object to draw.

Description

Draws filled polygon based on points within the `Polygon p`.

fillRect

public abstract void fillRect (int x, int y, int width, int height)

Parameters

x

x coordinate of rectangle origin.

y

y coordinate of rectangle origin

width

Width of rectangle to draw.

height

Height of rectangle to draw.

Description

Draws filled rectangle from (`x`, `y`) of size `width` x `height`.

fillRoundRect

public abstract void fillRoundRect (int x, int y, int width, int height, int arcWidth, int arcHeight)

Parameters

x

x coordinate of bounding rectangle origin.

y

y coordinate of bounding rectangle origin

width

Width of rectangle to draw.

height

Height of rectangle to draw.

arcWidth

Width of arc of rectangle corner.

arcHeight

Height of arc of rectangle corner.

Description

Draws a filled rectangle from (`x`, `y`) of size `width` x `height` with rounded corners.

finalize

public void finalize()

Overrides

`Object.finalize()`

Description

Tells the garbage collector to dispose of graphics context.

getClip

public abstract Shape getClip () ★

Returns

> Shape describing the clipping are of the graphics context.

getClipBounds

public abstract Rectangle getClipBounds() ★

Returns

> Rectangle describing the clipping area of the graphics context.

getClipRect

public abstract Rectangle getClipRect() ☆

Returns

> Replaced by `getClipBounds()`.

getColor

public abstract Color getColor()

Returns

> The current drawing `Color` of the graphics context.

getFont

public abstract Font getFont()

Returns

> The current `Font` of the graphics context.

getFontMetrics

public FontMetrics getFontMetrics()

Returns

The `FontMetrics` of the current font of the graphics context.

public abstract FontMetrics getFontMetrics (Font font)

Parameters

font

Font to get metrics for.

Returns

The `FontMetrics` of the given font for the graphics context.

setClip

public abstract void setClip (int x, int y, int width, int height) ★

Parameters

x

x coordinate of rectangle

y

y coordinate of rectangle

width

width of rectangle

height

height of rectangle

Description

Changes current clipping region to the specified rectangle.

public abstract void setClip (Shape clip) ★

Parameters

clip

The new clipping shape.

Description

Changes current clipping region to the specified shape.

setColor

public abstract void setColor (Color color)

Parameters

color

New color.

Description

Changes current drawing color of graphics context.

setFont

public abstract void setFont (Font font)

Parameters

font

New font.

Description

Changes current font of graphics context.

setPaintMode

public abstract void setPaintMode()

Description

Changes painting mode to normal mode.

setXORMode

public abstract void setXORMode (Color xorColor)

Parameters

xorColor

XOR mode drawing color.

Description

Changes painting mode to XOR mode; in this mode, drawing the same object in the same color at the same location twice has no net effect.

toString

public String toString()

Returns

A string representation of the `Graphics` object.

Overrides

`Object.toString()`

translate

public void translate (int x, int y)

Parameters

x

x coordinate of new drawing origin.

y

y coordinate of new drawing origin.

Description

Moves the origin of drawing operations to (x, y).

See Also

Color, Font, FontMetrics, Image, ImageObserver, Object, Polygon, Rectangle, Shape, String

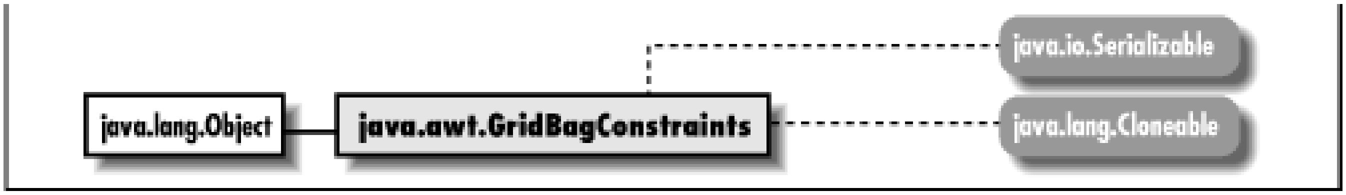
PREVIOUS HOME NEXT
Frame BOOK INDEX GridBagConstraints

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

GridBagConstraints

Name

GridBagConstraints



Description

The `GridBagConstraints` class provides the means to control the layout of components within a `Container` whose `LayoutManager` is `GridBagLayout`.

Class Definition

```
public class java.awt.GridBagConstraints
    extends java.lang.Object
    implements java.lang.Cloneable, java.io.Serializable {

  // Constants
  public final static int BOTH;
  public final static int CENTER;
  public final static int EAST;
  public final static int HORIZONTAL;
  public final static int NONE;
  public final static int NORTH;
  public final static int NORTHEAST;
  public final static int NORTHWEST;
  public final static int RELATIVE;
  public final static int REMAINDER;
  public final static int SOUTH;
  public final static int SOUTHEAST;
  public final static int SOUTHWEST;
  public final static int VERTICAL;
  public final static int WEST;

  // Variables
  public int anchor;
  public int fill;
  public int gridheight;
  public int gridwidth;
  public int gridx;
  public int gridy;
  public Insets insets;
  public int ipadx;
  public int ipady;
  public double weightx
  public double weighty

  // Constructors
  public GridBagConstraints();

  // Instance Methods
  public Object clone();
}
```

Constants

BOTH

public final static int BOTH

Constant for possible `fill` value.

CENTER

public final static int CENTER

Constant for possible `anchor` value.

EAST

public final static int EAST

Constant for possible `anchor` value.

HORIZONTAL

public final static int HORIZONTAL

Constant for possible `fill` value.

NONE

public final static int NONE

Constant for possible `fill` value.

NORTH

public final static int NORTH

Constant for possible `anchor` value.

NORTHEAST

public final static int NORTHEAST

Constant for possible `anchor` value.

NORTHWEST

public final static int NORTHWEST

Constant for possible `anchor` value.

RELATIVE

public final static int RELATIVE

Constant for possible `gridx`, `gridy`, `gridwidth`, or `gridheight` value.

REMAINDER

public final static int REMAINDER

Constant for possible `gridwidth` or `gridheight` value.

SOUTH

public final static int SOUTH

Constant for possible `anchor` value.

SOUTHEAST

public final static int SOUTHEAST

Constant for possible `anchor` value.

SOUTHWEST

public final static int SOUTHWEST

Constant for possible `anchor` value.

VERTICAL

public final static int VERTICAL

Constant for possible `fill` value.

WEST

public final static int WEST

Constant for possible `anchor` value.

Variables

anchor

public int anchor

Specifies the alignment of the component in the event that it is smaller than the space allotted for it by the layout manager; e.g., `CENTER` centers the object within the region.

fill

public int fill

The component's resize policy if additional space available.

gridheight

public int gridheight

Number of columns a component occupies.

gridwidth

public int gridwidth

Number of rows a component occupies.

gridx

public int gridx

Horizontal grid position at which to add component.

gridy

public int gridy

Vertical grid position at which to add component.

insets

public Insets insets

Specifies the outer padding around the component.

ipadx

public int ipadx

Serves as the internal padding within the component in both the right and left directions.

ipady

public int ipady

Serves as the internal padding within the component in both the top and bottom directions.

weightx

public double weightx

Represents the percentage of extra horizontal space that will be given to this component if there is additional space available within the container.

weighty

public double weighty

Represents the percentage of extra vertical space that will be given to this component if there is additional space available within the container.

Constructors

GridBagConstraints

public GridBagConstraints()

Description

> Constructs a `GridBagConstraints` object.

Instance Methods

clone

public Object clone()

Returns

> A new instance of `GridBagConstraints` with same values for constraints.

Overrides

> `Object.clone()`

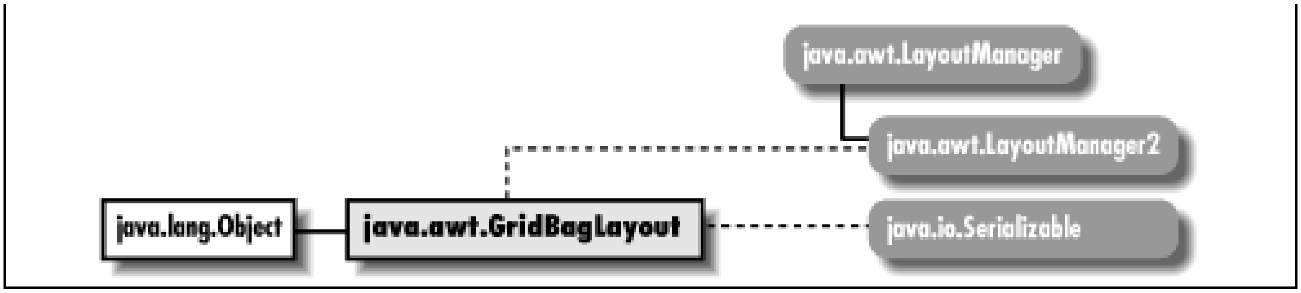
See Also

`Cloneable`, `GridBagLayout`, `Insets`, `Object`, `Serializable`

GridBagLayout

Name

GridBagLayout



Description

The `GridBagLayout LayoutManager` provides the means to layout components in a flexible grid-based display model.

Class Definition

```
public class java.awt.GridBagLayout
    extends java.lang.Object
    implements java.awt.LayoutManager2, java.io.Serializable {

  // Protected Constants
  protected static final MAXGRIDSIZE;
  protected static final MINSIZE;
  protected static final PREFERREDSIZE;

  // Variables
  public double columnWeights[];
  public int columnWidths[];
  public int rowHeights[];
  public double rowWeights[];

  // Protected Variables
  protected Hashtable comptable;
  protected GridBagConstraints defaultConstraints;
  protected GridBagLayoutInfo layoutInfo;

  // Constructors
  public GridBagLayout();

  // Instance Methods
  public void addLayoutComponent (Component comp, Object constraints); ★
  public void addLayoutComponent (String name, Component component);
  public GridBagConstraints getConstraints (Component component);
  public abstract float getLayoutAlignmentX(Container target); ★
  public abstract float getLayoutAlignmentY(Container target); ★
  public int[][] getLayoutDimensions();
  public Point getLayoutOrigin();
  public double[][] getLayoutWeights();
  public abstract void invalidateLayout(Container target); ★
  public void layoutContainer (Container target);
  public Point location (int x, int y);
  public abstract Dimension maximumLayoutSize(Container target); ★
  public Dimension minimumLayoutSize (Container target);
  public Dimension preferredLayoutSize (Container target);
  public void removeLayoutComponent (Component component);
  public void setConstraints (Component component,
      GridBagConstraints constraints);
  public String toString();

  // Protected Instance Methods
  protected void AdjustForGravity (GridBagConstraints constraints,
      Rectangle r);
  protected void ArrangeGrid (Container target);
  protected GridBagLayoutInfo GetLayoutInfo (Container target,
      int sizeFlag);
  protected Dimension GetMinSize (Container target,
      GridBagLayoutInfo info);
  protected GridBagConstraints lookupConstraints (Component comp);
}
```

Protected Constants

MAXGRIDSIZE

protected static final MAXGRIDSIZE

Maximum number of rows and columns within container managed by `GridBagLayout`.

MINSIZE

protected static final MINSIZE

Used for internal sizing purposes.

PREFERREDSIZE

protected static final PREFERREDSIZE

Used for internal sizing purposes.

Variables

columnWeights

public double[] columnWeights

The `weightx` values of the components in the row with the most elements.

columnWidths

public int[] columnWidths

The `width` values of the components in the row with the most elements.

rowHeights

public int[] rowHeights

The `height` values of the components in the column with the most elements.

rowWeights

public double[] rowWeights

The `weighty` values of the components in the column with the most elements.

Protected Variables

comptable

protected Hashtable comptable

Internal table to manage components.

defaultConstraints

protected GridBagConstraints defaultConstraints

Constraints to use for `Component`s that have none.

layoutInfo

protected GridBagLayoutInfo layoutInfo

Internal information about the `GridBagLayout`.

Constructors

GridBagLayout

public GridBagLayout()

Description

> Constructs a `GridBagLayout` object.

Instance Methods

addLayoutComponent

public void addLayoutComponent (Component comp, Object constraints) ★

Parameters

comp

The component being added.

constraints

An object describing the constraints on this component.

Implements

`LayoutManager2.addLayoutComponent()`

Description

Adds the component `comp` to container subject to the given `constraints`. This is a more generalized version of `addLayoutComponent(String, Component)`. It corresponds to `java.awt.Container's add(Component, Object)`.

public void addLayoutComponent (String name, Component component)

Parameters

name

Name of component to add.

component

Actual component being added.

Implements

`LayoutManager.addLayoutComponent()`

Description

Does nothing.

getConstraints

public GridBagConstraints getConstraints (Component component)

Parameters

component

Component whose constraints are desired

Returns

`GridBagConstraints` for component requested.

getLayoutAlignmentX

public abstract float getLayoutAlignmentX (Container target) ★

Parameters

target

The container to inspect.

Returns

The value .5 for all containers.

Description

This method returns the preferred alignment of the given container `target`. A return value of 0 is left aligned, .5 is centered, and 1 is right aligned.

getLayoutAlignmentY

public abstract float getLayoutAlignmentY (Container target) ★

Parameters

target

The container to inspect.

Returns

The value .5 for all containers.

Description

This method returns the preferred alignment of the given container `target`. A return value of 0 is top aligned, .5 is centered, and 1 is bottom aligned.

getLayoutDimensions

public int[][] getLayoutDimensions()

Returns

Returns two single dimension arrays as a multi-dimensional array. Index 0 is an array of widths (`columnWidths` instance variable), while index 1 is an array of heights (`rowHeights` instance variable).

getLayoutOrigin

public Point getLayoutOrigin()

Returns

Returns the origin of the components within the `Container` whose `LayoutManager` is `GridBagLayout`.

getLayoutWeights

public double[][] getLayoutWeights()

Returns

Returns two single dimension arrays as a multi-dimensional array. Index 0 is an array of columns weights (`columnWeights` instance variable), while index 1 is an array of row weights (`rowWeights` instance variable).

invalidateLayout

public abstract void invalidateLayout (Container target) ★

Parameters

target

The container to invalidate.

Description

Does nothing.

layoutContainer

public void layoutContainer (Container target)

Parameters

target

The container that needs to be redrawn.

Implements

`LayoutManager.layoutContainer()`

Description

Draws components contained within `target`.

location

public Point location (int x, int y)

Parameters

x

The x coordinate of the grid position to find.

y

The y coordinate of the grid position to find.

Returns

Returns the grid element under the location provided at position (`x`, `y`) in pixels. Note that the returned Point uses the `GridBagLayout`'s grid for its coordinate space.

Description

Locates the grid position in the `Container` under the given location.

maximumLayoutSize

public abstract Dimension maximumLayoutSize (Container target) ★

Parameters

target

The container to inspect.

Returns

A `Dimension` whose horizontal and vertical components are `Integer.MAX_VALUE`.

Description

For `GridBagLayout`, a maximal `Dimension` is always returned.

minimumLayoutSize

public Dimension minimumLayoutSize (Container target)

Parameters

target

The container whose size needs to be calculated.

Returns

Minimum `Dimension` of container target.

Implements

`LayoutManager.minimumLayoutSize()`

Description

Calculates minimum size of `target` container.

preferredLayoutSize

public Dimension preferredLayoutSize (Container target)

Parameters

target

The container whose size needs to be calculated.

Returns

Preferred `Dimension` of container target

Implements

`LayoutManager.preferredLayoutSize()`

Description

Calculates preferred size of `target` container.

removeLayoutComponent

public void removeLayoutComponent (Component component)

Parameters

component

Component to stop tracking.

Implements

`LayoutManager.removeLayoutComponent()`

Description

Does nothing.

setConstraints

public void setConstraints (Component component, GridBagConstraints constraints)

Parameters

component

Component to set constraints for

constraints

Constraints for component

Description

Changes the `GridBagConstraints` on `component` to those provided.

toString

public String toString()

Returns

A string representation of the `GridBagLayout` object.

Overrides

`Object.toString()`

Protected Instance Methods

AdjustForGravity

protected void AdjustForGravity (GridBagConstraints constraints, Rectangle r)

Parameters

constraints

Constraints to use for adjustment of Rectangle.

r

Rectangular area that needs to be adjusted.

Description

Helper routine for laying out a cell of the grid. The routine adjusts the values for `r` based upon the `constraints`.

ArrangeGrid

protected void ArrangeGrid (Container target)

Parameters

target

`Container` to layout.

Description

Helper routine that does the actual arrangement of components in `target`.

GetLayoutInfo

protected GridBagLayoutInfo GetLayoutInfo (Container target, int sizeFlag)

Parameters

target

Container to get information about.

sizeFlag

One of the constants MINSIZE or PREFERREDSIZE.

Returns

Returns an internal class used to help size the container.

GetMinSize

protected Dimension GetMinSize (Container target, GridBagLayoutInfo info)

Parameters

target

Container to calculate size.

info

Specifics about the container's constraints.

Returns

Minimum Dimension of container target based on info.

Description

Helper routine for calculating size of container.

lookupConstraints

protected GridBagConstraints lookupConstraints (Component comp)

Parameters

comp

`Component` in question.

Returns

A reference to the `GridBagConstraints` object for this component.

Description

Helper routine for calculating size of container.

See Also

`Component`, `Container`, `Dimension`, `GridBagConstraints`, `Hashtable`, `LayoutManager`, `LayoutManager2`, `Object`, `Point`, `Rectangle`, `String`

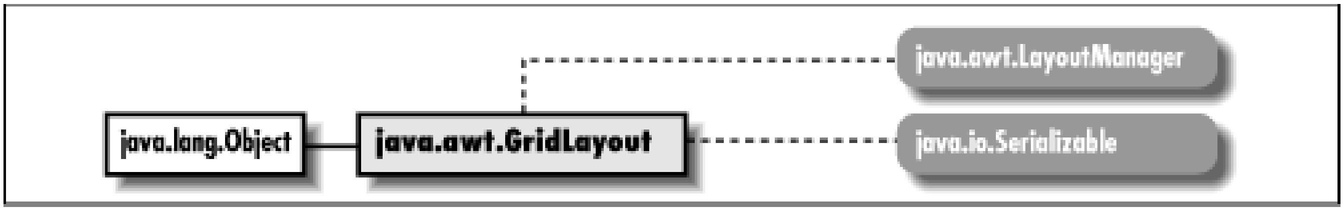
PREVIOUS GridBagConstraints | HOME BOOK INDEX | NEXT GridLayout

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

GridLayout

Name

GridLayout



Description

The `GridLayout LayoutManager` provides the means to layout components in a grid of rows and columns.

Class Definition

```
public class java.awt.GridLayout
        extends java.lang.Object
        implements java.awt.LayoutManager, java.io.Serializable
 {

// Constructors
  public GridLayout(); ★
  public GridLayout (int rows, int cols);
  public GridLayout (int rows, int cols, int hgap, int vgap);
// Instance Methods
  public void addLayoutComponent (String name, Component component);
  public int getColumns(); ★
  public int getHgap(); ★
  public int getRows(); ★

  public int getVgap(); ★
  public void layoutContainer (Container target);
  public Dimension minimumLayoutSize (Container target);
  public Dimension preferredLayoutSize (Container target);
  public void removeLayoutComponent (Component component);
  public int setColumns(int cols); ★
  public int setHgap(int hgap); ★
  public int setRows(int rows); ★

  public int setVgap(int vgap); ★
  public String toString();
}
```

Constructors

GridLayout

public GridLayout() ★

Description

> Constructs a `GridLayout` object with a default single row and one column per component.

public GridLayout (int rows, int cols)

Parameters

> *rows*
>
> > Requested number of rows in container.
>
> *cols*

Requested number of columns in container.

Description

Constructs a `GridLayout` object with the requested number of rows and columns. Note that the actual number of rows and columns depends on the number of objects in the layout, not the constructor's parameters.

public GridLayout (int rows, int cols, int hgap, int vgap)

Parameters

rows

Requested number of rows in container.

cols

Requested number of columns in container.

hgap

Horizontal space between each component in a row.

vgap

Vertical space between each row.

Description

Constructs a `GridLayout` object with the requested number of `rows` and `columns` and the values specified as the gaps between each component. Note that the actual number of rows and columns depends on the number of objects in the layout, not the constructor's parameters.

Instance Methods

addLayoutComponent

public void addLayoutComponent (String name, Component component)

Parameters

name

Name of component to add.

component

Actual component being added.

Implements

`LayoutManager.addLayoutComponent()`

Description

Does nothing.

getColumns

public int getColumns() ★

Returns

The number of columns.

getHgap

public int getHgap() ★

Returns

The horizontal gap for this `GridLayout` instance.

getRows

public int getRows() ★

Returns

The number of rows.

getVgap

public int getVgap() ★

Returns

The vertical gap for this `GridLayout` instance.

layoutContainer

public void layoutContainer (Container target)

Parameters

target

The container that needs to be redrawn.

Implements

`LayoutManager.layoutContainer()`

Description

Draws the components contained within the `target`.

minimumLayoutSize

public Dimension minimumLayoutSize (Container target)

Parameters

target

The container whose size needs to be calculated.

Returns

Minimum `Dimension` of the container `target`.

Implements

`LayoutManager.minimumLayoutSize()`

Description

Calculates the minimum size of the `target` container.

preferredLayoutSize

public Dimension preferredLayoutSize (Container target)

Parameters

target

The container whose size needs to be calculated.

Returns

Preferred `Dimension` of the container `target`.

Implements

`LayoutManager.preferredLayoutSize()`

Description

Calculates the preferred size of the `target` container.

removeLayoutComponent

public void removeLayoutComponent (Component component)

Parameters

component

Component to stop tracking.

Implements

`LayoutManager.removeLayoutComponent()`

Description

Does nothing.

setColumns

public void setColumns(int cols) ★

Parameters

cols

The new number of columns.

Description

Sets the number of columns.

setHgap

public void setHgap(int hgap) ★

Parameters

hgap

The horizontal gap value.

Description

Sets the horizontal gap between components.

setRows

public void setRows(int rows) ★

Parameters

rows

The new number of rows.

Description

Sets the number of rows.

setVgap

public void setVgap(int vgap) ★

Parameters

vgap

The vertical gap value.

Description

Sets the vertical gap between components.

toString

public String toString()

Returns

A string representation of the `GridLayout` object.

Overrides

`Object.toString()`

See Also

`Component`, `Container`, `Dimension`, `LayoutManager`, `Object`, `String`

PREVIOUS | HOME | NEXT
GridBagLayout | BOOK INDEX | IllegalComponentStateException

★

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

IllegalComponentStateException ★

Name

IllegalComponentStateException ★

[Graphic: Figure from the text]

Description

An `Exception` indicating that a `Component` was not in an appropriate state to perform a requested action.

Class Definition

```
public class java.awt.IllegalComponentStateException
    extends java.lang.IllegalStateException {

  // Constructors
  public IllegalComponentStateException();
  public IllegalComponentStateException (String s);
}
```

Constructors

IllegalComponentStateException

public IllegalComponentStateException()

Description

Constructs the exception object with no detail message.

public IllegalComponentStateException (String s)

Parameters

s

Detail message

Description

Constructs the exception object with the given detail message.

See Also

`Exception`, `String`

PREVIOUS | HOME | NEXT
GridLayout | BOOK INDEX | Image

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

SystemColor ★

Name

SystemColor ★

[Graphic: Figure from the text]

Description

`SystemColor` provides information on the colors that the windowing system uses to display windows and other graphic components. Most windowing systems allow the user to choose different color schemes; `SystemColor` enables programs to find out what colors are in use in order to paint themselves in a consistent manner.

Class Definition

```
public final class java.awt.SystemColor
    extends java.awt.Color
    implements java.io.Serializable {
  // Constants
  public final static int ACTIVE_CAPTION;
  public final static int ACTIVE_CAPTION_BORDER;
  public final static int ACTIVE_CAPTION_TEXT;
  public final static int CONTROL;
  public final static int CONTROL_DK_SHADOW;
  public final static int CONTROL_HIGHLIGHT;
  public final static int CONTROL_LT_HIGHLIGHT;
  public final static int CONTROL_SHADOW;
  public final static int CONTROL_TEXT;
  public final static int DESKTOP;
  public final static int INACTIVE_CAPTION;
  public final static int INACTIVE_CAPTION_BORDER;
  public final static int INACTIVE_CAPTION_TEXT;
  public final static int INFO;
  public final static int INFO_TEXT;
  public final static int MENU;
  public final static int MENU_TEXT;
  public final static int NUM_COLORS;
  public final static int SCROLLBAR;
  public final static int TEXT;
  public final static int TEXT_HIGHLIGHT;
  public final static int TEXT_HIGHLIGHT_TEXT;
  public final static int TEXT_INACTIVE_TEXT;
  public final static int TEXT_TEXT;
  public final static int WINDOW;
  public final static int WINDOW_BORDER;
  public final static int WINDOW_TEXT;
  public final static SystemColor activeCaption;
  public final static SystemColor activeCaptionBorder;
  public final static SystemColor activeCaptionText;
  public final static SystemColor control;
  public final static SystemColor controlDkShadow;
  public final static SystemColor controlHighlight;
  public final static SystemColor controlLtHighlight;
  public final static SystemColor controlShadow;
  public final static SystemColor controlText;
  public final static SystemColor desktop;
  public final static SystemColor inactiveCaption;
  public final static SystemColor inactiveCaptionBorder;
  public final static SystemColor inactiveCaptionText;
  public final static SystemColor info;
  public final static SystemColor infoText;
  public final static SystemColor menu;
  public final static SystemColor menuText;
  public final static SystemColor scrollbar;
  public final static SystemColor text;
  public final static SystemColor textHighlight;
  public final static SystemColor textHighlightText;
```

```
  public final static SystemColor textInactiveText;
  public final static SystemColor textText;
  public final static SystemColor window;
  public final static SystemColor windowBorder;
  public final static SystemColor windowText;
  // Public Instance Methods
  public int getRGB();
  public String toString();
}
```

Constants

ACTIVE_CAPTION

public static final int ACTIVE_CAPTION

ACTIVE_CAPTION_BORDER

public static final int ACTIVE_CAPTION_BORDER

ACTIVE_CAPTION_TEXT

public static final int ACTIVE_CAPTION_TEXT

CONTROL

public static final int CONTROL

CONTROL_DK_SHADOW

public static final int CONTROL_DK_SHADOW

CONTROL_HIGHLIGHT

public static final int CONTROL_HIGHLIGHT

CONTROL_LT_HIGHLIGHT

public static final int CONTROL_LT_HIGHLIGHT

CONTROL_SHADOW

public static final int CONTROL_SHADOW

CONTROL_TEXT

public static final int CONTROL_TEXT

DESKTOP

public static final int DESKTOP

INACTIVE_CAPTION

public static final int INACTIVE_CAPTION

INACTIVE_CAPTION_BORDER

public static final int INACTIVE_CAPTION_BORDER

INACTIVE_CAPTION_TEXT

public static final int INACTIVE_CAPTION_TEXT

INFO

public static final int INFO

INFO_TEXT

public static final int INFO_TEXT

MENU

public static final int MENU

MENU_TEXT

public static final int MENU_TEXT

NUM_COLORS

public static final int NUM_COLORS

SCROLLBAR

public static final int SCROLLBAR

TEXT

public static final int TEXT

TEXT_HIGHLIGHT

public static final int TEXT_HIGHLIGHT

TEXT_HIGHLIGHT_TEXT

public static final int TEXT_HIGHLIGHT_TEXT

TEXT_INACTIVE_TEXT

public static final int TEXT_INACTIVE_TEXT

TEXT_TEXT

public static final int TEXT_TEXT

WINDOW

public static final int WINDOW

WINDOW_BORDER

public static final int WINDOW_BORDER

WINDOW_TEXT

public static final int WINDOW_TEXT

activeCaption

public static final SystemColor activeCaption

Background color for captions in window borders.

activeCaptionBorder

public static final SystemColor activeCaptionBorder

Border color for captions in window borders.

activeCaptionText

public static final SystemColor activeCaptionText

Text color for captions in window borders.

control

public static final SystemColor control

Background color for controls.

controlDkShadow

public static final SystemColor controlDkShadow

Dark shadow color for controls.

controlHighlight

public static final SystemColor controlHighlight

Highlight color for controls.

controlLtHighlight

public static final SystemColor controlLtHighlight

Light highlight color for controls.

controlShadow

public static final SystemColor controlShadow

Shadow color for controls.

controlText

public static final SystemColor controlText

Text color for controls.

desktop

public static final SystemColor desktop

Desktop background color.

inactiveCaption

public static final SystemColor inactiveCaption

Background color for inactive captions in window borders.

inactiveCaptionBorder

public static final SystemColor inactiveCaptionBorder

Border color for inactive captions in window borders.

inactiveCaptionText

public static final SystemColor inactiveCaptionText

Text color for inactive captions in window borders.

info

public static final SystemColor info

Background color for informational text.

infoText

public static final SystemColor infoText

Text color for informational text.

menu

public static final SystemColor menu

Background color for menus.

menuText

public static final SystemColor menuText

Text color for menus.

scrollbar

public static final SystemColor scrollbar

Background color for scrollbars.

text

public static final SystemColor text

Background color for text components.

textHighlight

public static final SystemColor textHighlight

Background color for highlighted text.

textHighlightText

public static final SystemColor textHighlightText

Text color for highlighted text.

textInactiveText

public static final SystemColor textInactiveText

Text color for inactive text.

textText

public static final SystemColor textText

Text color for text components.

window

public static final SystemColor window

Background color for windows.

windowBorder

public static final SystemColor windowBorder

Border color for windows.

windowText

public static final SystemColor windowText

Text color for windows.

Instance Methods

getRGB

public int getRGB()

Returns

> Current color as a composite value

Overrides

> `Color.getRGB()`

Description

> Gets integer value of current system color.

toString

public String toString()

Returns

> A string representation of the `SystemColor` object.

Overrides

> `Color.toString()`

See Also

`Color`, `Serializable`, `String`

ClipboardOwner ★

Name

ClipboardOwner ★

[Graphic: Figure from the text]

Description

`ClipboardOwner` is implemented by classes that want to be notified when someone else sets the contents of a clipboard.

Interface Definition

```
public abstract interface java.awt.datatransfer.ClipboardOwner {
  // Interface Methods
  public abstract void lostOwnership (Clipboard clipboard, Transferable contents);
}
```

Interface Methods

lostOwnership

public abstract void lostOwnership (Clipboard clipboard, Transferable contents)

Parameters

clipboard

The clipboard whose contents have changed.

contents

The contents that this owner originally put on the clipboard.

Description

Tells the `ClipboardOwner` that the `contents` it placed on the given `clipboard` are no longer there.

See Also

`Clipboard, StringSelection, Transferable`

PREVIOUS
Clipboard ★

HOME
BOOK INDEX

NEXT
DataFlavor ★

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

DataFlavor ★

Name

DataFlavor ★

[Graphic: Figure from the text]

Description

The `DataFlavor` class encapsulates information about data formats.

Class Definition

```
public class java.awt.datatransfer.DataFlavor
   extends java.lang.Object {
  // Class Variables
  public static DataFlavor plainTextFlavor;
  public static DataFlavor stringFlavor;
  // Constructors
  public DataFlavor (Class representationClass,
    String humanPresentableName);
  public DataFlavor (String MIMEType, String humanPresentableName);
  // Instance Methods
  public boolean equals (DataFlavor dataFlavor);
  public String getHumanPresentableName();
  public String getMIMEType();
  public Class getRepresentationClass();
  public boolean isMIMETypeEqual (String MIMEType);
  public final boolean isMIMETypeEqual (DataFlavor dataFlavor);
  public void setHumanPresentableName (String humanPresentableName);
  // Protected Instance Methods
  protected String normalizeMIMEType (String MIMEType);
  protected String normalizeMIMETypeParameter (String parameterName,
    String parameterValue);
}
```

Class Variables

plainTextFlavor

public static DataFlavor plainTextFlavor

A preset `DataFlavor` object representing plain text.

stringFlavor

public static DataFlavor stringFlavor

A preset `DataFlavor` object representing a Java `String`.

Constructors

DataFlavor

public DataFlavor (Class representationClass, String humanPresentableName)

Parameters

representationClass

The Java class that represents data in this flavor.

humanPresentableName

A name for this flavor that humans will recognize.

Description

Constructs a `DataFlavor` object with the given characteristics. The MIME type for this `DataFlavor` is `application/x-java-serialized-object <Java ClassName>`.[1]

[1] The type name changed to `x-java-serialized-object` in the 1.1.1 release.

public DataFlavor (String MIMEType, String humanPresentableName)

Parameters

MIMEType

The MIME type string this `DataFlavor` represents.

humanPresentableName

A name for this flavor that humans will recognize.

Description

Constructs a `DataFlavor` object with the given characteristics. The representation class used for this `DataFlavor` is `java.io.InputStream`.

Instance Methods

equals

public boolean equals (DataFlavor dataFlavor)

Parameters

dataFlavor

The flavor to compare.

Returns

`true` if `dataFlavor` is equivalent to this `DataFlavor`, `false` otherwise.

Description

Compares two different `DataFlavor` instances for equivalence.

getHumanPresentableName

public String getHumanPresentableName()

Returns

The name of this flavor.

getMIMEType

public String getMIMEType()

Returns

The MIME type string for this flavor.

getRepresentationClass

public Class getRepresentationClass()

Returns

The Java class that will be used to represent data in this flavor.

isMIMETypeEqual

public boolean isMIMETypeEqual (String MIMEType)

Parameters

MIMEType

The type to compare.

Returns

`true` if the given MIME type is the same as this `DataFlavor`'s MIME type; `false` otherwise.

Description

Compares two different `DataFlavor` MIME types for equivalence.

public final boolean isMIMETypeEqual (DataFlavor dataFlavor)

Parameters

dataFlavor

The flavor to compare.

Returns

`true` if `DataFlavor`'s MIME type is the same as this `DataFlavor`'s MIME type; `false` otherwise.

Description

Compares two different `DataFlavor` MIME types for equivalence.

setHumanPresentableName

public void setHumanPresentableName (String humanPresentableName)

Parameters

humanPresentableName

A name for this flavor that humans will recognize.

Description

Changes the name of the `DataFlavor`.

Protected Instance Methods

normalizeMIMEType

protected String normalizeMIMEType (String MIMEType)

Parameters

MIMEType

The MIME type string to normalize.

Returns

Normalized MIME type string.

Description

This method is called for each MIME type string. Subclasses can override this method to add default parameter/value pairs to MIME strings.

normalizeMIMETypeParameter

protected String normalizeMIMETypeParameter (String parameterName, String parameterValue)

Parameters

parameterName

The MIME type parameter to normalize.

parameterValue

The corresponding value.

Returns

Normalized MIME type parameter string.

Description

> This method is called for each MIME type parameter string. Subclasses can override this method to handle special parameters, such as those that are case-insensitive.

See Also

`Class`, `String`

PREVIOUS | HOME | NEXT
ClipboardOwner ★ | BOOK INDEX | StringSelection ★

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

StringSelection ★

Name

StringSelection ★

[Graphic: Figure from the text]

Description

`StringSelection` is a "convenience" class that can be used for copy and paste operations on Unicode text strings. For example, you could place a string on the system's clipboard with the following code:

```
Clipboard c =
  Toolkit.getDefaultToolkit().getSystemClipboard();
StringSelection s = new StringSelection(
  "Be safe when you cut and paste.");
c.setContents(s, s);
```

Class Definition

```
public class java.awt.datatransfer.StringSelection
   extends java.lang.Object
   implements java.awt.datatransfer.ClipboardOwner,
         java.awt.datatransfer.Transferable {
```

```
  // Constructor
  public StringSelection(String data);
  // Instance Methods
  public synchronized Object getTransferData (DataFlavor flavor)
    throws UnsupportedFlavorException, IOException;
  public synchronized DataFlavor[] getTransferDataFlavors();
  public boolean isDataFlavorSupported (DataFlavor flavor);
  public void lostOwnership (Clipboard clipboard, Transferable contents);
}
```

Constructors

StringSelection

public StringSelection (String data)

Parameters

data

The string to be placed in a clipboard.

Description

Constructs a `StringSelection` object from the given string.

Instance Methods

getTransferData

public synchronized Object getTransferData (DataFlavor flavor) throws UnsupportedFlavorException, IOException

Parameters

flavor

The requested flavor for the returned data, which can be either `DataFlavor.stringFlavor` or `DataFlavor.plainTextFlavor`.

Returns

The string that the `StringSelection` was constructed with. This is returned either as a `String` object or a `Reader` object, depending on the flavor requested.

Throws

UnsupportedFlavorException

If the requested flavor is not supported.

IOException

If a `Reader` representing the string could not be created.

Implements

`Transferable.getTransferData(DataFlavor)`

Description

Returns the string this `StringSelection` represents. This is returned either as a `String` object or a `Reader` object, depending on the flavor requested.

getTransferDataFlavors

public synchronized DataFlavor[] getTransferDataFlavors()

Returns

An array of the data flavors the `StringSelection` supports.

Implements

`Transferable.getTransferDataFlavors()`

Description

`DataFlavor.stringFlavor` and `DataFlavor.plainTextFlavor` are returned.

isDataFlavorSupported

public boolean isDataFlavorSupported (DataFlavor flavor)

Parameters

flavor

The flavor in question.

Returns

`true` if `flavor` is supported; `false` otherwise.

Implements

`Transferable.isDataFlavorSupported(DataFlavor)`

lostOwnership

public void lostOwnership (Clipboard clipboard, Transferable contents)

Parameters

clipboard

The clipboard whose contents are changing.

contents

The contents that were on the clipboard.

Implements

`ClipboardOwner.lostOwnership(Clipboard, Transferable)`

Description

Does nothing.

See Also

`Clipboard`, `ClipboardOwner`, `DataFlavor`, `String`, `Transferable`

Transferable ★

Name

Transferable ★

[Graphic: Figure from the text]

Description

The `Transferable` interface is implemented by objects that can be placed on `Clipboard`s.

Interface Definition

```
public abstract interface Transferable {
  // Instance Methods
  public abstract Object getTransferData (DataFlavor flavor)
    throws UnsupportedFlavorException, IOException;
  public abstract DataFlavor[] getTransferDataFlavors();
  public abstract boolean isDataFlavorSupported (DataFlavor flavor);
}
```

Interface Methods

getTransferData

public abstract Object getTransferData (DataFlavor flavor) throws UnsupportedFlavorException, IOException

Parameters

flavor

The requested flavor for the returned data.

Returns

The data represented by this `Transferable` object, in the requested flavor.

Throws

UnsupportedFlavorException

If the requested flavor is not supported.

IOException

If a `Reader` representing the data could not be created.

Description

Returns the data this `Transferable` object represents. The class of object returned depends on the flavor requested.

getTransferDataFlavors

public abstract DataFlavor[] getTransferDataFlavors()

Returns

An array of the supported data flavors.

Description

The data flavors should be returned in order, sorted from most to least descriptive.

isDataFlavorSupported

public abstract boolean isDataFlavorSupported (DataFlavor flavor)

Parameters

flavor

The flavor in question.

Returns

`true` if `flavor` is supported; `false` otherwise.

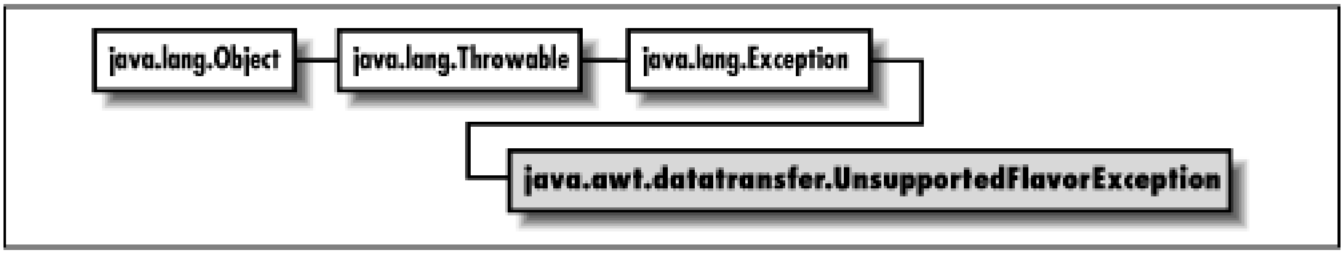
See Also

`Clipboard`, `DataFlavor`, `Reader`, `StringSelection`, `Transferable`

UnsupportedFlavorException ★

Name

UnsupportedFlavorException ★



Description

This exception is thrown from `Transferable.getTransferData(DataFlavor)` to indicate that the `DataFlavor` requested is not available.

Class Definition

```
public class java.awt.datatransfer.UnsupportedFlavorException
   extends java.lang.Exception {
  // Constructor
  public UnsupportedFlavorException (DataFlavor flavor);
}
```

Constructors

UnsupportedFlavorException

public UnsupportedFlavorException (DataFlavor flavor)

Parameters

flavor

The flavor that caused the exception.

See Also

`DataFlavor`, `Exception`, `Transferable`

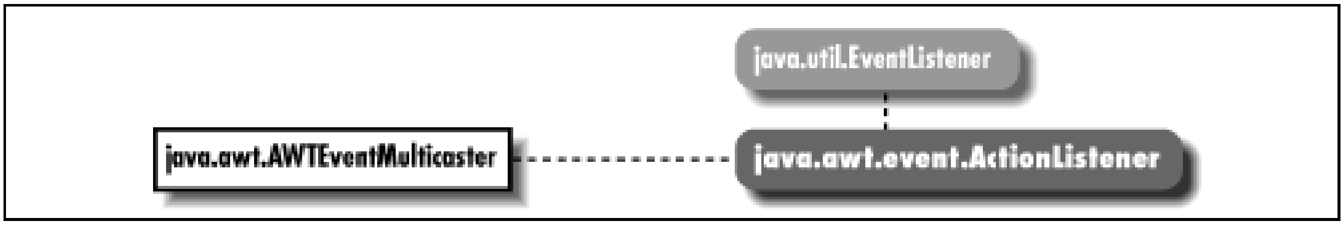
PREVIOUS | HOME | NEXT
Transferable | BOOK INDEX | ActionEvent

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

ActionListener ★

Name

ActionListener ★



Description

Objects that implement the `ActionListener` interface can receive `ActionEvent` objects. Listeners must first register themselves with objects that produce events. When events occur, they are then automatically propagated to all registered listeners.

Interface Definition

```
public abstract interface java.awt.event.ActionListener
   extends java.util.EventListener {
  // Interface Methods
  public abstract void actionPerformed (ActionEvent e);
}
```

Interface Methods

actionPerformed

public abstract void actionPerformed (ActionEvent e)

Parameters

e

The action event that occurred.

Description

Notifies the `ActionListener` that an event occurred.

See Also

`ActionEvent`, `AWTEventMultisaster`, `EventListener`

PREVIOUS | HOME | NEXT
ActionEvent ★ | BOOK INDEX | AdjustmentEvent ★

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

AdjustmentEvent ★

Name

AdjustmentEvent ★

[Graphic: Figure from the text]

Description

`AdjustmentEvent`s are generated by objects that implement the `Adjustable` interface. `Scrollbar` is one example of such an object.

Class Definition

```
public class java.awt.event.AdjustmentEvent
   extends java.awt.AWTEvent {
  // Constants
  public final static int ADJUSTMENT_FIRST;
  public final static int ADJUSTMENT_LAST;
  public final static int ADJUSTMENT_VALUE_CHANGED;
  public final static int BLOCK_DECREMENT;
  public final static int BLOCK_INCREMENT;
  public final static int TRACK;
  public final static int UNIT_DECREMENT;
  public final static int UNIT_INCREMENT;
  // Constructors
  public AdjustmentEvent (Adjustable source, int id, int type, int value);
  // Instance Methods
  public Adjustable getAdjustable();
  public int getAdjustmentType();
  public int getValue();
  public String paramString();
}
```

Constants

ADJUSTMENT_FIRST

public final static int ADJUSTMENT_FIRST

Specifies the beginning range of adjustment event ID values.

ADJUSTMENT_LAST

public final static int ADJUSTMENT_LAST

Specifies the ending range of adjustment event ID values.

ADJUSTMENT_VALUE_CHANGED

public final static int ADJUSTMENT_VALUE_CHANGED

Event type ID for value changed.

BLOCK_DECREMENT

public final static int BLOCK_DECREMENT

Adjustment type for block decrement.

BLOCK_INCREMENT

public final static int BLOCK_INCREMENT

Adjustment type for block increment.

TRACK

public final static int TRACK

Adjustment type for tracking.

UNIT_DECREMENT

public final static int UNIT_DECREMENT

Adjustment type for unit decrement.

UNIT_INCREMENT

public final static int UNIT_INCREMENT

Adjustment type for unit increment.

Constructors

AdjustmentEvent

public AdjustmentEvent (Adjustable source, int id, int type, int value)

Parameters

source

The object that generated the event.

id

The event type ID of the event.

type

The type of adjustment event.

value

The value of the `Adjustable` object.

Description

Constructs an `AdjustmentEvent` with the given characteristics.

Instance Methods

getAdjustable

public Adjustable getAdjustable()

Returns

The source of this event.

getAdjustmentType

public int getAdjustmentType()

Returns

One of the adjustment type constants.

Description

The type will be `BLOCK_DECREMENT`, `BLOCK_INCREMENT`, `TRACK`, `UNIT_DECREMENT`, or `UNIT_INCREMENT`.

getValue

public int getValue()

Returns

The new value of the `Adjustable` object.

paramString

public String paramString()

Returns

String with current settings of the `AdjustmentEvent`.

Overrides

`AWTEvent.paramString()`

Description

Helper method for `toString()` to generate string of current settings.

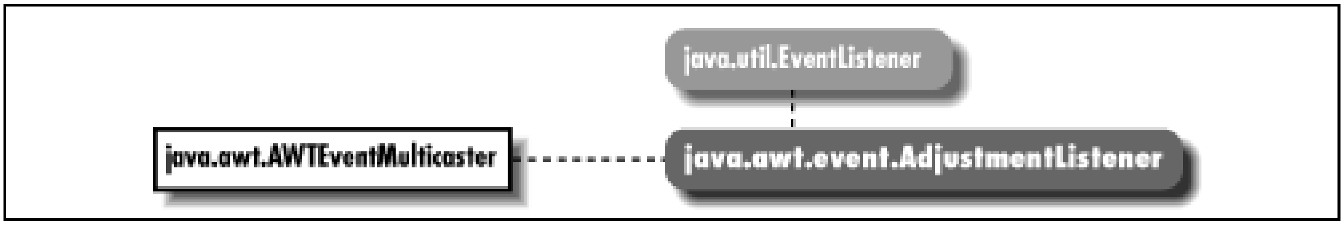
See Also

`Adjustable`, `AdjustmentListener`, `AWTEvent`, `Scrollbar`

AdjustmentListener ★

Name

AdjustmentListener ★


java.util.EventListener
java.awt.AWTEventMulticaster
java.awt.event.AdjustmentListener

Description

Objects that implement the `AdjustmentListener` interface can receive `AdjustmentEvent` objects. Listeners must first register themselves with objects that produce events. When events occur, they are then automatically propagated to all registered listeners.

Interface Definition

```
public abstract interface java.awt.event.AdjustmentListener
   extends java.util.Eventlistener {
  // Interface Methods
  public abstract void adjustmentValueChanged (AdjustmentEvent e);
}
```

Interface Methods

adjustmentPerformed

public abstract void adjustmentValueChanged (AdjustmentEvent e)

Parameters

e

The adjustment event that occurred.

Description

Notifies the `AdjustmentListener` that an event occurred.

See Also

`AdjustmentEvent`, `AWTEventMultiCaster`, `EventListener`

PREVIOUS | HOME | NEXT

AdjustmentEvent ★ | BOOK INDEX | ComponentAdapter ★

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

ComponentAdapter ★

Name

ComponentAdapter ★

[Graphic: Figure from the text]

Description

`ComponentAdapter` is a class that implements the methods of `ComponentListener` with empty functions. It may be easier for you to extend `ComponentAdapter`, overriding only those methods you are interested in, than to implement `ComponentListener` and provide the empty functions yourself.

Class Definition

```
public abstract class java.awt.event.ComponentAdapter
   extends java.lang.Object
   implements java.awt.event.ComponentListener {
  // Instance Methods
  public void componentHidden (ComponentEvent e);
  public void componentMoved (ComponentEvent e);
  public void componentResized (ComponentEvent e);
  public void componentShown (ComponentEvent e);
}
```

Instance Methods

componentHidden

public void componentHidden (ComponentEvent e)

Parameters

e

The event that has occurred.

Description

Does nothing. Override this function to be notified when a component is hidden.

componentMoved

public void componentMoved (ComponentEvent e)

Parameters

e

The event that has occurred.

Description

Does nothing. Override this function to be notified when a component is moved.

componentResized

public void componentResized (ComponentEvent e)

Parameters

e

The event that has occurred.

Description

Does nothing. Override this function to be notified when a component is resized.

componentShown

public void componentShown (ComponentEvent e)

Parameters

e

The event that has occurred.

Description

Does nothing. Override this function to be notified when a component is shown.

See Also

`Component`, `ComponentEvent`, `ComponentListener`

PREVIOUS
AdjustmentListener ★

HOME
BOOK INDEX

NEXT
ComponentEvent ★

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

ComponentEvent ★

Name

ComponentEvent ★

[Graphic: Figure from the text]

Description

Component events are generated when a component is shown, hidden, moved, or resized. AWT automatically deals with component moves and resizing; these events are provided only for notification. Subclasses of `ComponentEvent` deal with other specific component-level events.

Class Definition

```
public class java.awt.event.ComponentEvent
```

```
   extends java.awt.AWTEvent {
  // Constants
  public final static int COMPONENT_FIRST;
  public final static int COMPONENT_HIDDEN;
  public final static int COMPONENT_LAST;
  public final static int COMPONENT_MOVED;
  public final static int COMPONENT_RESIZED;
  public final static int COMPONENT_SHOWN;
  // Constructors
  public ComponentEvent (Component source, int id);
  // Instance Methods
  public Component getComponent();
  public String paramString();
}
```

Constants

COMPONENT_FIRST

public final static int COMPONENT_FIRST

Specifies the beginning range of component event ID values.

COMPONENT_HIDDEN

public final static int COMPONENT_HIDDEN

Event type ID indicating that the component was hidden.

COMPONENT_LAST

public final static int COMPONENT_LAST

Specifies the ending range of component event ID values.

COMPONENT_MOVED

public final static int COMPONENT_MOVED

Event type ID indicating that the component was moved.

COMPONENT_RESIZED

public final static int COMPONENT_RESIZED

Event type ID indicating that the component was resized.

COMPONENT_SHOWN

public final static int COMPONENT_SHOWN

Event type ID indicating that the component was shown.

Constructors

ComponentEvent

public ComponentEvent (Component source, int id)

Parameters

source

The object that generated the event.

id

The event type ID of the event.

Description

Constructs a `ComponentEvent` with the given characteristics.

Instance Methods

getComponent

public Component getComponent()

Returns

The source of this event.

paramString

public String paramString()

Returns

String with current settings of the `ComponentEvent`.

Overrides

`AWTEvent.paramString()`

Description

Helper method for `toString()` to generate string of current settings.

See Also

`AWTEvent`, `Component`, `ComponentAdapter`, `ComponentListener`, `ContainerEvent`, `FocusEvent`, `InputEvent`, `PaintEvent`, `WindowEvent`

ComponentListener ★

Name

ComponentListener ★

[Graphic: Figure from the text]

Description

Objects that implement the `ComponentListener` interface can receive `ComponentEvent` objects. Listeners must first register themselves with objects that produce events. When events occur, they are then automatically propagated to all registered listeners.

Interface Definition

```
public abstract interface java.awt.event.ComponentListener
   extends java.util.EventListener {
  // Instance Methods
  public abstract void componentHidden (ComponentEvent e);
  public abstract void componentMoved (ComponentEvent e);
  public abstract void componentResized (ComponentEvent e);
  public abstract void componentShown (ComponentEvent e);
}
```

Interface Methods

componentHidden

public abstract void componentHidden (ComponentEvent e)

Parameters

e

The component event that occurred.

Description

Notifies the `ComponentListener` that a component was hidden.

componentMoved

public abstract void componentMoved (ComponentEvent e)

Parameters

e

The component event that occurred.

Description

Notifies the `ComponentListener` that a component was moved.

componentResized

public abstract void componentResized (ComponentEvent e)

Parameters

e

The component event that occurred.

Description

Notifies the `ComponentListener` that a component was resized.

componentShown

public abstract void componentShown (ComponentEvent e)

Parameters

e

The component event that occurred.

Description

Notifies the `ComponentListener` that a component was shown.

See Also

`AWTEventMulticaster`, `ComponentAdapter`, `ComponentEvent`, `EventListener`

PREVIOUS | HOME | NEXT
ComponentEvent ★ | BOOK INDEX | ContainerAdapter ★

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

ContainerAdapter ★

Name

ContainerAdapter ★

[Graphic: Figure from the text]

Description

The ContainerAdapter class implements the methods of ContainerListener with empty functions. It may be easier for you to extend ContainerAdapter, overriding only those methods you are interested in, than to implement ContainerListener and provide the empty functions yourself.

Class Definition

```
public abstract class java.awt.event.ContainerAdapter
   extends java.lang.Object
   implements java.awt.event.ContainerListener {
  // Instance Methods
  public void componentAdded (ContainerEvent e);
  public void componentRemoved (ContainerEvent e);
}
```

Instance Methods

componentAdded

public void componentAdded (ComponentEvent e)

Parameters

e

The event that has occurred.

Description

Does nothing. Override this function to be notified when a component is added to a container.

componentRemoved

public void componentRemoved (ComponentEvent e)

Parameters

e

The event that has occurred.

Description

Does nothing. Override this function to be notified when a component is removed from a container.

See Also

`ContainerEvent`, `ContainerListener`

ContainerEvent ★

Name

ContainerEvent ★

[Graphic: Figure from the text]

Description

Container events are fired off when a component is added to or removed from a container. The AWT automatically deals with adding components to containers; these events are provided only for notification.

Class Definition

```
public class java.awt.event.ContainerEvent
   extends java.awt.event.ComponentEvent {
  // Constants
  public final static int COMPONENT_ADDED;
  public final static int COMPONENT_REMOVED;
  public final static int CONTAINER_FIRST;
  public final static int CONTAINER_LAST;
  // Constructors
  public ContainerEvent (Component source, int id, Component child);
  // Instance Methods
  public Component getChild();
  public Container getContainer();
  public String paramString();
}
```

Constants

COMPONENT_ADDED

public final static int COMPONENT_ADDED

Event type ID indicating that a component was added to a container.

CONTAINER_FIRST

public final static int CONTAINER_FIRST

Specifies the beginning range of container event ID values.

CONTAINER_LAST

public final static int CONTAINER_LAST

Specifies the ending range of container event ID values.

COMPONENT_REMOVED

public final static int COMPONENT_REMOVED

Event type ID indicating that a component was removed from a container.

Constructors

ContainerEvent

public ContainerEvent (Component source, int id, Component child)

Parameters

source

The object that generated the event.

id

The event type ID of the event.

child

The component that was added or removed.

Description

Constructs a `ContainerEvent` with the given characteristics.

Instance Methods

getChild

public Component getChild()

Returns

The component that is being added or removed.

getContainer

public Container getContainer()

Returns

The container for this event.

paramString

public String paramString()

Returns

String with current settings of the `ContainerEvent`.

Overrides

`ComponentEvent.paramString()`

Description

Helper method for `toString()` to generate string of current settings.

See Also

`Component`, `ComponentEvent`, `Container`, `ContainerAdapter`, `ContainerListener`

ContainerListener ★

Name

ContainerListener ★

[Graphic: Figure from the text]

Description

Objects that implement the `ContainerListener` interface can receive `ContainerEvent` objects. Listeners must first register themselves with objects that produce events. When events occur, they are then automatically propagated to all registered listeners.

Interface Definition

```
public abstract interface java.awt.event.ContainerListener
   extends java.util.EventListener {
  // Instance Methods
  public abstract void componentAdded (ContainerEvent e);
  public abstract void componentRemoved (ContainerEvent e);
}
```

Interface Methods

componentAdded

public abstract void componentAdded (ContainerEvent e)

Parameters

e

The event that occurred.

Description

Notifies the `ContainerListener` that a component has been added to the container.

componentRemoved

public abstract void componentRemoved (ContainerEvent e)

Parameters

e

The event that occurred.

Description

Notifies the `ContainerListener` that a component has been removed from the container.

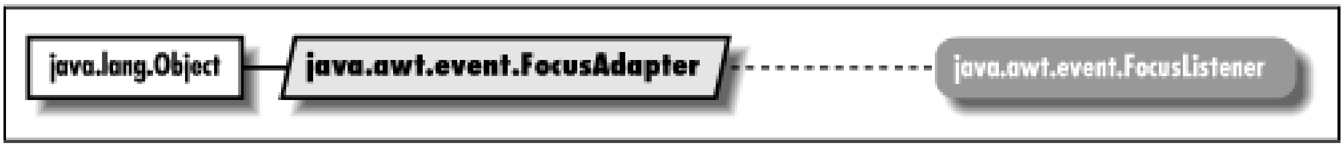
See Also

`ContainerAdapter`, `ContainerEvent`, `EventListener`

FocusAdapter ★

Name

FocusAdapter ★



Description

The `FocusAdapter` class implements the methods of `FocusListener` with empty functions. It may be easier for you to extend `FocusAdapter`, overriding only those methods you are interested in, than to implement `FocusListener` and provide the empty functions yourself.

Class Definition

```
public abstract class java.awt.event.FocusAdapter
   extends java.lang.Object
   implements java.awt.event.FocusListener {
  // Instance Methods
  public void focusGained (FocusEvent e);
  public void focusLost (FocusEvent e);
}
```

Instance Methods

focusGained

public void focusGained (FocusEvent e)

Parameters

e

The event that has occurred.

Description

Does nothing. Override this function to be notified when a component gains focus.

focusLost

public void focusLost (FocusEvent e)

Parameters

e

The event that has occurred.

Description

Does nothing. Override this function to be notified when a component loses focus.

See Also

`FocusEvent`, `FocusListener`

FocusEvent ★

Name

FocusEvent ★

[Graphic: Figure from the text]

Description

Focus events are generated when a component gets or loses input focus. Focus events come in two flavors, permanent and temporary. Permanent focus events occur with explicit focus changes. For example, when the user tabs through components, this causes permanent focus events. An example of a temporary focus event is when a component loses focus as its containing window is deactivated.

Class Definition

```
public class java.awt.event.FocusEvent
   extends java.awt.event.ComponentEvent {
  // Constants
  public final static int FOCUS_FIRST;
  public final static int FOCUS_GAINED;
  public final static int FOCUS_LAST;
  public final static int FOCUS_LOST;
  // Constructors
  public FocusEvent (Component source, int id);
  public FocusEvent (Component source, int id, boolean temporary);
  // Instance Methods
  public boolean isTemporary();
  public String paramString();
}
```

Constants

FOCUS_FIRST

public final static int FOCUS_FIRST

Specifies the beginning range of focus event ID values.

FOCUS_GAINED

public final static int FOCUS_GAINED

Event type ID indicating that the component gained the input focus.

FOCUS_LAST

public final static int FOCUS_LAST

Specifies the ending range of focus event ID values.

FOCUS_LOST

public final static int FOCUS_LOST

Event type ID indicating that the component lost the input focus.

Constructors

FocusEvent

public FocusEvent (Component source, int id)

Parameters

source

The object that generated the event.

id

The event type ID of the event.

Description

Constructs a non-temporary `FocusEvent` with the given characteristics.

public FocusEvent (Component source, int id, boolean temporary)

Parameters

source

The object that generated the event.

id

The event type ID of the event.

temporary

A flag indicating whether this is a temporary focus event.

Description

Constructs a `FocusEvent` with the given characteristics.

Instance Methods

isTemporary

public boolean isTemporary()

Returns

> `true` if this is a temporary focus event; `false` otherwise.

paramString

public String paramString()

Returns

> String with current settings of the `FocusEvent`.

Overrides

> `ComponentEvent.paramString()`

Description

> Helper method for `toString()` to generate string of current settings.

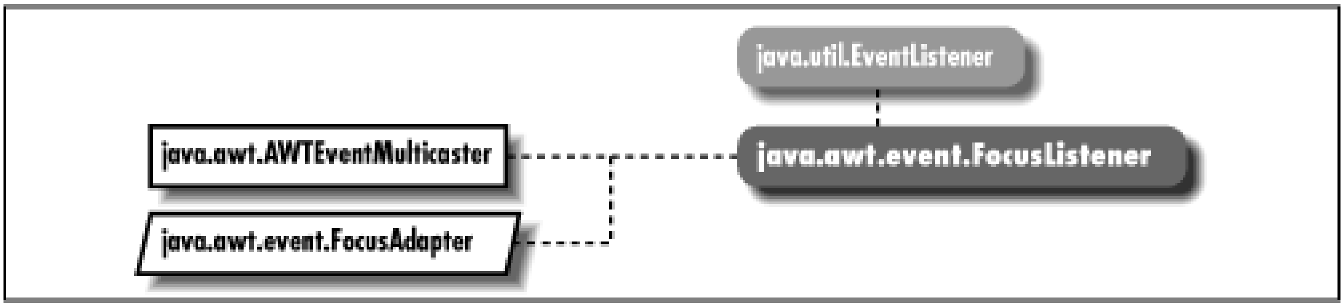
See Also

`Component`, `ComponentEvent`, `FocusAdapter`, `FocusListener`

FocusListener ★

Name

FocusListener ★


java.util.EventListener
java.awt.AWTEventMulticaster
java.awt.event.FocusListener
java.awt.event.FocusAdapter

Description

Objects that implement the `FocusListener` interface can receive `FocusEvent` objects. Listeners must first register themselves with objects that produce events. When events occur, they are then automatically propagated to all registered listeners.

Interface Definition

```
public abstract interface java.awt.event.FocusListener
   extends java.util.EventListener {
  // Instance Methods
  public abstract void focusGained (FocusEvent e);
  public abstract void focusLost (FocusEvent e);
}
```

Interface Methods

focusGained

public abstract void focusGained (FocusEvent e)

Parameters

e

The component event that occurred.

Description

Notifies the `FocusListener` that a component gained the input focus.

focusLost

public abstract void focusLost (FocusEvent e)

Parameters

e

The component event that occurred.

Description

Notifies the `FocusListener` that a component lost the input focus.

See Also

`AWTEventMulticaster`, `EventListener`, `FocusAdapter`, `FocusEvent`

PREVIOUS HOME NEXT

FocusEvent ★ BOOK INDEX InputEvent ★

InputEvent ★

Name

InputEvent ★

[Graphic: Figure from the text]

Description

`InputEvent` is the root class for representing user input events. Input events are passed to listeners before the event source processes them. If one of the listeners consumes an event by using `consume()`, the event will not be processed by the event source peer.

Class Definition

```
public abstract class java.awt.event.InputEvent
   extends java.awt.event.ComponentEvent {
  // Constants
  public final static int ALT_MASK;
  public final static int BUTTON1_MASK;
  public final static int BUTTON2_MASK;
  public final static int BUTTON3_MASK;
  public final static int CTRL_MASK;
  public final static int META_MASK;
  public final static int SHIFT_MASK;
  // Instance Methods
  public void consume();
  public int getModifiers();
  public long getWhen();
  public boolean isAltDown();
  public boolean isConsumed();
  public boolean isControlDown();
  public boolean isMetaDown();
  public boolean isShiftDown();
}
```

Constants

ALT_MASK

public final static int ALT_MASK

The ALT key mask. ORed with other masks to form modifiers setting of event.

BUTTON1_MASK

public final static int BUTTON1_MASK

The mouse button 1 key mask. ORed with other masks to form modifiers setting of event.

BUTTON2_MASK

public final static int BUTTON2_MASK

The mouse button 2 key mask. ORed with other masks to form modifiers setting of event. This constant is identical to `ALT_MASK`.

BUTTON3_MASK

public final static int BUTTON3_MASK

The mouse button 3 key mask. ORed with other masks to form modifiers setting of event. This constant is identical to `ALT_MASK`.

CTRL_MASK

public final static int CTRL_MASK

The Control key mask. ORed with other masks to form modifiers setting of event.

META_MASK

public final static int META_MASK

The Meta key mask. ORed with other masks to form modifiers setting of event.

SHIFT_MASK

public final static int SHIFT_MASK

The Shift key mask. ORed with other masks to form modifiers setting of event.

Instance Methods

consume

public void consume()

Description

> A consumed event will not be delivered to its source for default processing.

getModifiers

public int getModifiers()

Returns

The modifier flags, a combination of the `_MASK` constants.

Description

Use this method to find out what modifier keys were pressed when an input event occurred.

getWhen

public long getWhen()

Returns

The time at which this event occurred.

Description

The time of the event is returned as the number of milliseconds since the epoch (00:00:00 UTC, January 1, 1970). Conveniently, `java.util.Date` has a constructor that accepts such values.

isAltDown

public boolean isAltDown()

Returns

`true` if the Alt key was pressed; `false` otherwise.

isConsumed

public boolean isConsumed()

Returns

`true` if the event has been consumed; `false` otherwise.

isControlDown

public boolean isControlDown()

Returns

> `true` if the Control key was pressed; `false` otherwise.

isMetaDown

public boolean isMetaDown()

Returns

> `true` if the Meta key was pressed; `false` otherwise.

isShiftDown

public boolean isShiftDown()

Returns

> `true` if the Shift key was pressed; `false` otherwise.

See Also

`ComponentEvent`, `KeyEvent`, `MouseEvent`

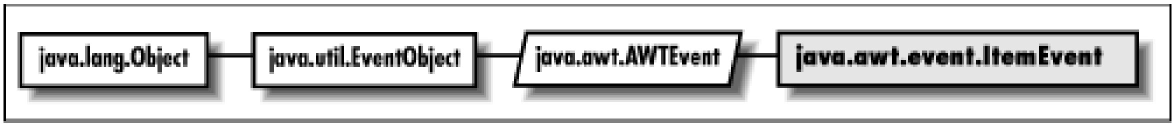
PREVIOUS HOME NEXT
FocusListener ★ BOOK INDEX ItemEvent ★

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

ItemEvent ★

Name

ItemEvent ★



Description

`ItemEvents` are generated by objects that implement the `ItemSelectable` interface. `Choice` is one example of such an object.

Class Definition

```
public class java.awt.event.ItemEvent
   extends java.awt.AWTEvent {
  // Constants
  public final static int DESELECTED;
  public final static int ITEM_FIRST;
  public final static int ITEM_LAST;
  public final static int ITEM_STATE_CHANGED;
  public final static int SELECTED;
  // Constructors
  public ItemEvent (ItemSelectable source, int id, Object item, int stateChange);

// Instance Methods
  public Object getItem();
  public ItemSelectable getItemSelectable();
  public int getStateChange();
  public String paramString();
}
```

Constants

DESELECTED

public final static int DESELECTED

Indicates that an item was deselected.

ITEM_FIRST

public final static int ITEM_FIRST

Specifies the beginning range of item event ID values.

ITEM_LAST

public final static int ITEM_LAST

Specifies the ending range of item event ID values.

ITEM_STATE_CHANGED

public final static int ITEM_STATE_CHANGED

An event type indicating that an item was selected or deselected.

SELECTED

public final static int SELECTED

Indicates that an item was selected.

Constructors

ItemEvent

public ItemEvent (ItemSelectable source, int id, Object item, int stateChange)

Parameters

source

The object that generated the event.

id

The type ID of the event.

item

The item whose state is changing.

stateChange

Either `SELECTED` or `DESELECTED`.

Description

Constructs an `ItemEvent` with the given characteristics.

Instance Methods

getItem

public Object getItem()

Returns

The item pertaining to this event.

Description

Returns the item whose changed state triggered this event.

getItemSelectable

public ItemSelectable getItemSelectable()

Returns

The source of this event.

Description

Returns an object that implements the `ItemSelectable` interface.

getStateChange

public int getStateChange()

Returns

The change in state that triggered this event. The new state is returned.

Description

This method will return SELECTED or DESELECTED.

paramString

public String paramString()

Returns

String with current settings of ItemEvent.

Overrides

AWTEvent.paramString()

Description

Helper method for toString() to generate string of current settings.

See Also

AWTEvent, ItemSelectable, ItemListener

PREVIOUS HOME NEXT
InputEvent ★ BOOK INDEX ItemListener ★

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

PREVIOUS

NEXT

ItemListener ★

Name

ItemListener ★

[Graphic: Figure from the text]

Description

Objects that implement the `ItemListener` interface can receive `ItemEvent` objects. Listeners must first register themselves with objects that produce events. When events occur, they are then automatically propagated to all registered listeners.

Interface Definition

```
public abstract interface java.awt.event.ItemListener
   extends java.util.EventListener {
  // Interface Methods
  public abstract void itemStateChanged (ItemEvent e);
}
```

Interface Methods

itemStateChanged

public abstract void itemStateChanged (ItemEvent e)

Parameters

e

The item event that occurred.

Description

Notifies the `ItemListener` that an event occurred.

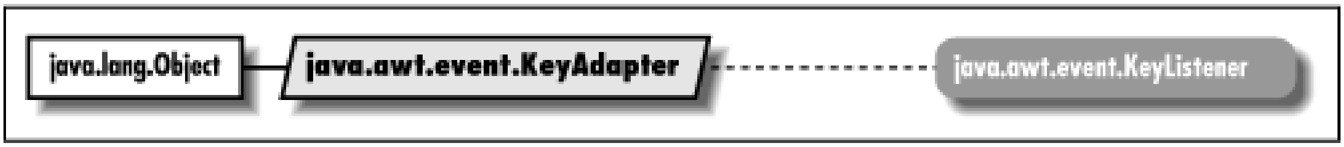
See Also

`AWTEventMultiCaster`, `EventListener`, `ItemEvent`

KeyAdapter ★

Name

KeyAdapter ★



Description

The `KeyAdapter` class implements the methods of `KeyListener` with empty functions. It may be easier for you to extend `KeyAdapter`, overriding only those methods you are interested in, than to implement `KeyListener` and provide the empty functions yourself.

Class Definition

```
public abstract class java.awt.event.KeyAdapter
   extends java.lang.Object
   implements java.awt.event.KeyListener {
  // Instance Methods
  public void keyPressed (KeyEvent e);
  public void keyReleased (KeyEvent e);
  public void keyTyped (KeyEvent e);
}
```

Instance Methods

keyPressed

public void keyPressed (KeyEvent e)

Parameters

e

The event that has occurred.

Description

Does nothing. Override this function to be notified when a key is pressed.

keyReleased

public void keyReleased (KeyEvent e)

Parameters

e

The event that has occurred.

Description

Does nothing. Override this function to be notified when a pressed key is released.

keyTyped

public void keyTyped (KeyEvent e)

Parameters

e

The event that has occurred.

Description

Does nothing. Override this function to be notified when a key has been pressed and released.

See Also

`KeyEvent`, `KeyListener`

KeyEvent ★

Name

KeyEvent ★

[Graphic: Figure from the text]

Description

Key events are generated when the user types on the keyboard.

Class Definition

```
public class java.awt.event.KeyEvent
   extends java.awt.event.InputEvent {
  // Constants
  public final static int CHAR_UNDEFINED;
  public final static int KEY_FIRST;
  public final static int KEY_LAST;
  public final static int KEY_PRESSED;
  public final static int KEY_RELEASED;
  public final static int KEY_TYPED;
  public final static int VK_0;
  public final static int VK_1;
  public final static int VK_2;
  public final static int VK_3;
  public final static int VK_4;
  public final static int VK_5;
  public final static int VK_6;
  public final static int VK_7;
  public final static int VK_8;
  public final static int VK_9;
  public final static int VK_A;
  public final static int VK_ACCEPT;
  public final static int VK_ADD;
  public final static int VK_ALT;
  public final static int VK_B;
  public final static int VK_BACK_QUOTE;
  public final static int VK_BACK_SLASH;
  public final static int VK_BACK_SPACE;
  public final static int VK_C;
  public final static int VK_CANCEL;
  public final static int VK_CAPS_LOCK;
  public final static int VK_CLEAR;
  public final static int VK_CLOSE_BRACKET;
  public final static int VK_COMMA;
  public final static int VK_CONTROL;
  public final static int VK_CONVERT;
  public final static int VK_D;
  public final static int VK_DECIMAL;
  public final static int VK_DELETE;
  public final static int VK_DIVIDE;
  public final static int VK_DOWN;
  public final static int VK_E;
  public final static int VK_END;
  public final static int VK_ENTER;
  public final static int VK_EQUALS;
  public final static int VK_ESCAPE;
  public final static int VK_F;
  public final static int VK_F1;
  public final static int VK_F2;
  public final static int VK_F3;
  public final static int VK_F4;
  public final static int VK_F5;
  public final static int VK_F6;
  public final static int VK_F7;
  public final static int VK_F8;
  public final static int VK_F9;
  public final static int VK_F10;
  public final static int VK_F11;
  public final static int VK_F12;
  public final static int VK_FINAL;
  public final static int VK_G;
  public final static int VK_H;
  public final static int VK_HELP;
  public final static int VK_HOME;
  public final static int VK_I;
  public final static int VK_INSERT;
  public final static int VK_J;
  public final static int VK_K;
  public final static int VK_KANA;
  public final static int VK_KANJI;
  public final static int VK_L;
  public final static int VK_LEFT;
  public final static int VK_M;
  public final static int VK_META;
  public final static int VK_MODECHANGE;
  public final static int VK_MULTIPLY;
  public final static int VK_N;
  public final static int VK_NONCONVERT;
  public final static int VK_NUM_LOCK;
  public final static int VK_NUMPAD0;
  public final static int VK_NUMPAD1;
  public final static int VK_NUMPAD2;
  public final static int VK_NUMPAD3;
  public final static int VK_NUMPAD4;
  public final static int VK_NUMPAD5;
  public final static int VK_NUMPAD6;
  public final static int VK_NUMPAD7;
  public final static int VK_NUMPAD8;
  public final static int VK_NUMPAD9;
  public final static int VK_O;
  public final static int VK_OPEN_BRACKET;
  public final static int VK_P;
  public final static int VK_PAGE_DOWN;
  public final static int VK_PAGE_UP;
  public final static int VK_PAUSE;
  public final static int VK_PERIOD;
  public final static int VK_PRINTSCREEN;
```

```
  public final static int VK_Q;
  public final static int VK_QUOTE;
  public final static int VK_R;
  public final static int VK_RIGHT;
  public final static int VK_S;
  public final static int VK_SCROLL_LOCK;
  public final static int VK_SEMICOLON;
  public final static int VK_SEPARATER;
  public final static int VK_SHIFT;
  public final static int VK_SLASH;
  public final static int VK_SPACE;
  public final static int VK_SUBTRACT;
  public final static int VK_T;
  public final static int VK_TAB;
  public final static int VK_U;
  public final static int VK_UNDEFINED;
  public final static int VK_UP;
  public final static int VK_V;
  public final static int VK_W;
  public final static int VK_X;
  public final static int VK_Y;
  public final static int VK_Z;
  // Constructors
  public KeyEvent (Component source, int id, long when, int modifiers,
    int keyCode, char keyChar);
  // Class Methods
  public static String getKeyModifiersText(int modifiers);
  public static String getKeyText(int keyCode);
  // Instance Methods
  public char getKeyChar();
  public int getKeyCode();
  public boolean isActionKey();
  public String paramString();
  public void setKeyChar (char keyChar);
  public void setKeyCode (int keyCode);
  public void setModifiers (int modifiers);
}
```

Constants

CHAR_UNDEFINED

public final static int CHAR_UNDEFINED

This constant is used for key presses have that no associated character.

KEY_FIRST

public final static int KEY_FIRST

Specifies the beginning range of key event ID values.

KEY_LAST

public final static int KEY_LAST

Specifies the ending range of key event ID values.

KEY_PRESSED

public final static int KEY_PRESSED

An event ID type for a key press.

KEY_RELEASED

public final static int KEY_RELEASED

An event ID type for a key release.

KEY_TYPED

public final static int KEY_TYPED

An event ID type for a typed key (a press and a release).

VK_0

public final static int VK_0

The 0 key.

VK_1

public final static int VK_1

The 1 key.

VK_2

public final static int VK_2

The 2 key.

VK_3

public final static int VK_3

The 3 key.

VK_4

public final static int VK_4

The 4 key.

VK_5

public final static int VK_5

The 5 key.

VK_6

public final static int VK_6

The 6 key.

VK_7

public final static int VK_7

The 7 key.

VK_8

public final static int VK_8

The 8 key.

VK_9

public final static int VK_9

The 9 key.

VK_A

public final static int VK_A

The `a' key.

VK_ACCEPT

public final static int VK_ACCEPT

This constant is used for Asian keyboards.

VK_ADD

public final static int VK_ADD

The plus (+) key on the numeric keypad.

VK_ALT

public final static int VK_ALT

The Alt key.

VK_B

public final static int VK_B

The `b' key.

VK_BACK_QUOTE

public final static int VK_BACK_QUOTE

The backquote (`) key.

VK_BACK_SLASH

public final static int VK_BACK_SLASH

The backslash key.

VK_BACK_SPACE

public final static int VK_BACK_SPACE

The Backspace key.

VK_C

public final static int VK_C

The `c' key.

VK_CANCEL

public final static int VK_CANCEL

The Cancel key.

VK_CAPS_LOCK

public final static int VK_CAPS_LOCK

The Caps Lock key.

VK_CLEAR

public final static int VK_CLEAR

The Clear key.

VK_CLOSE_BRACKET

public final static int VK_CLOSE_BRACKET

The close bracket `]' key.

VK_COMMA

public final static int VK_COMMA

The comma (,) key.

VK_CONTROL

public final static int VK_CONTROL

The Control key.

VK_CONVERT

public final static int VK_CONVERT

This constant is used for Asian keyboards.

VK_D

public final static int VK_D

The `d' key.

VK_DECIMAL

public final static int VK_DECIMAL

The decimal (.) key on the numeric keypad.

VK_DELETE

public final static int VK_DELETE

The Delete key.

VK_DIVIDE

public final static int VK_DIVIDE

The divide (/) key on the numeric keypad.

VK_DOWN

public final static int VK_DOWN

The Down arrow key.

VK_E

public final static int VK_E

The `e' key.

VK_END

public final static int VK_END

The End key.

VK_ENTER

public final static int VK_ENTER

The Enter key.

VK_EQUALS

public final static int VK_ EQUALS

The equals (=) key.

VK_ESCAPE

public final static int VK_ESCAPE

The Escape key.

VK_F

public final static int VK_F

The `f' key.

VK_F1

public final static int VK_F1

The F1 key.

VK_F2

public final static int VK_F2

The F2 key.

VK_F3

public final static int VK_F3

The F3 key.

VK_F4

public final static int VK_F4

The F4 key.

VK_F5

public final static int VK_F5

The F5 key.

VK_F6

public final static int VK_F6

The F6 key.

VK_F7

public final static int VK_F7

The F7 key.

VK_F8

public final static int VK_F8

The F8 key.

VK_F9

public final static int VK_F9

The F9 key.

VK_F10

public final static int VK_F10

The F10 key.

VK_F11

public final static int VK_F11

The F11 key.

VK_F12

public final static int VK_F12

The F12 key.

VK_FINAL

public final static int VK_FINAL

This constant is used for Asian keyboards.

VK_G

public final static int VK_G

The `g' key.

VK_H

public final static int VK_H

The `h' key.

VK_HELP

public final static int VK_HELP

The Help key.

VK_HOME

public final static int VK_HOME

The Home key.

VK_I

public final static int VK_I

The `i' key.

VK_INSERT

public final static int VK_INSERT

The Insert key.

VK_J

public final static int VK_J

The `j' key.

VK_K

public final static int VK_K

The `k' key.

VK_KANA

public final static int VK_KANA

This constant is used for Asian keyboards.

VK_KANJI

public final static int VK_KANJI

This constant is used for Asian keyboards.

VK_L

public final static int VK_L

The `l' key.

VK_LEFT

public final static int VK_LEFT

The Left arrow key.

VK_M

public final static int VK_M

The `m' key.

VK_MODECHANGE

public final static int VK_MODECHANGE

This constant is used for Asian keyboards.

VK_META

public final static int VK_META

The Meta key.

VK_MULTIPLY

public final static int VK_MULTIPLY

The * key on the numeric keypad.

VK_N

public final static int VK_N

The `n' key.

VK_NONCONVERT

public final static int VK_NONCONVERT

This constant is used for Asian keyboards.

VK_NUM_LOCK

public final static int VK_NUM_LOCK

The Num Lock key.

VK_NUMPAD0

public final static int VK_NUMPAD0

The 0 key on the numeric keypad.

VK_NUMPAD1

public final static int VK_NUMPAD1

The 1 key on the numeric keypad.

VK_NUMPAD2

public final static int VK_NUMPAD2

The 2 key on the numeric keypad.

VK_NUMPAD3

public final static int VK_NUMPAD3

The 3 key on the numeric keypad.

VK_NUMPAD4

public final static int VK_NUMPAD4

The 4 key on the numeric keypad.

VK_NUMPAD5

public final static int VK_NUMPAD5

The 5 key on the numeric keypad.

VK_NUMPAD6

public final static int VK_NUMPAD6

The 6 key on the numeric keypad.

VK_NUMPAD7

public final static int VK_NUMPAD7

The 7 key on the numeric keypad.

VK_NUMPAD8

public final static int VK_NUMPAD8

The 8 key on the numeric keypad.

VK_NUMPAD9

public final static int VK_NUMPAD9

The 9 key on the numeric keypad.

VK_O

public final static int VK_O

The `o' key.

VK_OPEN_BRACKET

public final static int VK_OPEN_BRACKET

The open bracket `[` key.

VK_P

public final static int VK_P

The `p' key.

VK_PAGE_DOWN

public final static int VK_PAGE_DOWN

The Page Down key.

VK_PAGE_UP

public final static int VK_PAGE_UP

The Page Up key.

VK_PAUSE

public final static int VK_PAUSE

The Pause key.

VK_PERIOD

public final static int VK_PERIOD

The period (.) key.

VK_PRINTSCREEN

public final static int VK_PRINTSCREEN

The Print Screen key.

VK_Q

public final static int VK_Q

The `q' key.

VK_QUOTE

public final static int VK_QUOTE

The quotation mark (") key.

VK_R

public final static int VK_R

The `r' key.

VK_RIGHT

public final static int VK_RIGHT

The Right arrow key.

VK_S

public final static int VK_S

The `s' key.

VK_SCROLL_LOCK

public final static int VK_SCROLL_LOCK

The Scroll Lock key.

VK_SEMICOLON

public final static int VK_SEMICOLON

The semicolon (;) key.

VK_SEPARATER

public final static int VK_SEPARATER

The numeric separator key on the numeric keypad (i.e., the locale-dependent key used to separate groups of digits). A misspelling of VK_SEPARATOR.

VK_SHIFT

public final static int VK_SHIFT

The Shift key.

VK_SLASH

public final static int VK_SLASH

The slash (/) key.

VK_SPACE

public final static int VK_SPACE

The space key.

VK_SUBTRACT

public final static int VK_SUBTRACT

The subtract (-) key on the numeric keypad.

VK_T

public final static int VK_T

The `t' key.

VK_TAB

public final static int VK_TAB

The Tab key.

VK_U

public final static int VK_U

The `u' key.

VK_UNDEFINED

public final static int VK_UNDEFINED

An undefined key.

VK_UP

public final static int VK_UP

The Up arrow key.

VK_V

public final static int VK_V

The `v' key.

VK_W

public final static int VK_W

The `w' key.

VK_X

public final static int VK_X

The `x' key.

VK_Y

public final static int VK_Y

The `y' key.

VK_Z

public final static int VK_Z

The `z' key.

Constructors

KeyEvent

public KeyEvent (Component source, int id, long when, int modifiers, int keyCode, char keyChar)

Parameters

> *source*
>
> > The object that generated the event.

id

The event type ID of the event.

when

When the event occurred, in milliseconds from the epoch.

modifiers

What modifier keys were pressed with this key.

keyCode

The code of the key.

keyChar

The character for this key.

Description

Constructs a `KeyEvent` with the given characteristics.

Class Methods

getKeyModifiersText

public static String getKeyModifiersText(int modifiers)

Parameters

modifiers

One or more modifier keys.

Returns

A string describing the modifiers.

getKeyText

public static String getKeyText(int keyCode)

Parameters

keyCode

One of the key codes.

Returns

A string describing the given key.

Instance Methods

getKeyChar

public char getKeyChar()

Returns

The character corresponding to this event. KEY_TYPED events have characters.

getKeyCode

public int getKeyCode()

Returns

The integer key code corresponding to this event. This will be one of the constants defined above. KEY_PRESSED and KEY_RELEASED events have codes. Key codes are virtual keys, not actual. Pressing the `a' key is identical to `A', but has different modifiers. Same for `/' and `?' on a standard keyboard.

isActionKey

public boolean isActionKey()

Returns

`true` if this event is for one of the action keys; `false` otherwise.

Description

In general, an action key is a key that causes an action but has no printing equivalent. The action keys are the function keys, the arrow keys, Caps Lock, End, Home, Insert, Num Lock, Pause, Page Down, Page Up, Print Screen, and Scroll Lock. They do not generate a KEY_TYPED event, only KEY_PRESSED and KEY_RELEASED.

paramString

public String paramString()

Returns

A string with current settings of the `KeyEvent`.

Overrides

`ComponentEvent.paramString()`

Description

Helper method for `toString()` to generate string of current settings.

setKeyChar

public void setKeyChar(char keyChar)

Parameters

keyChar

The new key character.

Description

Sets the character code of this `KeyEvent`.

setKeyCode

public void setKeyCode (int keyCode)

Parameters

keyCode

The new key code.

Description

Sets the key code of this `KeyEvent`.

setModifiers

public void setModifiers (int modifiers)

Parameters

modifiers

The new modifiers.

Description

This is a combination of the mask constants defined in `java.awt.event.InputEvent`.

See Also

`Component`, `ComponentEvent`, `InputEvent`, `KeyAdapter`, `KeyListener`

PREVIOUS | HOME | NEXT

KeyAdapter ★ | BOOK INDEX | KeyListener ★

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

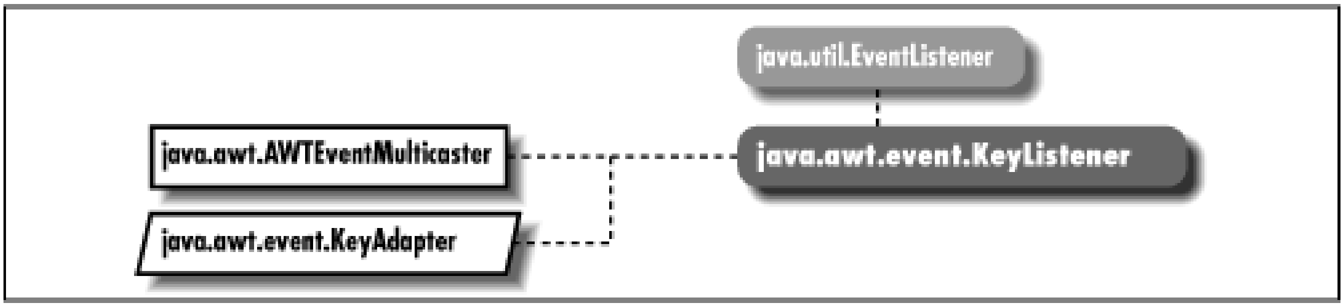
PREVIOUS

NEXT

KeyListener ★

Name

KeyListener ★


java.util.EventListener
java.awt.AWTEventMulticaster
java.awt.event.KeyListener
java.awt.event.KeyAdapter

Description

Objects that implement the `KeyListener` interface can receive `KeyEvent` objects. Listeners must first register themselves with objects that produce events. When events occur, they are then automatically propagated to all registered listeners.

Interface Definition

```
public abstract interface java.awt.event.KeyListener
   extends java.util.EventListener {
  // Instance Methods
  public abstract void keyPressed (KeyEvent e);
  public abstract void keyReleased (KeyEvent e);
  public abstract void keyTyped (KeyEvent e);
}
```

Interface Methods

keyPressed

public abstract void keyPressed (KeyEvent e)

Parameters

e

The key event that occurred.

Description

Notifies the `KeyListener` that a key was pressed.

keyReleased

public abstract void keyReleased (KeyEvent e)

Parameters

e

The key event that occurred.

Description

Notifies the `KeyListener` that a key was released.

keyTyped

public abstract void keyTyped (KeyEvent e)

Parameters

e

The key event that occurred.

Description

Notifies the `KeyListener` that a key was typed (pressed and released).

See Also

`AWTEventMulticaster`, `EventListener`, `KeyEvent`, `KeyListener`

MouseAdapter ★

Name

MouseAdapter ★

[Graphic: Figure from the text]

Description

The `MouseAdapter` class implements the methods of `MouseListener` with empty functions. It may be easier for you to extend `MouseAdapter`, overriding only those methods you are interested in, than to implement `MouseListener` and provide the empty functions yourself.

Class Definition

```
public abstract class java.awt.event.MouseAdapter
   extends java.lang.Object
   implements java.awt.event.MouseListener {
  // Instance Methods
  public void mouseClicked (MouseEvent e);
  public void mouseEntered (MouseEvent e);
  public void mouseExited (MouseEvent e);
  public void mousePressed (MouseEvent e);
  public void mouseReleased (MouseEvent e);
}
```

Instance Methods

mouseClicked

public void mouseClicked (MouseEvent e)

Parameters

e

The event that has occurred.

Description

Does nothing. Override this function to be notified when the mouse button is clicked (pressed and released).

mouseEntered

public void mouseEntered (MouseEvent e)

Parameters

e

The event that has occurred.

Description

Does nothing. Override this function to be notified when the user moves the mouse cursor into a component.

mouseExited

public void mouseExited (MouseEvent e)

Parameters

e

The event that has occurred.

Description

Does nothing. Override this function to be notified when the moves the mouse cursor out of a component.

mousePressed

public void mousePressed (MouseEvent e)

Parameters

e

The event that has occurred.

Description

Does nothing. Override this function to be notified when the mouse button is pressed.

mouseReleased

public void mouseReleased (MouseEvent e)

Parameters

e

The event that has occurred.

Description

Does nothing. Override this function to be notified when the mouse button is released.

See Also

`MouseEvent`, `MouseListener`

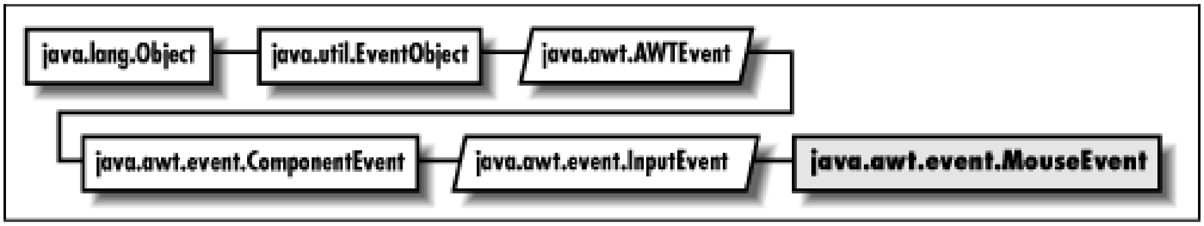
PREVIOUS | HOME | NEXT
KeyListener ★ | BOOK INDEX | MouseEvent ★

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

MouseEvent ★

Name

MouseEvent ★


java.lang.Object
java.util.EventObject
java.awt.AWTEvent
java.awt.event.ComponentEvent
java.awt.event.InputEvent
java.awt.event.MouseEvent

Description

Mouse events are generated when the user moves and clicks the mouse.

Class Definition

```
public class java.awt.event.MouseEvent
   extends java.awt.event.InputEvent {
  // Constants
  public final static int MOUSE_CLICKED;
  public final static int MOUSE_DRAGGED;
  public final static int MOUSE_ENTERED;
  public final static int MOUSE_EXITED;
  public final static int MOUSE_FIRST;
  public final static int MOUSE_LAST;
  public final static int MOUSE_MOVED;
  public final static int MOUSE_PRESSED;
  public final static int MOUSE_RELEASED;
  // Constructors
  public MouseEvent (Component source, int id, long when, int modifiers, int x,
    int y, int clickCount, boolean popupTrigger);
  // Instance Methods
  public int getClickCount();
  public synchronized Point getPoint();
  public int getX();
  public int getY();
  public boolean isPopupTrigger();
  public String paramString();
  public synchronized void translatePoint (int x, int y);
}
```

Constants

MOUSE_CLICKED

public final static int MOUSE_CLICKED

An event type ID indicating a mouse click.

MOUSE_DRAGGED

public final static int MOUSE_DRAGGED

An event type ID indicating a mouse move with the button held down.

MOUSE_ENTERED

public final static int MOUSE_ENTERED

An event type ID indicating that a mouse entered a component.

MOUSE_EXITED

public final static int MOUSE_EXITED

An event type ID indicating that a mouse left a component.

MOUSE_FIRST

public final static int MOUSE_FIRST

Specifies the beginning range of mouse event ID values.

MOUSE_LAST

public final static int MOUSE_LAST

Specifies the ending range of mouse event ID values.

MOUSE_MOVED

public final static int MOUSE_MOVED

An event type ID indicating a mouse move.

MOUSE_PRESSED

public final static int MOUSE_PRESSED

An event type ID indicating a mouse button press.

MOUSE_RELEASED

public final static int MOUSE_RELEASED

An event type ID indicating a mouse button release.

Constructors

MouseEvent

public MouseEvent (Component source, int id, long when, int modifiers, int x, int y, int clickCount, boolean popupTrigger)

Parameters

source

The object that generated the event.

id

The event type ID of the event.

when

When the event occurred, in milliseconds from the epoch.

modifiers

What modifier keys were pressed with this key.

x

The horizontal location of the event.

y

The vertical location of the event.

clickCount

The number of times the mouse button has been clicked.

popupTrigger

A flag indicating if this event is a popup trigger event.

Description

Constructs a `MouseEvent` with the given characteristics.

Instance Methods

getClickCount

public int getClickCount()

Returns

The number of consecutive mouse button clicks for this event.

getPoint

public synchronized Point getPoint()

Returns

The location where the event happened.

getX

public int getX()

Returns

The horizontal location where the event happened.

getY

public int getY()

Returns

The vertical location where the event happened.

isPopupTrigger

public boolean isPopupTrigger()

Returns

Returns `true` if this event is the popup menu event for the run-time system.

paramString

public String paramString()

Returns

String with current settings of the `MouseEvent`.

Overrides

`ComponentEvent.paramString()`

Description

Helper method for `toString()` to generate string of current settings.

translatePoint

public synchronized void translatePoint (int x, int y)

Parameters

x

The horizontal amount of translation.

y

The vertical amount of translation.

Description

Translates the location of the event by the given amounts.

See Also

`Component`, `ComponentEvent`, `InputEvent`, `MouseAdapter`, `MouseListener`, `Point`

PREVIOUS | HOME | NEXT
MouseAdapter ★ | BOOK INDEX | MouseListener ★

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

MouseListener ★

Name

MouseListener ★

[Graphic: Figure from the text]

Description

Objects that implement the `MouseListener` interface can receive non-motion oriented `MouseEvent` objects. Listeners must first register themselves with objects that produce events. When events occur, they are then automatically propagated to all registered listeners.

Interface Definition

```
public abstract interface java.awt.event.MouseListener
   extends java.util.EventListener {
  // Instance Methods
  public abstract void mouseClicked (MouseEvent e);
  public abstract void mouseEntered (MouseEvent e);
  public abstract void mouseExited (MouseEvent e);
  public abstract void mousePressed (MouseEvent e);
  public abstract void mouseReleased (MouseEvent e);
}
```

Interface Methods

mouseClicked

public abstract void mouseClicked (MouseEvent e)

Parameters

e

The key event that occurred.

Description

Notifies the `MouseListener` that the mouse button was clicked (pressed and released).

mouseEntered

public abstract void mouseEntered (MouseEvent e)

Parameters

e

The key event that occurred.

Description

Notifies the `MouseListener` that the mouse cursor has been moved into a component's coordinate space.

mouseExited

public abstract void mouseExited (MouseEvent e)

Parameters

e

The key event that occurred.

Description

Notifies the `MouseListener` that the mouse cursor has been moved out of a component's coordinate space.

mousePressed

public abstract void mousePressed (MouseEvent e)

Parameters

e

The key event that occurred.

Description

Notifies the `MouseListener` that the mouse button was pressed.

mouseReleased

public abstract void mouseReleased (MouseEvent e)

Parameters

e

The key event that occurred.

Description

Notifies the `MouseListener` that the mouse button was released.

See Also

`EventListener`, `MouseAdapter`, `MouseEvent`

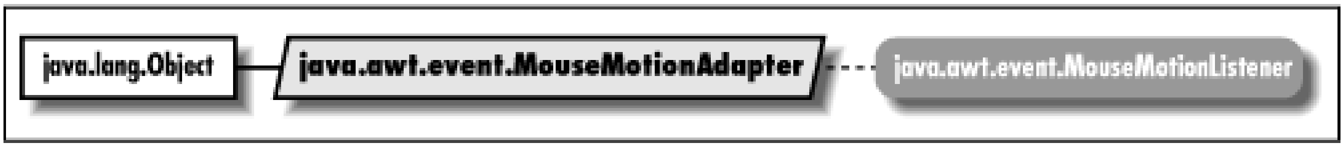
PREVIOUS HOME NEXT
MouseEvent ★ BOOK INDEX MouseMotionAdapter ★

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

MouseMotionAdapter ★

Name

MouseMotionAdapter ★


java.lang.Object
java.awt.event.MouseMotionAdapter
java.awt.event.MouseMotionListener

Description

The `MouseMotionAdapter` class implements the methods of `MouseMotionListener` with empty functions. It may be easier for you to extend `MouseMotionAdapter`, overriding only those methods you are interested in, than to implement `MouseMotionListener` and provide the empty functions yourself.

Class Definition

```
public abstract class java.awt.event.MouseMotionAdapter
   extends java.lang.Object
   implements java.awt.event.MouseMotionListener {
  // Instance Methods
  public void mouseDragged (MouseEvent e);
  public void mouseMoved (MouseEvent e);
}
```

Instance Methods

mouseDragged

public void mouseDragged (MouseEvent e)

Parameters

e

The event that has occurred.

Description

Does nothing. Override this function to be notified when the mouse is dragged.

mouseMoved

public void mouseEntered (MouseEvent e)

Parameters

e

The event that has occurred.

Description

Does nothing. Override this function to be notified when the mouse moves.

See Also

`MouseEvent`, `MouseMotionListener`

MouseMotionListener ★

Name

MouseMotionListener ★

[Graphic: Figure from the text]

Description

Objects that implement the `MouseMotionListener` interface can receive motion-oriented `MouseEvent` objects. Listeners must first register themselves with objects that produce events. When events occur, they are automatically propagated to all registered listeners.

Interface Definition

```
public abstract interface java.awt.event.MouseMotionListener
   extends java.util.EventListener {
  // Instance Methods
  public abstract void mouseDragged (MouseEvent e);
  public abstract void mouseMoved (MouseEvent e);
}
```

Interface Methods

mouseDragged

public abstract void mouseDragged (MouseEvent e)

Parameters

e

The key event that occurred.

Description

Notifies the `MouseMotionListener` that the mouse has been dragged.

mouseMoved

public abstract void mouseMoved (MouseEvent e)

Parameters

e

The key event that occurred.

Description

Notifies the `MouseMotionListener` that the mouse has been moved.

See Also

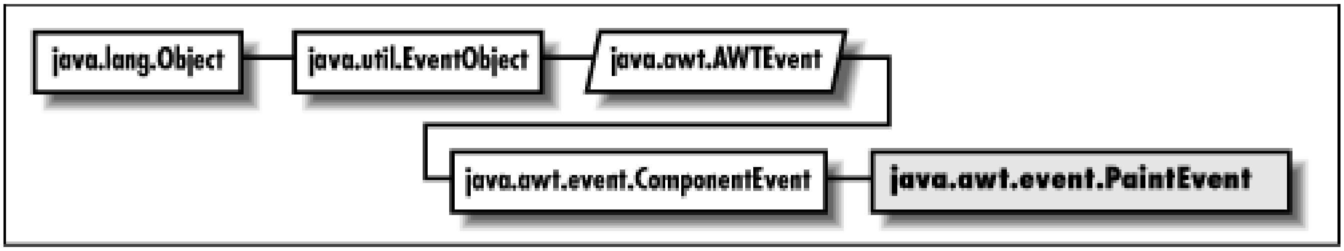
`AWTEventMulticaster`, `EventListener`, `MouseEvent`, `MouseMotionAdapter`

PREVIOUS | HOME | NEXT
MouseMotionAdapter ★ | BOOK INDEX | PaintEvent ★

PaintEvent ★

Name

PaintEvent ★



Description

The `PaintEvent` class represents the paint and update operations that the AWT performs on components. There is no `PaintListener` interface, so the only way to catch these events is to override `paint(Graphics)` and `update(Graphics)` in `Component`. This class exists so that paint events will get serialized properly.

Class Definition

```
public class java.awt.event.PaintEvent
   extends java.awt.event.ComponentEvent {
  // Constants
  public final static int PAINT;
  public final static int PAINT_FIRST;
  public final static int PAINT_LAST;
  public final static int UPDATE;
  // Constructor
  public PaintEvent (Component source, int id, Rectangle updateRect);
  // Instance Methods
  public Rectangle getUpdateRect();
  public String paramString();
  public void setUpdateRect (Rectangle updateRect);
}
```

Class Definition

```
public class java.awt.event.PaintEvent
        extends java.awt.event.ComponentEvent {
  // Constants
  public final static int PAINT;
  public final static int PAINT_FIRST;
  public final static int PAINT_LAST;
  public final static int UPDATE;
  //Constructor
  public PaintEvent (Component source, int id, Rectangle updateRect);
  // Instance Methods
  public Rectangle getUpdateRect();
  public String paramString();
  public void setUpdateRect (Rectangle updateRect);
}
```

Constants

PAINT

public final static int PAINT

The paint event type.

PAINT_FIRST

public final static int PAINT_FIRST

Specifies the beginning range of paint event ID values.

PAINT_LAST

public final static int PAINT_LAST

Specifies the ending range of paint event ID values.

UPDATE

public final static int UPDATE

The update event type.

Constructor

PaintEvent

public PaintEvent (Component source, ind id, Rectangle updateRect)

Parameters

> *source*
>
> > The source of the event.
>
> *id*
>
> > The event type ID.
>
> *g*
>
> > The rectangular area to paint.

Description

> Constructs a `PaintEvent` with the given characteristics.

Instance Methods

getUpdateRect

public Rectangle getUpdateRect()

Returns

> The rectangular area that needs painting.

paramString

public String paramString()

Returns

> String with current settings of the `PaintEvent`.

Overrides

> `ComponentEvent.paramString()`

Description

> Helper method for `toString()` to generate string of current settings.

setUpdateRect

public void setUpdateRect (Rectangle updateRect)

Parameters

> *updateRect*
>
> > The rectangular area to paint.

Description

> Changes the rectangular area that this `PaintEvent` will paint.

See Also

`Component`, `ComponentEvent`, `Graphics`

PREVIOUS HOME NEXT

MouseMotionListener ★ BOOK INDEX TextEvent ★

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

TextEvent ★

Name

TextEvent ★

[Graphic: Figure from the text]

Description

Text events are generated by text components when their contents change, either programmatically or by a user typing.

Class Definition

```
public class java.awt.event.TextEvent
   extends java.awt.AWTEvent {
  // Constants
  public final static int TEXT_FIRST;
  public final static int TEXT_LAST;
  public final static int TEXT_VALUE_CHANGED;
  // Constructors
  public TextEvent (Object source, int id);
  // Instance Methods
  public String paramString();
}
```

Constants

TEXT_FIRST

public final static int TEXT_FIRST

Specifies the beginning range of text event ID values.

TEXT_LAST

public final static int TEXT_LAST

Specifies the ending range of text event ID values.

TEXT_VALUE_CHANGED

public final static int TEXT_VALUE_CHANGED

The only text event type; it indicates that the contents of something have changed.

Constructors

TextEvent

public TextEvent (Object source, int id)

Parameters

source

The object that generated the event.

id

The type ID of the event.

Description

Constructs a `TextEvent` with the given characteristics.

Instance Methods

paramString

public String paramString()

Returns

String with current settings of the `TextEvent`.

Overrides

`AWTEvent.paramString()`

Description

Helper method for `toString()` to generate string of current settings.

See Also

`AWTEvent`, `TextListener`

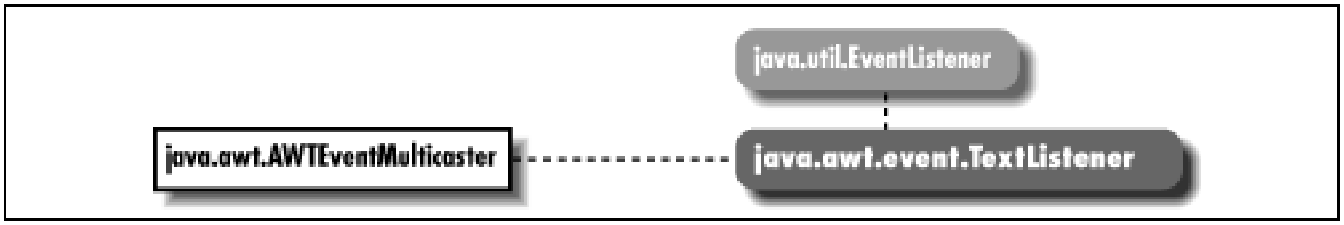
PREVIOUS | HOME | NEXT
PaintEvent ★ | BOOK INDEX | TextListener ★

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

TextListener ★

Name

TextListener ★



Description

Objects that implement the `TextListener` interface can receive `TextEvent` objects. Listeners must first register themselves with objects that produce events. When events occur, they are then automatically propagated to all registered listeners.

Interface Definition

```
public abstract interface java.awt.event.TextListener
   extends java.util.EventListener {
  // Interface Methods
  public abstract void textValueChanged (TextEvent e);
}
```

Interface Methods

textValueChanged

public abstract void textValueChanged (TextEvent e)

Parameters

e

The text event that occurred.

Description

Notifies the `TextListener` that an event occurred.

See Also

`AWTEventMulticaster`, `EventListener`, `TextEvent`

PREVIOUS HOME NEXT
TextEvent BOOK INDEX WindowAdapter

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

WindowAdapter ★

Name

WindowAdapter ★

[Graphic: Figure from the text]

Description

The `WindowAdapter` class implements the methods of `WindowListener` with empty functions. It may be easier for you to extend `WindowAdapter`, overriding only those methods you are interested in, than to implement `WindowListener` and provide the empty functions yourself.

Class Definition

```
public abstract class java.awt.event.WindowAdapter
   extends java.lang.Object
   implements java.awt.event.WindowListener {
  // Instance Methods
  public void windowActivated (WindowEvent e);
  public void windowClosed (WindowEvent e);
  public void windowClosing (WindowEvent e);
  public void windowDeactivated (WindowEvent e);
  public void windowDeiconified (WindowEvent e);
  public void windowIconified (WindowEvent e);
  public void windowOpened (WindowEvent e);
}
```

Instance Methods

windowActivated

public void windowActivated (WindowEvent e)

Parameters

e

The event that has occurred.

Description

Does nothing. Override this function to be notified when a window is activated.

windowClosed

public void windowClosed (WindowEvent e)

Parameters

e

The event that has occurred.

Description

Does nothing. Override this function to be notified when a window is closed.

windowClosing

public void windowClosing (WindowEvent e)

Parameters

e

The event that has occurred.

Description

Does nothing. Override this function to be notified when a window is in the process of closing.

windowDeactivated

public void windowDeactivated (WindowEvent e)

Parameters

e

The event that has occurred.

Description

Does nothing. Override this function to be notified when a window is deactivated.

windowDeiconified

public void windowDeiconified (WindowEvent e)

Parameters

e

The event that has occurred.

Description

Does nothing. Override this function to be notified when an iconified window is restored.

windowIconified

public void windowIconified (WindowEvent e)

Parameters

e

The event that has occurred.

Description

Does nothing. Override this function to be notified when a window is iconified (minimized).

windowOpened

public void windowOpened (WindowEvent e)

Parameters

e

The event that has occurred.

Description

Does nothing. Override this function to be notified when a window is opened.

See Also

`WindowEvent`, `WindowListener`

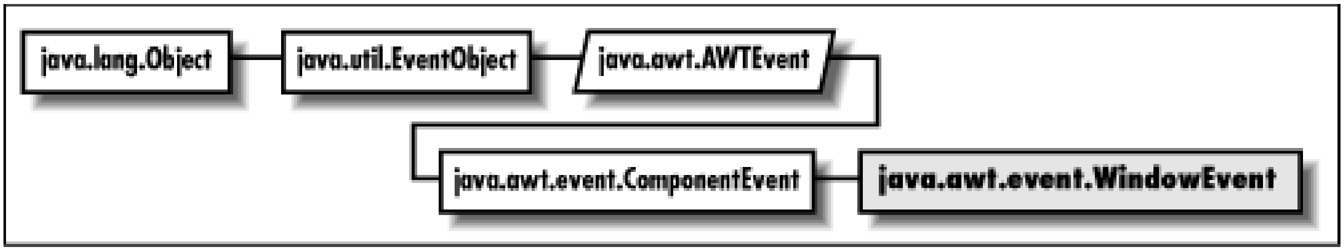
PREVIOUS HOME NEXT
TextListener ★ BOOK INDEX WindowEvent ★

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

WindowEvent ★

Name

WindowEvent ★



Description

Window events are generated when a window is opened, closed, iconified, or deiconified.

Class Definition

```
public class java.awt.event.WindowEvent
   extends java.awt.event.ComponentEvent {
  // Constants
  public final static int WINDOW_ACTIVATED;
  public final static int WINDOW_CLOSED;
  public final static int WINDOW_CLOSING;
  public final static int WINDOW_DEACTIVATED;
  public final static int WINDOW_DEICONIFIED;
  public final static int WINDOW_FIRST;
  public final static int WINDOW_ICONIFIED;
  public final static int WINDOW_LAST;
  public final static int WINDOW_OPENED;
  // Constructors
  public WindowEvent (Window source, int id);
  // Instance Methods
  public Window getWindow();
  public String paramString();
}
```

Constants

WINDOW_ACTIVATED

public final static int WINDOW_ACTIVATED

Event type ID indicating the window has been activated, brought to the foreground.

WINDOW_CLOSED

public final static int WINDOW_CLOSED

Event type ID indicating the window has closed.

WINDOW_CLOSING

public final static int WINDOW_CLOSING

Event type ID indicating the window is closing.

WINDOW_DEACTIVATED

public final static int WINDOW_DEACTIVATED

Event type ID indicating the window has been deactivated, placed in the background.

WINDOW_DEICONIFIED

public final static int WINDOW_DEICONIFIED

Event type ID indicating the window has been restored from an iconified state.

WINDOW_FIRST

public final static int WINDOW_FIRST

Specifies the beginning range of window event ID values.

WINDOW_ICONIFIED

public final static int WINDOW_ICONIFIED

Event type ID indicating the window has been iconified (minimized).

WINDOW_LAST

public final static int WINDOW_LAST

Specifies the ending range of window event ID values.

WINDOW_OPENED

public final static int WINDOW_OPENED

Event type ID indicating the window has opened.

Constructors

WindowEvent

public WindowEvent (Window source, int id)

Parameters

source

The object that generated the event.

id

The event type ID of the event.

Description

Constructs a `WindowEvent` with the given characteristics.

Instance Methods

getWindow

public Window getWindow()

Returns

The window that generated this event.

paramString

public String paramString()

Returns

String with current settings of the `WindowEvent`.

Overrides

`ComponentEvent.paramString()`

Description

Helper method for `toString()` to generate string of current settings.

See Also

`ComponentEvent`, `Window`, `WindowAdapter`, `WindowListener`

PREVIOUS HOME NEXT
WindowAdapter ★ BOOK INDEX WindowListener ★

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

WindowListener ★

Name

WindowListener ★

[Graphic: Figure from the text]

Description

Objects that implement the `WindowListener` interface can receive `WindowEvent` objects. Listeners must first register themselves with objects that produce events. When events occur, they are then automatically propagated to all registered listeners.

Interface Definition

```
public abstract interface java.awt.event.WindowListener
   extends java.util.EventListener {
  // Instance Methods
  public abstract void windowActivated (WindowEvent e);
  public abstract void windowClosed (WindowEvent e);
  public abstract void windowClosing (WindowEvent e);
  public abstract void windowDeactivated (WindowEvent e);
  public abstract void windowDeiconified (WindowEvent e);
  public abstract void windowIconified (WindowEvent e);
  public abstract void windowOpened (WindowEvent e);
}
```

Interface Methods

windowActivated

public abstract void windowActivated (WindowEvent e)

Parameters

e

The event that occurred.

Description

Notifies the `WindowListener` that a window has been activated.

windowClosed

public abstract void windowClosed (WindowEvent e)

Parameters

e

The event that occurred.

Description

Notifies the `WindowListener` that a window has closed.

windowClosing

public abstract void windowClosing (WindowEvent e)

Parameters

e

The event that occurred.

Description

Notifies the `WindowListener` that a window is closing.

windowDeactivated

public abstract void windowDeactivated (WindowEvent e)

Parameters

e

The event that occurred.

Description

Notifies the `WindowListener` that a window has been deactivated.

windowDeiconified

public abstract void windowDeiconified (WindowEvent e)

Parameters

e

The event that occurred.

Description

Notifies the `WindowListener` that a window has been restored from an iconified state.

windowIconified

public abstract void windowIconified (WindowEvent e)

Parameters

e

The event that occurred.

Description

Notifies the `WindowListener` that a window has iconified (minimized).

windowOpened

public abstract void windowOpened (WindowEvent e)

Parameters

e

The event that occurred.

Description

Notifies the `WindowListener` that a window has opened.

See Also

`AWTEventMulticaster`, `EventListener`, `Window`, `WindowAdapter`, `WindowEvent`

ColorModel

Name

ColorModel

[Graphic: Figure from the text]

Description

The abstract `ColorModel` class defines the way a Java program represents colors. It provides methods for extracting different color components from a pixel.

Class Definition

```
public class java.awt.image.ColorModel
    extends java.lang.Object {
  // Variables
  protected int pixel_bits;
  // Constructors
  public ColorModel (int bits);
  // Class Methods
  public static ColorModel getRGBdefault();
  // Instance Methods
  public void finalize(); ★
  public abstract int getAlpha (int pixel);
  public abstract int getBlue (int pixel);
  public abstract int getGreen (int pixel);
  public int getPixelSize();
  public abstract int getRed (int pixel);
  public int getRGB (int pixel);
}
```

ProtectedVariables

pixel_bits

protected int pixel_bits

The `pixel_bits` variable saves the ColorModel's `bits` setting (the total number of bits per pixel).

Constructors

ColorModel

public ColorModel (int bits)

Parameters

bits

The number of bits required per pixel using this model.

Description

Constructs a `ColorModel` object.

Class Methods

getRGBdefault

public static ColorModel getRGBdefault()

Returns

> The default `ColorModel` format, which uses 8 bits for each of a pixel's color components: alpha (transparency), red, green, and blue.

Instance Methods

finalize

public void finalize() ★

Overrides

> `Object.finalize()`

Description

> Cleans up when this object is garbage collected.

getAlpha

public abstract int getAlpha (int pixel)

Parameters

> *pixel*
>
> > A pixel encoded with this `ColorModel`.

Returns

> The current alpha setting of the pixel.

getBlue

public abstract int getBlue (int pixel)

Parameters

pixel

A pixel encoded with this `ColorModel`.

Returns

The current blue setting of the pixel.

getGreen

public abstract int getGreen (int pixel)

Parameters

pixel

A pixel encoded with this `ColorModel`.

Returns

The current green setting of the pixel.

getPixelSize

public int getPixelSize()

Returns

The current pixel size for the color model.

getRed

public abstract int getRed (int pixel)

Parameters

pixel

A pixel encoded with this `ColorModel`.

Returns

The current red setting of the pixel.

getRGB

public int getRGB (int pixel)

Parameters

pixel

A pixel encoded with this `ColorModel`.

Returns

The current combined red, green, and blue settings of the pixel.

Description

Gets the color of `pixel` in the default RGB color model.

See Also

`DirectColorModel`, `IndexColorModel`, `Object`

CropImageFilter

Name

CropImageFilter

[Graphic: Figure from the text]

Description

The `CropImageFilter` class creates a smaller image by cropping (i.e., extracting a rectangular region from) a larger image.

Class Definition

```
public class java.awt.image.CropImageFilter
    extends java.awt.image.ImageFilter {
  // Constructors
  public CropImageFilter (int x, int y, int width, int height);
  // Instance Methods
  public void setDimensions (int width, int height);
  public void setPixels (int x, int y, int width, int height, ColorModel model,
    byte[] pixels, int offset, int scansize);
  public void setPixels (int x, int y, int width, int height, ColorModel model,
    int[] pixels, int offset, int scansize);
  public void setProperties (Hashtable properties);
}
```

Constructors

CropImageFilter

public CropImageFilter (int x, int y, int width, int height)

Parameters

x

x-coordinate of top-left corner of piece to crop.

y

y-coordinate of top-left corner of piece to crop.

width

Width of image to crop.

height

Height of image to crop.

Description

Constructs a `CropImageFilter` that crops the specified region from the original image.

Instance Methods

setDimensions

public void setDimensions (int width, int height)

Parameters

width

Ignored parameter.

height

Ignored parameter.

Overrides

`ImageFilter.setDimensions(int, int)`

Description

Called with the original image's dimensions; these dimensions are ignored. The method in turn calls the `ImageConsumer` with the dimensions of the cropped image.

setPixels

public void setPixels (int x, int y, int width, int height, ColorModel model, byte[] pixels, int offset, int scansize)

Parameters

x

x-coordinate of top-left corner of pixel data delivered with this method call.

y

y-coordinate of top-left corner of pixel data delivered with this method call.

width

Width of the rectangle of pixel data delivered with this method call.

height

Height of the rectangle of pixel data delivered with this method call.

model

Color model of image data.

pixels

Image data.

offset

Offset from beginning of the pixels array.

scansize

Size of each line of data in pixels array.

Overrides

`ImageFilter.setPixels(int, int, int, int, ColorModel, byte[], int, int)`

Description

Receives a rectangle of image data from the `ImageProducer`; crops these pixels and delivers them to any `ImageConsumer`s.

public void setPixels (int x, int y, int width, int height, ColorModel model, int[] pixels, int offset, int scansize)

Parameters

x

x-coordinate of top-left corner of pixel data delivered with this method call.

y

y-coordinate of top-left corner of pixel data delivered with this method call.

width

Width of the rectangle of pixel data delivered with this method call.

height

Height of the rectangle of pixel data delivered with this method call.

model

Color model of image data.

pixels

Image data.

offset

Offset from beginning of the pixels array.

scansize

Size of each line of data in pixels array.

Overrides

ImageFilter.setPixels(int, int, int, int, ColorModel, int[], int, int)

Description

Receives a rectangle of image data from the ImageProducer; crops these pixels and delivers them to any ImageConsumers.

setProperties

public void setProperties (Hashtable properties)

Parameters

properties

The properties for the image.

Overrides

ImageFilter.setProperties(Hashtable)

Description

Adds the "croprect" image property to the properties list.

See Also

ColorModel, Hashtable, ImageFilter

PREVIOUS
ColorModel

HOME
BOOK INDEX

NEXT
DirectColorModel

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

DirectColorModel

Name

DirectColorModel

[Graphic: Figure from the text]

Description

The `DirectColorModel` class provides a `ColorModel` that specifies a translation between pixels and alpha, red, green, and blue component values, where the color values are embedded directly within the pixel.

Class Definition

```
public class java.awt.image.DirectColorModel
    extends java.awt.image.ColorModel {
  // Constructors
  public DirectColorModel (int bits, int redMask, int greenMask,
    int blueMask);
  public DirectColorModel (int bits, int redMask, int greenMask,
    int blueMask,
    int alphaMask);
  // Instance Methods
  public final int getAlpha (int pixel);
  public final int getAlphaMask();
  public final int getBlue (int pixel);
  public final int getBlueMask();
  public final int getGreen (int pixel);
  public final int getGreenMask()
  public final int getRed (int pixel);
  public final int getRedMask();
  public final int getRGB (int pixel);
}
```

Constructors

DirectColorModel

public DirectColorModel (int bits, int redMask, int greenMask, int blueMask)

Parameters

bits

The number of bits required per pixel of an image using this model.

redMask

The location of the red component of a pixel.

greenMask

The location of the green component of a pixel.

blueMask

The location of the blue component of a pixel.

Throws

`IllegalArgumentException`

If the mask bits are not contiguous or overlap.

Description

Constructs a `DirectColorModel` object with the given size and color masks; the alpha (transparency) component is not used.

public DirectColorModel (int bits, int redMask, int greenMask, int blueMask, int alphaMask)

Parameters

bits

The number of bits required per pixel of an image using this model.

redMask

The location of the red component of a pixel.

greenMask

The location of the green component of a pixel.

blueMask

The location of the blue component of a pixel.

alphaMask

The location of the alpha component of a pixel.

Throws

`IllegalArgumentException`

If the mask bits are not contiguous or overlap.

Description

Constructs a `DirectColorModel` object with the given size and color masks.

Instance Methods

getAlpha

public final int getAlpha (int pixel)

Parameters

pixel

A pixel encoded with this `ColorModel`.

Returns

The current alpha setting of the pixel.

Overrides

`ColorModel.getAlpha(int)`

getAlphaMask

public final int getAlphaMask()

Returns

The current alpha mask setting of the color model.

getBlue

public final int getBlue (int pixel)

Parameters

pixel

A pixel encoded with this `ColorModel`.

Returns

The current blue setting of the pixel.

Overrides

`ColorModel.getBlue(int)`

getBlueMask

public final int getBlueMask()

Returns

The current blue mask setting of the color model.

getGreen

public final int getGreen (int pixel)

Parameters

pixel

A pixel encoded with this `ColorModel`.

Returns

The current green setting of the pixel.

Overrides

`ColorModel.getGreen(int)`

getGreenMask

public final int getGreenMask()

Returns

The current green mask setting of the color model.

getRed

public final int getRed (int pixel)

Parameters

pixel

A pixel encoded with this `ColorModel`.

Returns

The current red setting of the pixel.

Overrides

`ColorModel.getRed(int)`

getRedMask

public final int getRedMask()

Returns

The current red mask setting of the color model.

getRGB

public final int getRGB (int pixel)

Parameters

pixel

A pixel encoded with this `ColorModel`.

Returns

The current combined red, green, and blue settings of the pixel.

Overrides

`ColorModel.getRGB(int)`

Description

Gets the color of `pixel` in the default RGB color model.

See Also

`ColorModel`

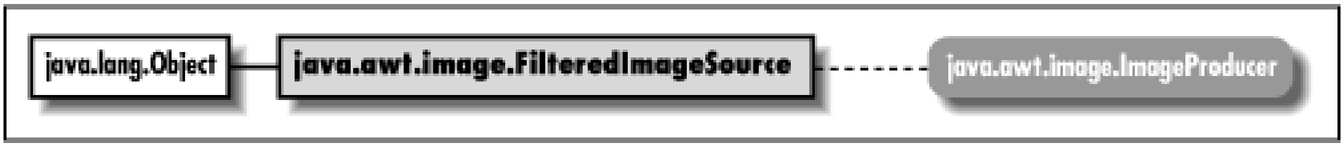
PREVIOUS | HOME | NEXT
CropImageFilter | BOOK INDEX | FilteredImageSource

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

FilteredImageSource

Name

FilteredImageSource


java.lang.Object
java.awt.image.FilteredImageSource
java.awt.image.ImageProducer

Description

The `FilteredImageSource` class acts as glue to put an original `ImageProducer` and `ImageFilter` together to create a new image. As the `ImageProducer` for the new image, `FilteredImageSource` is responsible for registering image consumers for the new image.

Class Definition

```
public class java.awt.image.FilteredImageSource
    extends java.lang.Object
    implements java.awt.image.ImageProducer {
  // Constructors
  public FilteredImageSource (ImageProducer original,
      ImageFilter filter);
  // Instance Methods
  public synchronized void addConsumer (ImageConsumer ic);
  public synchronized boolean isConsumer (ImageConsumer ic);
  public synchronized void removeConsumer (ImageConsumer ic);
  public void requestTopDownLeftRightResend (ImageConsumer ic);
  public void startProduction (ImageConsumer ic);
}
```

Constructors

FilteredImageSource

public FilteredImageSource (ImageProducer original, ImageFilter filter)

Parameters

original

An `ImageProducer` that generates the image to be filtered.

filter

The `ImageFilter` to use to process image data delivered by `original`.

Description

Constructs a `FilteredImageSource` object to filter an image generated by an `ImageProducer`.

Class Methods

addConsumer

public synchronized void addConsumer (ImageConsumer ic)

Parameters

ic

`ImageConsumer` interested in receiving the new image.

Implements

`ImageProducer.addConsumer(ImageConsumer)`

Description

Registers an `ImageConsumer` as interested in `Image` information.

isConsumer

public synchronized boolean isConsumer (ImageConsumer ic)

Parameters

ic

`ImageConsumer` to check.

Returns

`true` if `ImageConsumer` is registered with this `ImageProducer`, `false` otherwise.

Implements

`ImageProducer.isConsumer(ImageConsumer)`

removeConsumer

public synchronized void removeConsumer (ImageConsumer ic)

Parameters

ic

`ImageConsumer` to remove.

Implements

`ImageProducer.removeConsumer(ImageConsumer)`

Description

Removes an `ImageConsumer` from the registered consumers for this `ImageProducer`.

requestTopDownLeftRightResend

public void requestTopDownLeftRightResend (ImageConsumer ic)

Parameters

ic

`ImageConsumer` to communicate with.

Implements

`ImageProducer.requestTopDownLeftRightResend()`

Description

Requests the retransmission of the `Image` data in top-down, left-to-right order.

startProduction

public void startProduction (ImageConsumer ic)

Parameters

ic

`ImageConsumer` to communicate with.

Implements

`ImageProducer.startProduction(ImageConsumer)`

Description

Registers `ImageConsumer` as interested in `Image` information and tells `ImageProducer` to start creating the filtered `Image` data immediately.

See Also

`ImageFilter`, `ImageConsumer`, `ImageProducer`, `Object`

ImageConsumer

Name

ImageConsumer

[Graphic: Figure from the text]

Description

`ImageConsumer` is an interface that provides the means to consume pixel data and render it for display.

Interface Definition

```
public abstract interface java.awt.image.ImageConsumer {
  // Constants
  public final static int COMPLETESCANLINES;
  public final static int IMAGEABORTED;
  public final static int IMAGEERROR;
  public final static int RANDOMPIXELORDER;
  public final static int SINGLEFRAME;
  public final static int SINGLEFRAMEDONE;
  public final static int SINGLEPASS;
  public final static int STATICIMAGEDONE;
  public final static int TOPDOWNLEFTRIGHT;
  // Interface Methods
  public abstract void imageComplete (int status);
  public abstract void setColorModel (ColorModel model);
  public abstract void setDimensions (int width, int height);
  public abstract void setHints (int hints);
  public abstract void setPixels (int x, int y, int width, int height,
      ColorModel model, byte[] pixels, int offset, int scansize);
  public abstract void setPixels (int x, int y, int width, int height,
      ColorModel model, int[] pixels, int offset, int scansize);
  public abstract void setProperties (Hashtable properties);
}
```

Constants

COMPLETESCANLINES

public final static int COMPLETESCANLINES

Hint flag for the `setHints(int)` method; indicates that the image will be delivered one or more scanlines at a time.

IMAGEABORTED

public final static int IMAGEABORTED

Status flag for the `imageComplete(int)` method indicating that the loading process for the image aborted.

IMAGEERROR

public final static int IMAGEERROR

Status flag for the `imageComplete(int)` method indicating that an error happened during image loading.

RANDOMPIXELORDER

public final static int RANDOMPIXELORDER

Hint flag for the `setHints(int)` method; indicates that the pixels will be delivered in no particular order.

SINGLEFRAME

public final static int SINGLEFRAME

Hint flag for the `setHints(int)` method; indicates that the image consists of a single frame.

SINGLEFRAMEDONE

public final static int SINGLEFRAMEDONE

Status flag for the `imageComplete(int)` method indicating a single frame of the image has loaded.

SINGLEPASS

public final static int SINGLEPASS

Hint flag for the `setHints(int)` method; indicates that each pixel will be delivered once (i.e., the producer will not make multiple passes over the image).

STATICIMAGEDONE

public final static int STATICIMAGEDONE

Status flag for the `imageComplete(int)` method indicating that the image has fully and successfully loaded, and that there are no additional frames.

TOPDOWNLEFTRIGHT

public final static int TOPDOWNLEFTRIGHT

Hint flag for the `setHints(int)` method; indicates that pixels will be delivered in a top to bottom, left to right order.

Interface Methods

imageComplete

public abstract void imageComplete (int status)

Parameters

status

Image loading status flags.

Description

Called when the image, or a frame of an image sequence, is complete to report the completion status.

setColorModel

public abstract void setColorModel (ColorModel model)

Parameters

model

The color model for the image.

Description

Tells the `ImageConsumer` the color model used for most of the pixels in the image.

setDimensions

public abstract void setDimensions (int width, int height)

Parameters

width

Width for image.

height

Height for image.

Description

Tells the consumer the image's dimensions.

setHints

public abstract void setHints (int hints)

Parameters

hints

Image consumption hints.

Description

Gives the consumer information about how pixels will be delivered.

setPixels

public abstract void setPixels (int x, int y, int width, int height, ColorModel model, byte[] pixels, int offset, int scansize)

Parameters

x

x-coordinate of top-left corner of pixel data delivered with this method call.

y

y-coordinate of top-left corner of pixel data delivered with this method call.

width

Width of the rectangle of pixel data delivered with this method call.

height

Height of the rectangle of pixel data delivered with this method call.

model

Color model of image data.

pixels

Image data.

offset

Offset from beginning of the pixels array.

scansize

Size of each line of data in pixels array.

Description

Delivers a rectangular block of pixels to the image consumer.

public abstract void setPixels (int x, int y, int width, int height, ColorModel model, int[] pixels, int offset, int scansize)

Parameters

x

x-coordinate of top-left corner of pixel data delivered with this method call.

y

y-coordinate of top-left corner of pixel data delivered with this method call.

width

Width of the rectangle of pixel data delivered with this method call.

height

Height of the rectangle of pixel data delivered with this method call.

model

Color model of image data.

pixels

Image data.

offset

Offset from beginning of the pixels array.

scansize

Size of each line of data in pixels array.

Description

Delivers a rectangular block of pixels to the image consumer.

setProperties

public abstract void setProperties (Hashtable properties)

Parameters

properties

The properties for the image.

Description

Delivers a Hashtable that contains the image's properties.

See Also

`ColorModel`, `Hashtable`, `ImageFilter`, `PixelGrabber`, `Object`

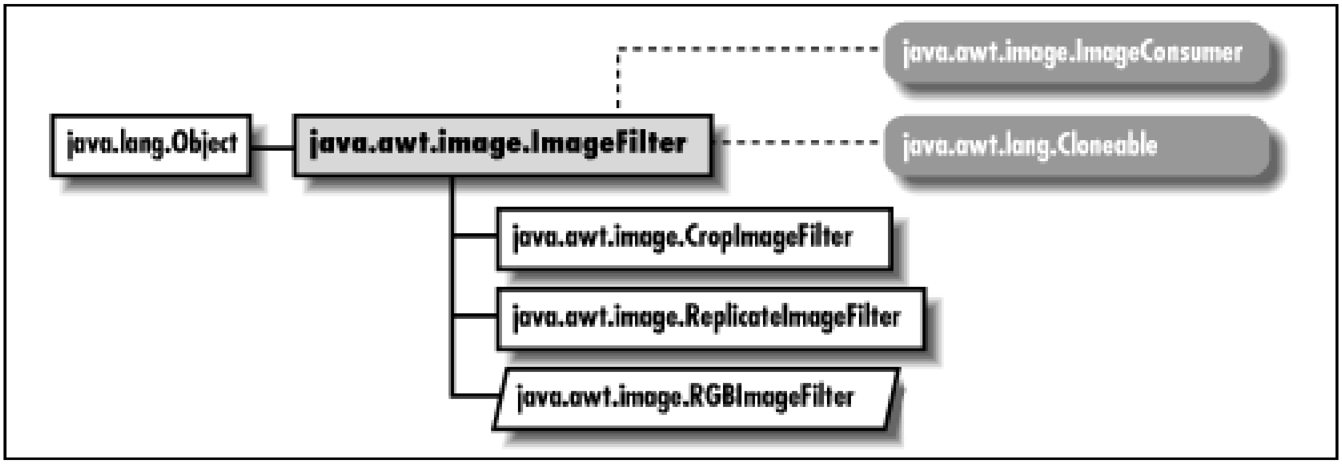
PREVIOUS HOME NEXT
FilteredImageSource BOOK INDEX ImageFilter

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

ImageFilter

Name

ImageFilter


java.awt.image.ImageConsumer
java.lang.Object
java.awt.image.ImageFilter
java.awt.lang.Cloneable
java.awt.image.CropImageFilter
java.awt.image.ReplicateImageFilter
java.awt.image.RGBImageFilter

Description

The `ImageFilter` class sits between the `ImageProducer` and `ImageConsumer` as an image is being created to provide a filtered version of that image. Image filters are always used in conjunction with a `FilteredImageSource`. As an implementer of the `ImageConsumer` interface, an image filter receives pixel data from the original image's source and delivers it to another image consumer. The `ImageFilter` class implements a null filter (i.e., the new image is the same as the original); to produce a filter that modifies an image, create a subclass of `ImageFilter`.

Class Definition

```
public class java.awt.image.ImageFilter
    extends java.lang.Object
    implements java.awt.image.ImageConsumer, java.lang.Cloneable {
  // Variables
  protected ImageConsumer consumer;
  // Constructors
  public ImageFilter();
  // Instance Methods
  public Object clone();
  public ImageFilter getFilterInstance (ImageConsumer ic);
  public void imageComplete (int status);
  public void resendTopDownLeftRight (ImageProducer ip);
  public void setColorModel (ColorModel model);
  public void setDimensions (int width, int height);
  public void setHints (int hints);
  public void setPixels (int x, int y, int width, int height,
      ColorModel model, byte[] pixels, int offset, int scansize);
  public void setPixels (int x, int y, int width, int height,
      ColorModel model, int[] pixels, int offset, int scansize);
  public void setProperties (Hashtable properties);
}
```

Protected Variables

consumer

protected ImageConsumer consumer

The consumer variable is a reference to the actual `ImageConsumer` for the `Image`.

Constructors

ImageFilter

public ImageFilter()

Description

> Constructs an empty `ImageFilter` instance.

Instance Methods

clone

public Object clone()

Overrides

`Object.clone()`

Returns

A copy of the `ImageFilter` instance.

getFilterInstance

public ImageFilter getFilterInstance (ImageConsumer ic)

Parameters

ic

The consumer in question.

Returns

A copy of the `ImageFilter` instance.

Description

Returns the filter that will do the filtering for `ic`.

imageComplete

void imageComplete (int status)

Parameters

status

Image loading completion status flags.

Implements

`ImageConsumer.imageComplete(int)`

Description

Called by the `ImageProducer` to indicate an image's completion status. `ImageFilter` passes these flags to the consumer unchanged.

resendTopDownLeftRight

public void resendTopDownLeftRight (ImageProducer ip)

Parameters

ip

The `ImageProducer` generating the original image.

Description

Called by the `ImageConsumer` to ask the filter to resend the image data in the top-down, left-to-right order. In `ImageFilter`, this method calls the same method in the `ImageProducer`, thus relaying the request.

setColorModel

void setColorModel (ColorModel model)

Parameters

model

The color model for the image.

Implements

`ImageConsumer.setColorModel(ColorModel)`

Description

Sets the image's color model.

setDimensions

void setDimensions (int width, int height)

Parameters

width

Width for image.

height

Height for image.

Implements

```
ImageConsumer.setDimensions(int, int)
```

Description

Sets the image's dimensions.

setHints

void setHints (int hints)

Parameters

hints

Image consumption hints.

Implements

```
ImageConsumer.setHints(int)
```

Description

Called by the `ImageProducer` to deliver hints about how the image data will be delivered. `ImageFilter` passes these hints on to the `ImageConsumer`.

setPixels

void setPixels (int x, int y, int width, int height, ColorModel model, byte[] pixels, int offset, int scansize)

Parameters

x

x-coordinate of top-left corner of pixel data delivered with this method call.

y

y-coordinate of top-left corner of pixel data delivered with this method call.

width

Width of the rectangle of pixel data delivered with this method call.

height

Height of the rectangle of pixel data delivered with this method call.

model

Color model of image data.

pixels

Image data.

offset

Offset from beginning of the pixels array.

scansize

Size of each line of data in pixels array.

Implements

`ImageConsumer.setPixels(int, int, int, int, ColorModel, byte[], int, int)`

Description

Delivers a rectangular block of pixels to the `ImageFilter`. `ImageFilter` passes these pixels on to the consumer unchanged.

void setPixels (int x, int y, int width, int height, ColorModel model, int[] pixels, int offset, int scansize)

Parameters

x

x-coordinate of top-left corner of pixel data delivered with this method call.

y

y-coordinate of top-left corner of pixel data delivered with this method call.

width

Width of the rectangle of pixel data delivered with this method call.

height

Height of the rectangle of pixel data delivered with this method call.

model

Color model of image data.

pixels

Image data.

offset

Offset from beginning of the pixels array.

scansize

Size of each line of data in pixels array.

Implements

`ImageConsumer.setPixels(int, int, int, int, ColorModel, int[], int, int)`

Description

Delivers a rectangular block of pixels to the `ImageFilter`. `ImageFilter` passes these pixels on to the consumer unchanged.

setProperties

void setProperties (Hashtable properties)

Parameters

properties

The properties for the image.

Implements

`ImageConsumer.setProperties(Hashtable)`

Description

Initializes the image's properties. `ImageFilter` adds the property "filter" to the `Hashtable`, and passes the result on to the image consumer; the value of the property is the string returned by the filter's `toString()` method. If the property "filter" is already in the `Hashtable`, `ImageFilter` adds the string returned by its `toString()` method to the value already associated with that property.

See Also

`Cloneable`, `ColorModel`, `CropImageFilter`, `Hashtable`, `ImageConsumer`, `ImageProducer`, `Object`, `ReplicateImageFilter`, `RGBImageFilter`

ImageObserver

Name

ImageObserver

[Graphic: Figure from the text]

Description

`ImageObserver` is an interface that provides constants and the callback mechanism to receive asynchronous information about the status of an image as it loads.

Interface Definition

```
public abstract interface java.awt.image.ImageObserver {
  // Constants
  public static final int ABORT;
  public static final int ALLBITS;
  public static final int ERROR;
  public static final int FRAMEBITS;
  public static final int HEIGHT;
  public static final int PROPERTIES;
  public static final int SOMEBITS;
  public static final int WIDTH;
  // Interface Methods
  public abstract boolean imageUpdate (Image image, int infoflags,
     int x, int y, int width, int height);
}
```

Constants

ABORT

public static final int ABORT

The ABORT flag indicates that the image aborted during loading. An attempt to reload the image may succeed, unless ERROR is also set.

ALLBITS

public static final int ALLBITS

The ALLBITS flag indicates that the image has completely loaded successfully. The x, y, width, and height arguments to imageUpdate() should be ignored.

ERROR

public static final int ERROR

The ERROR flag indicates that an error happened during the image loading process. An attempt to reload the image will fail.

FRAMEBITS

public static final int FRAMEBITS

The FRAMEBITS flag indicates that a complete frame of a multi-frame image has loaded. The x, y, width, and height arguments to imageUpdate() should be ignored.

HEIGHT

public static final int HEIGHT

The HEIGHT flag indicates that the height information is available for an image; the image's height is in the height argument to imageUpdate().

PROPERTIES

public static final int PROPERTIES

The `PROPERTIES` flag indicates that the properties information is available for an image.

SOMEBITS

public static final int SOMEBITS

The `SOMEBITS` flag indicates that the image has started loading and some pixels are available. The bounding rectangle for the pixels that have been delivered so far is indicated by the `x`, `y`, `width`, and `height` arguments to `imageUpdate()`.

WIDTH

public static final int WIDTH

The `WIDTH` flag indicates that the width information is available for an image; the image's width is in the `width` argument to `imageUpdate()`.

Interface Methods

imageUpdate

public abstract boolean imageUpdate (Image image, int infoflags, int x, int y, int width, int height)

Parameters

image

Image that is being loaded.

infoflags

The `ImageObserver` flags for the information that is currently available.

x

Meaning depends on `infoflags` that are set.

y

Meaning depends on `infoflags` that are set.

width

Meaning depends on `infoflags` that are set.

height

Meaning depends on `infoflags` that are set.

Returns

`true` if image has completed loading (successfully or unsuccessfully), `false` if additional information needs to be loaded.

Description

Provides the callback mechanism for the asynchronous loading of images.

See Also

`Component`, `Image`, `Object`

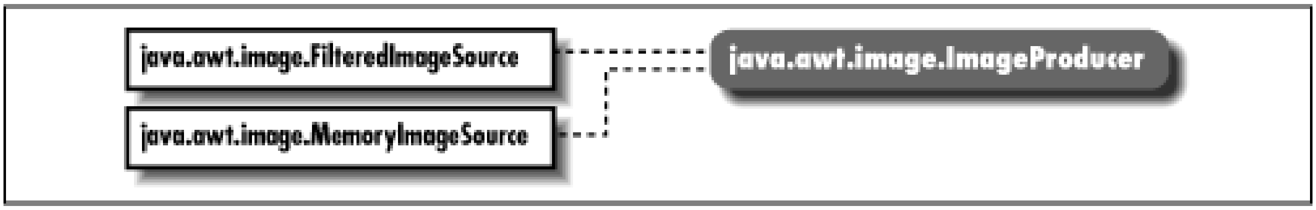
PREVIOUS HOME NEXT
ImageFilter BOOK INDEX ImageProducer

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

ImageProducer

Name

ImageProducer


java.awt.image.FilteredImageSource
java.awt.image.MemoryImageSource
java.awt.image.ImageProducer

Description

`ImageProducer` is an interface that provides the methods necessary for the production of images and the communication with classes that implement the `ImageConsumer` interface.

Interface Definition

```
public abstract interface java.awt.image.ImageProducer {
  // Interface Methods
  public abstract void addConsumer (ImageConsumer ic);
  public abstract boolean isConsumer (ImageConsumer ic);
  public abstract void removeConsumer (ImageConsumer ic);
  public abstract void requestTopDownLeftRightResend (ImageConsumer ic);
  public abstract void startProduction (ImageConsumer ic);
}
```

Interface Methods

addConsumer

public abstract void addConsumer (ImageConsumer ic)

Parameters

ic

An `ImageConsumer` that wants to receive image data.

Description

Registers an `ImageConsumer` as interested in image information.

isConsumer

public abstract boolean isConsumer (ImageConsumer ic)

Parameters

ic

`ImageConsumer` to check.

Returns

`true` if `ImageConsumer` has registered with the `ImageProducer`, `false` otherwise.

removeConsumer

public abstract void removeConsumer (ImageConsumer ic)

Parameters

ic

`ImageConsumer` to remove.

Description

Removes an `ImageConsumer` from registered consumers for this `ImageProducer`.

requestTopDownLeftRightResend

public abstract void requestTopDownLeftRightResend (ImageConsumer ic)

Parameters

ic

`ImageConsumer` to communicate with.

Description

Requests the retransmission of the image data in top-down, left-to-right order.

startProduction

public abstract void startProduction (ImageConsumer ic)

Parameters

ic

ImageConsumer to communicate with.

Description

Registers `ImageConsumer` as interested in image information and tells `ImageProducer` to start sending the image data immediately.

See Also

`FilteredImageSource`, `Image`, `ImageConsumer`, `ImageFilter`, `MemoryImageSource`, `Object`

PREVIOUS

NEXT

IndexColorModel

Name

IndexColorModel

[Graphic: Figure from the text]

Description

The `IndexColorModel` class is a `ColorModel` that uses a color map lookup table (with a maximum size of 256) to convert pixel values into their alpha, red, green, and blue component parts.

Class Definition

```
public class java.awt.image.IndexColorModel
    extends java.awt.image.ColorModel {
  // Constructors
  public IndexColorModel (int bits, int size,
      byte[] colorMap, int start, boolean hasalpha);
  public IndexColorModel (int bits, int size,
      byte[] colorMap, int start, boolean hasalpha, int transparent);
  public IndexColorModel (int bits, int size,
      byte[] red, byte[] green, byte[] blue);
  public IndexColorModel (int bits, int size,
      byte[] red, byte[] green, byte[] blue, byte[] alpha);
  public IndexColorModel (int bits, int size,
      byte[] red, byte[] green, byte[] blue, int transparent);
  // Instance Methods
  public final int getAlpha (int pixel);
  public final void getAlphas (byte[] alphas);
  public final int getBlue (int pixel);
  public final void getBlues (byte[] blues);
  public final int getGreen (int pixel);
  public final void getGreens (byte[] greens);
  public final int getMapSize();
  public final int getRed (int pixel);
  public final void getReds (byte[] reds);
  public final int getRGB (int pixel);
  public final int getTransparentPixel();
}
```

Constructors

IndexColorModel

public IndexColorModel (int bits, int size, byte[] colorMap, int start, boolean hasalpha)

Parameters

bits

The number of bits in a pixel.

size

The number of entries in the color map. Note: this is not the size of the `colorMap` parameter.

colorMap

Color component values in red, green, blue, alpha order; the alpha component is optional, and may not be present.

start

The starting position in `colorMap` array.

hasalpha

If `hasalpha` is `true`, alpha components are present in `colorMap` array.

Throws

`ArrayIndexOutOfBoundsException`

If `size` is invalid.

Description

Constructs an `IndexColorModel` object with the given component settings. The size of `colorMap` must be at least `3*size+start`, if `hasalpha` is `false`; if `hasalpha` is `true`, `colorMap.length` must be at least `4*size+start`.

public IndexColorModel (int bits, int size, byte[] colorMap, int start, boolean hasalpha, int transparent)

Parameters

bits

The number of bits in a pixel.

size

The number of entries in the color map. Note: this is not the size of the `colorMap` parameter.

colorMap

Color component values in red, green, blue, alpha order; the alpha component is optional, and may not be present.

start

The starting position in `colorMap` array.

hasalpha

If `hasalpha` is `true`, alpha components are present in `colorMap` array.

transparent

Position of `colorMap` entry for transparent pixel entry.

Throws

`ArrayIndexOutOfBoundsException`

If `size` invalid.

Description

Constructs an `IndexColorModel` object with the given component settings. The size of `colorMap` must be at least `3*size+start`, if `hasalpha` is `false`; if `hasalpha` is `true`, `colorMap.length` must be at least `4*size+start`. The color map has a transparent pixel; its location is given by `transparent`.

public IndexColorModel (int bits, int size, byte[] red, byte[] green, byte[] blue)

Parameters

bits

The number of bits in a pixel.

size

The number of entries in the color map.

red

Red color component values.

green

Green color component values.

blue

Blue color component values.

Throws

`ArrayIndexOutOfBoundsException`

If `size` invalid.

Description

Constructs an `IndexColorModel` object with the given component settings. There is no alpha component. The length of the `red`, `green`, and `blue` arrays must be greater than `size`.

public IndexColorModel (int bits, int size, byte[] red, byte[] green, byte[] blue, byte[] alpha)

Parameters

bits

The number of bits in a pixel.

size

The number of entries in the color map.

red

Red color component values.

green

Green color component values.

blue

Blue color component values.

alpha

Alpha component values.

Throws

`ArrayIndexOutOfBoundsException`

If `size` is invalid.

`NullPointerException`

If `size` is positive and `alpha` array is `null`.

Description

Constructs an `IndexColorModel` object with the given component settings. The length of the `red`, `green`, `blue`, and `alpha` arrays must be greater than `size`.

public IndexColorModel (int bits, int size, byte[] red, byte[] green, byte[] blue, int transparent)

Parameters

bits

The number of bits in a pixel.

size

The number of entries in the color map.

red

Red color component values.

green

Green color component values.

blue

Blue color component values.

transparent

Position of transparent pixel entry.

Throws

`ArrayIndexOutOfBoundsException`

If `size` is invalid.

Description

Constructs an `IndexColorModel` object with the given component settings. The length of the `red`, `green`, `blue`, and `alpha` arrays must be greater than `size`. The color map has a transparent pixel; its location is given by `transparent`.

Instance Methods

getAlpha

public final int getAlpha (int pixel)

Parameters

pixel

A pixel encoded with this `ColorModel`.

Returns

The current alpha setting of the pixel.

Overrides

`ColorModel.getAlpha(int)`

getAlphas

public final void getAlphas (byte[] alphas)

Parameters

alphas

The alpha values of the pixels in the color model.

Description

Copies the alpha values from the color map into the array `alphas[]`.

getBlue

public final int getBlue (int pixel)

Parameters

pixel

A pixel encoded with this `ColorModel`.

Returns

The current blue setting of the pixel.

Overrides

`ColorModel.getBlue(int)`

getBlues

public final void getBlues (byte[] blues)

Parameters

blues

The blue values of the pixels in the color model.

Description

Copies the blue values from the color map into the array `blues[]`.

getGreen

public final int getGreen (int pixel)

Parameters

pixel

A pixel encoded with this `ColorModel`.

Returns

The current green setting of the pixel.

Overrides

`ColorModel.getGreen(int)`

getGreens

public final void getGreens (byte[] greens)

Parameters

greens

The green values of the pixels in the color model.

Description

Copies the green values from the color map into the array `greens[]`.

getMapSize

public final int getMapSize()

Returns

The current size of the color map table.

getRed

public final int getRed (int pixel)

Parameters

pixel

A pixel encoded with this `ColorModel`.

Returns

The current red setting of the pixel.

Overrides

`ColorModel.getRed(int)`

getReds

public final void getReds (byte[] reds)

Parameters

reds

The red values of the pixels in the color model.

Description

Copies the red values from the color map into the array `reds[]`.

getRGB

public final int getRGB (int pixel)

Parameters

pixel

A pixel encoded with this `ColorModel`.

Returns

The current combined red, green, and blue settings of the pixel.

Overrides

`ColorModel.getRGB(int)`

Description

Gets the color of `pixel` in the default RGB color model.

getTransparentPixel

public final int getTransparentPixel()

Returns

The array index for the transparent pixel in the color model.

See Also

`ColorModel`

PREVIOUS | HOME | NEXT

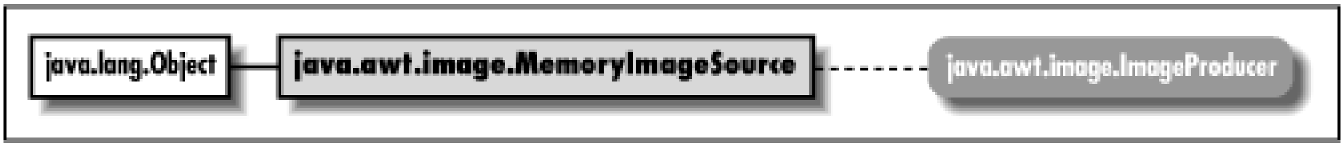
ImageProducer | BOOK INDEX | MemoryImageSource

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

MemoryImageSource

Name

MemoryImageSource


java.lang.Object
java.awt.image.MemoryImageSource
java.awt.image.ImageProducer

Description

The `MemoryImageSource` class allows you to create images completely in memory. You provide an array of data; it serves as an image producer for that data. In the 1.1 release, new methods support using this class for animation (notably `setAnimated()` and the various overrides of `newPixels()`).

Class Definition

```
public class java.awt.image.MemoryImageSource
    extends java.lang.Object
    implements java.awt.image.ImageProducer {
  // Constructors
  public MemoryImageSource (int w, int h, ColorModel cm,
      byte[] pix, int off, int scan);
  public MemoryImageSource (int w, int h, ColorModel cm,
      byte[] pix, int off, int scan, Hashtable props);
  public MemoryImageSource (int w, int h, ColorModel cm,
      int[] pix, int off, int scan);
  public MemoryImageSource (int w, int h, ColorModel cm,
      int[] pix, int off, int scan, Hashtable props);
  public MemoryImageSource (int w, int h, int[] pix,
      int off, int scan);
  public MemoryImageSource (int w, int h, int[] pix,
      int off, int scan, Hashtable props);
  // Instance Methods
  public synchronized void addConsumer (ImageConsumer ic);
  public synchronized boolean isConsumer (ImageConsumer ic);
  public void newPixels(); ★
  public synchronized void newPixels (int x, int y,
      int w, int h); ★
  public synchronized void newPixels (int x, int y,
      int w, int h, boolean framenotify); ★
  public synchronized void newPixels (byte[] newpix,
      ColorModel newmodel, int offset, int scansize); ★
  public synchronized void newPixels (int[] newpix,
      ColorModel newmodel, int offset, int scansize); ★
  public synchronized void removeConsumer (ImageConsumer ic);
  public void requestTopDownLeftRightResend (ImageConsumer ic);
  public synchronized void setAnimated (boolean animated); ★
  public synchronized void setFullBufferUpdates
      (boolean fullbuffers); ★
  public void startProduction (ImageConsumer ic);
}
```

Constructors

MemoryImageSource

public MemoryImageSource (int w, int h, ColorModel cm, byte[] pix, int off, int scan)

Parameters

w

Width of the image being created.

h

Height of the image being created.

cm

`ColorModel` of the image being created.

pix

Array of pixel information.

off

The offset of the first pixel in the array; elements prior to this pixel are ignored.

scan

The number of pixels per scan line in the array.

Description

Constructs a `MemoryImageSource` object with the given parameters to serve as an `ImageProducer` for a new image.

public MemoryImageSource (int w, int h, ColorModel cm, byte[] pix, int off, int scan, Hashtable props)

Parameters

w

Width of the image being created.

h

Height of the image being created.

cm

`ColorModel` of the image being created.

pix

Array of pixel information.

off

The offset of the first pixel in the array; elements prior to this pixel are ignored.

scan

The number of pixels per scan line in the array.

props

`Hashtable` of properties associated with image.

Description

Constructs a `MemoryImageSource` object with the given parameters to serve as an `ImageProducer` for a new image.

public MemoryImageSource (int w, int h, ColorModel cm, int[] pix, int off, int scan)

Parameters

w

Width of the image being created.

h

Height of the image being created.

cm

`ColorModel` of the image being created.

pix

Array of pixel information.

off

The offset of the first pixel in the array; elements prior to this pixel are ignored.

scan

The number of pixels per scan line in the array.

Description

Constructs a `MemoryImageSource` object with the given parameters to serve as an `ImageProducer` for a new image.

public MemoryImageSource (int w, int h, ColorModel cm, int[] pix, int off, int scan, Hashtable props)

Parameters

w

Width of the image being created.

h

Height of the image being created.

cm

`ColorModel` of the image being created.

pix

Array of pixel information.

off

The offset of the first pixel in the array; elements prior to this pixel are ignored.

scan

The number of pixels per scan line in the array.

props

Hashtable of properties associated with image.

Description

Constructs a MemoryImageSource object with the given parameters to serve as an ImageProducer for a new image.

public MemoryImageSource (int w, int h, int[] pix, int off, int scan)

Parameters

w

Width of the image being created.

h

Height of the image being created.

pix

Array of pixel information.

off

The offset of the first pixel in the array; elements prior to this pixel are ignored.

scan

The number of pixels per scan line in the array.

Description

Constructs a MemoryImageSource object with the given parameters to serve as an ImageProducer for a new image.

public MemoryImageSource (int w, int h, int[] pix, int off, int scan, Hashtable props)

Parameters

w

Width of the image being created.

h

Height of the image being created.

pix

Array of pixel information.

off

The offset of the first pixel in the array; elements prior to this pixel are ignored.

scan

The number of pixels per scan line in the array.

props

`Hashtable` of properties associated with image.

Description

Constructs a `MemoryImageSource` object with the given parameters to serve as an `ImageProducer` for a new image.

Class Methods

addConsumer

public synchronized void addConsumer (ImageConsumer ic)

Parameters

ic

`ImageConsumer` requesting image data.

Implements

`ImageProducer.addConsumer(ImageConsumer)`

Description

Registers an `ImageConsumer` as interested in `Image` information.

isConsumer

public synchronized boolean isConsumer (ImageConsumer ic)

Parameters

ic

`ImageConsumer` to check.

Returns

`true` if `ImageConsumer` is registered with this `ImageProducer`, `false` otherwise.

Implements

`ImageProducer.isConsumer(ImageConsumer)`

newPixels

public synchronized void newPixels() ★

Description

Notifies the `MemoryImageSource` that there is new data available. The `MemoryImageSource` notifies all `ImageConsumer`s that there is new data, sending the full rectangle and notifying the consumers that the frame is complete.

public synchronized void newPixels (int x, int y, int w, int h, boolean framenotify) ★

Parameters

x

x coordinate of the top left corner of the new image data.

y

y coordinate of the top left corner of the new image data.

w

Width of the new image data.

h

Height of the new image data.

Description

Notifies the `MemoryImageSource` that there is new data available. The `MemoryImageSource` notifies all `ImageConsumer`s that there is new data in the rectangle described by `x`, `y`, `w`, and `h`. The consumers are notified that the frame is complete.

public synchronized void newPixels (int x, int y, int w, int h, boolean framenotify) ★

Parameters

x

x coordinate of the top left corner of the new image data.

y

y coordinate of the top left corner of the new image data.

w

Width of the new image data.

h

Height of the new image data.

framenotify

Determines whether this is a complete frame or not.

Description

Notifies the `MemoryImageSource` that there is new data available. The `MemoryImageSource` notifies all `ImageConsumer`s that there is new data in the rectangle described by `x`, `y`, `w`, and `h`. If `framenotify` is `true`, the consumers will also be notified that a frame is complete.

public synchronized void newPixels (byte[] newpix, ColorModel newmodel, int offset, int scansize) ★

Parameters

newpix

New array of image data.

newmodel

The color model to use for the new data.

offset

Offset into the data array

scansize

Size of each line.

Description

Changes the image data for this `MemoryImageSource` and notifies its `ImageConsumer`s that new data is available.

public synchronized void newPixels (int[] newpix, ColorModel newmodel, int offset, int scansize) ★

Parameters

newpix

New array of image data.

newmodel

The color model to use for the new data.

offset

Offset into the data array

scansize

Size of each line.

Description

Changes the image data for this `MemoryImageSource` and notifies its `ImageConsumer`s that new data is available.

removeConsumer

public void removeConsumer (ImageConsumer ic)

Parameters

ic

`ImageConsumer` to remove.

Implements

`ImageProducer.removeConsumer(ImageConsumer)`

Description

Removes an `ImageConsumer` from registered consumers for this `ImageProducer`.

requestTopDownLeftRightResend

public void requestTopDownLeftRightResend (ImageConsumer ic)

Parameters

ic

`ImageConsumer` requesting image data.

Implements

`ImageProducer.requestTopDownLeftRightResend(ImageConsumer)`

Description

Requests the retransmission of the `Image` data in top-down, left-to-right order.

setAnimated

public void setAnimated (boolean animated) ★

Parameters

animated

Flag indicating whether this image is animated.

Description

To use this MemoryImageSource for animation, call `setAnimated(true)`. The `newPixels()` methods will not work otherwise.

setFullBufferUpdates

public void setFullBufferUpdates (boolean fullbuffers) ★

Parameters

fullbuffers

`true` to send full buffers; `false` otherwise.

Description

This method is only important for animations; i.e., you should call `setAnimated(true)` before using this function. If you do request to send full buffers, then any rectangle parameters passed to `newPixels()` will be ignored and the entire image will be sent to the consumers.

startProduction

public void startProduction (ImageConsumer ic)

Parameters

ic

`ImageConsumer` requesting image data.

Implements

`ImageProducer.startProduction(ImageConsumer)`

Description

Registers `ImageConsumer` as interested in Image information and tells `ImageProducer` to start sending the image data immediately.

See Also

`ColorModel`, `Hashtable`, `ImageConsumer`, `ImageProducer`, `Object`

PREVIOUS HOME NEXT
IndexColorModel BOOK INDEX PixelGrabber

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

PixelGrabber

Name

PixelGrabber

[Graphic: Figure from the text]

Description

The `PixelGrabber` class is an `ImageConsumer` that captures the pixels from an image and saves them in an array.

Class Definition

```
public class java.awt.image.PixelGrabber
    extends java.lang.Object
    implements java.awt.image.ImageConsumer {
  // Constructors
  public PixelGrabber (Image img, int x, int y, int w, int h,
      boolean forceRGB); ★
  public PixelGrabber (Image image, int x, int y, int width,
      int height, int[] pixels, int offset, int scansize);
  public PixelGrabber (ImageProducer ip, int x, int y, int width,
      int height, int[] pixels, int offset, int scansize);
  // Instance Methods
  public synchronized void abortGrabbing(); ★
  public synchronized ColorModel getColorModel(); ★
  public synchronized int getHeight(); ★
  public synchronized Object getPixels(); ★
  public synchronized int getStatus(); ★
  public synchronized int getWidth(); ★
  public boolean grabPixels() throws InterruptedException;
  public synchronized boolean grabPixels (long ms)
      throws InterruptedException;
  public synchronized void imageComplete (int status);
  public void setColorModel (ColorModel model);
  public void setDimensions (int width, int height);
  public void setHints (int hints);
  public void setPixels (int x, int y, int width, int height,
      ColorModel model, byte[] pixels, int offset, int scansize);
  public void setPixels (int x, int y, int width, int height,
      ColorModel model, int[] pixels, int offset, int scansize);
  public void setProperties (Hashtable properties);
  public synchronized void startGrabbing(); ★
  public synchronized int status(); ☆
}
```

Constructors

PixelGrabber

public PixelGrabber (Image img, int x, int y, int w, int h, boolean forceRGB) ★

Parameters

img

`Image` to use as source of pixel data.

x

x-coordinate of top-left corner of pixel data.

y

y-coordinate of top-left corner of pixel data.

w

Width of pixel data.

h

Height of pixel data.

forceRGB

`true` to force the use of the RGB color model; `false` otherwise.

Description

Constructs a `PixelGrabber` instance to grab the specified area of the image.

public PixelGrabber (Image image, int x, int y, int width, int height, int[] pixels, int offset, int scansize)

Parameters

image

`Image` to use as source of pixel data.

x

x-coordinate of top-left corner of pixel data.

y

y-coordinate of top-left corner of pixel data.

width

Width of pixel data.

height

Height of pixel data.

pixels

Where to store pixel data when `grabPixels()` called.

offset

Offset from beginning of each line in pixels array.

scansize

Size of each line of data in pixels array.

Description

Constructs a `PixelGrabber` instance to grab the specified area of the image and store the pixel data from this area in the array `pixels[]`.

public PixelGrabber (ImageProducer ip, int x, int y, int width, int height, int[] pixels, int offset, int scansize)

Parameters

ip

`ImageProducer` to use as source of pixel data.

x

x-coordinate of top-left corner of pixel data.

y

y-coordinate of top-left corner of pixel data.

width

Width of pixel data.

height

Height of pixel data.

pixels

Where to store pixel data when `grabPixels()` called.

offset

Offset from beginning of each line in pixels array.

scansize

Size of each line of data in pixels array.

Description

Constructs a `PixelGrabber` instance to grab data from the specified area of the image generated by an `ImageProducer` and store the pixel data from this area in the array `pixels[]`.

Instance Methods

abortGrabbing

public synchronized void abortGrabbing() ★

Description

Stops the `PixelGrabber`'s image-grabbing process.

getColorModel

public synchronized ColorModel getColorModel() ★

Returns

The color model the `PixelGrabber` is using for its array.

getHeight

public synchronized int getHeight() ★

Returns

> The height of the grabbed image, or -1 if the height is not known.

getPixels

public synchronized Object getPixels() ★

Returns

> The array of pixels.

Description

> Either a byte array or an integer array is returned, or `null` if the size and format of the image are not yet known. Because the `PixelGrabber` may change its mind about what `ColorModel` it's using, different calls to this method may return different arrays until the image acquisition is complete.

getStatus

public synchronized int getStatus() ★

Returns

> A combination of `ImageObserver` flags indicating what data is available.

getWidth

public synchronized int getWidth() ★

Returns

> The width of the grabbed image, or -1 if the width is not known.

grabPixels

public boolean grabPixels() throws InterruptedException

Throws

`InterruptedException`

If image grabbing is interrupted before completion.

Returns

`true` if the image has completed loading, `false` if the loading process aborted or an error occurred.

Description

Starts the process of grabbing the pixel data from the source and storing it in the array `pixels[]` from constructor. Returns when the image is complete, loading aborts, or an error occurs.

public synchronized boolean grabPixels (long ms) throws InterruptedException

Parameters

ms

Milliseconds to wait for completion.

Returns

`true` if image has completed loading, `false` if the loading process aborted, or an error or a timeout occurred.

Throws

`InterruptedException`

If image grabbing is interrupted before completion.

Description

Starts the process of grabbing the pixel data from the source and storing it in the array `pixels[]` from constructor. Returns when the image is complete, loading aborts, an error occurs, or a timeout occurs.

imageComplete

public synchronized void imageComplete (int status)

Parameters

status

Image loading completion status flags.

Implements

`ImageConsumer.imageComplete(int)`

Description

Called by the `ImageProducer` to indicate that the image has been delivered.

setColorModel

void setColorModel (ColorModel model)

Parameters

model

The color model for the image.

Implements

`ImageConsumer.setColorModel(ColorModel)`

Description

Does nothing.

setDimensions

void setDimensions (int width, int height)

Parameters

width

Width for image.

height

Height for image.

Implements

`ImageConsumer.setDimensions(int, int)`

Description

Does nothing.

setHints

void setHints (int hints)

Parameters

hints

Image consumption hints.

Implements

`ImageConsumer.setHints(int)`

Description

Does nothing.

setPixels

void setPixels (int x, int y, int width, int height, ColorModel model, byte[] pixels, int offset, int scansize)

Parameters

x

x-coordinate of top-left corner of pixel data delivered with this method call.

y

y-coordinate of top-left corner of pixel data delivered with this method call.

width

Width of the rectangle of pixel data delivered with this method call.

height

Height of the rectangle of pixel data delivered with this method call.

model

Color model of image data.

pixels

Image data.

offset

Offset from beginning of the pixels array.

scansize

Size of each line of data in pixels array.

Implements

`ImageConsumer.setPixels(int, int, int, int, ColorModel, byte[], int, int)`

Description

Called by the `ImageProducer` to deliver pixel data from the image.

void setPixels (int x, int y, int width, int height, ColorModel model, int[] pixels, int offset, int scansize)

Parameters

x

x-coordinate of top-left corner of pixel data delivered with this method call.

y

y-coordinate of top-left corner of pixel data delivered with this method call.

width

Width of the rectangle of pixel data delivered with this method call.

height

Height of the rectangle of pixel data delivered with this method call.

model

Color model of image data.

pixels

Image data.

offset

Offset from beginning of the pixels array.

scansize

Size of each line of data in pixels array.

Implements

`ImageConsumer.setPixels(int, int, int, int, ColorModel, int[], int, int)`

Description

Called by the `ImageProducer` to deliver pixel data from the image.

setProperties

void setProperties (Hashtable properties)

Parameters

properties

The properties for the image.

Implements

`ImageConsumer.setProperties(Hashtable)`

Description

Does nothing.

startGrabbing

public synchronized void startGrabbing() ★

Description

Starts the `PixelGrabber`'s image-grabbing process.

status

public synchronized int status () ☆

Returns

> The `ImageObserver` flags OR'ed together representing the available information about the image. Replaced by `getStatus()`.

See Also

`ColorModel`, `Hashtable`, `Image`, `ImageConsumer`, `ImageProducer`, `InterruptedException`, `MemoryImageSource`, `Object`

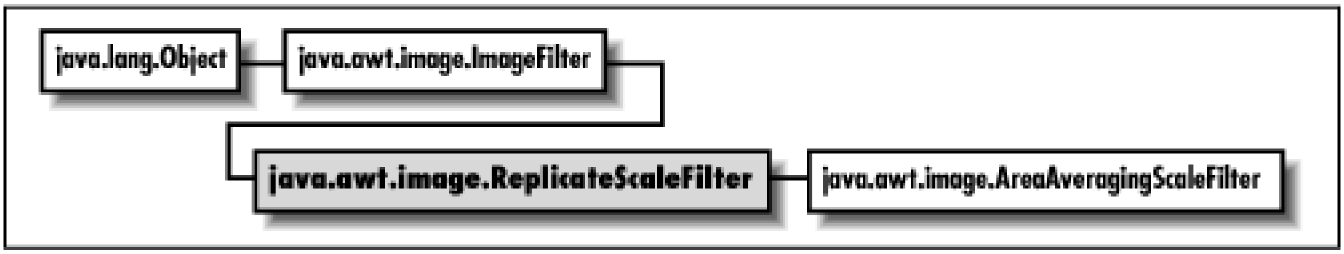
PREVIOUS | HOME | NEXT
MemoryImageSource | BOOK INDEX | ReplicateScaleFilter ★

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

ReplicateScaleFilter ★

Name

ReplicateScaleFilter ★


java.lang.Object
java.awt.image.ImageFilter
java.awt.image.ReplicateScaleFilter
java.awt.image.AreaAveragingScaleFilter

Description

The `ReplicateScaleFilter` class uses a simple-minded algorithm to scale an image. If the image is to be reduced, rows and columns of pixels are removed. If the image is to be expanded, rows and columns are duplicated (replicated).

Class Definition

```
public class ReplicateScaleFilter
    extends java.awt.image.ImageFilter {
  // Variables
  protected int destHeight;
  protected int destWidth;
  protected Object outpixbuf;
  protected int srcHeight;
  protected int srcWidth;
  protected int[] srccols;
  protected int[] srcrows;
  // Constructor
  public ReplicateScaleFilter(int width, int height);
  // Instance Methods
  public void setDimensions (int w, int h);
  public void setPixels(int x, int y, int w, int h, ColorModel model,
      byte[] pixels, int off, int scansize);
  public void setPixels(int x, int y, int w, int h, ColorModel model,
      int[] pixels, int off, int scansize);
  public void setProperties(Hashtable props);
}
```

Variables

destHeight

protected int destHeight

Height of the scaled image.

destWidth

protected int destWidth

Width of the scaled image.

outpixbuf

protected Object outpixbuf

An internal buffer.

srcHeight

protected int srcHeight

Height of the original image.

srcWidth

protected int srcWidth

Width of the original image.

srccols

protected int[] srccols

Internal array used to map incoming columns to outgoing columns.

srcrows

protected int[] srcrows

Internal array used to map incoming rows to outgoing rows.

Constructor

ReplicateScaleFilter

public ReplicateScaleFilter (int width, int height)

Parameters

> *width*
>
> > Width of scaled image.
>
> *height*
>
> > Height of scaled image.

Description

> Constructs a `ReplicateScaleFilter` that scales the original image to the specified size. If both `width` and `height` are -1, the destination image size will be set to the source image size. If either one of the parameters is -1, it will be set to preserve the aspect ratio of the original image.

Instance Methods

setDimensions

public void setDimensions (int w, int h)

Parameters

w

Width of the source image.

h

Height of the source image.

Overrides

`ImageFilter.setDimensions(int, int)`

Description

Sets the size of the source image.

setPixels

void setPixels (int x, int y, int w, int h, ColorModel model, byte[] pixels, int off, int scansize)

Parameters

x

x-coordinate of top-left corner of pixel data delivered with this method call.

y

y-coordinate of top-left corner of pixel data delivered with this method call.

w

Width of the rectangle of pixel data delivered with this method call.

h

Height of the rectangle of pixel data delivered with this method call.

model

Color model of image data.

pixels

Image data.

off

Offset from beginning of the pixels array.

scansize

Size of each line of data in pixels array.

Overrides

`ImageFilter.setPixels(int, int, int, int, ColorModel, byte[], int, int)`

Description

Receives a rectangle of image data from the `ImageProducer`; scales these pixels and delivers them to any `ImageConsumer`s.

void setPixels (int x, int y, int w, int h, ColorModel model, int[] pixels, int off, int scansize)

Parameters

x

x-coordinate of top-left corner of pixel data delivered with this method call.

y

y-coordinate of top-left corner of pixel data delivered with this method call.

w

Width of the rectangle of pixel data delivered with this method call.

h

Height of the rectangle of pixel data delivered with this method call.

model

Color model of image data.

pixels

Image data.

off

Offset from beginning of the pixels array.

scansize

Size of each line of data in pixels array.

Overrides

`ImageFilter.setPixels(int, int, int, int, ColorModel, int[], int, int)`

Description

Receives a rectangle of image data from the `ImageProducer`; scales these pixels and delivers them to any `ImageConsumer`s.

setProperties

public void setProperties (Hashtable props)

Parameters

props

The properties for the image.

Overrides

`ImageFilter.setProperties(Hashtable)`

Description

Adds the "rescale" image property to the properties list.

See Also

`ColorModel, Hashtable, ImageConsumer, ImageFilter, ImageProducer`

PREVIOUS
PixelGrabber

HOME
BOOK INDEX

NEXT
RGBImageFilter

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

RGBImageFilter

Name

RGBImageFilter

[Graphic: Figure from the text]

Description

`RGBImageFilter` is an abstract class that helps you filter images based on each pixel's color and position. In most cases, the only method you need to implement in subclasses is `filterRGB()`, which returns a new pixel value based on the old pixel's color and position. `RGBImageFilter` cannot be used to implement filters that depend on the value of neighboring pixels, or other factors aside from color and position.

Class Definition

```
public abstract class java.awt.image.RGBImageFilter
    extends java.awt.image.ImageFilter {
  // Variables
  protected boolean canFilterIndexColorModel;
  protected ColorModel newmodel;
  protected ColorModel oldmodel;
  // Instance Methods
  public IndexColorModel filterIndexColorModel (IndexColorModel icm);
  public abstract int filterRGB (int x, int y, int rgb);
  public void filterRGBPixels (int x, int y, int width,
      int height, int[] pixels, int off, int scansize);
  public void setColorModel (ColorModel model);
  public void setPixels (int x, int y, int width, int height,
      ColorModel model, byte[] pixels, int offset, int scansize);
  public void setPixels (int x, int y, int width, int height,
      ColorModel model, int[] pixels, int offset, int scansize);
  public void substituteColorModel (ColorModel oldModel,
      ColorModel newModel);
}
```

Variables

canFilterIndexColorModel

protected boolean canFilterIndexColorModel

Setting the `canFilterIndexColorModel` variable to `true` indicates the filter can filter `IndexColorModel` images. To filter an `IndexColorModel`, the filter must depend only on color, not on position.

newmodel

protected ColorModel newmodel

A place to store a new `ColorModel`.

origmodel

protected ColorModel origmodel

A place to store an old `ColorModel`.

Instance Methods

filterIndexColorModel

public IndexColorModel filterIndexColorModel (IndexColorModel icm)

Parameters

icm

Color model to filter.

Returns

Filtered color model.

Description

Helper method for `setColorModel()` that runs the entire color table of `icm` through the `filterRGB()` method of the subclass. Used only if `canFilterIndexColorModel` is `true`, and the image uses an `IndexColorModel`.

filterRGB

public abstract int filterRGB (int x, int y, int rgb)

Parameters

x

x-coordinate of pixel data.

y

y-coordinate of pixel data.

rgb

Color value of pixel to filter.

Returns

New color value of pixel.

Description

Subclasses implement this method to provide a filtering function that generates new pixels.

filterRGBPixels

public void filterRGBPixels (int x, int y, int width, int height, int[] pixels, int off, int scansize)

Parameters

x

x-coordinate of top-left corner of pixel data within entire image.

y

y-coordinate of top-left corner of pixel data within entire image.

width

Width of pixel data within entire image.

height

Height of pixel data within entire image.

pixels

Image data.

off

Offset from beginning of each line in pixels array.

scansize

Size of each line of data in pixels array.

Description

Helper method for `setPixels()` that filters each element of the `pixels` buffer through the subclass's `filterRGB()` method.

setColorModel

public void setColorModel (ColorModel model)

Parameters

model

The color model for the image.

Overrides

`ImageFilter.setColorModel(ColorModel)`

Description

Sets the image's color model.

setPixels

public void setPixels (int x, int y, int width, int height, ColorModel model, byte[] pixels, int offset, int scansize)

Parameters

x

x-coordinate of top-left corner of pixel data delivered with this method call.

y

y-coordinate of top-left corner of pixel data delivered with this method call.

width

Width of the rectangle of pixel data delivered with this method call.

height

Height of the rectangle of pixel data delivered with this method call.

model

Color model of image data.

pixels

Image data.

offset

Offset from beginning of the pixels array.

scansize

Size of each line of data in pixels array.

Overrides

`ImageFilter.setPixels(int, int, int, int, ColorModel, byte[], int, int)`

Description

Called by the `ImageProducer` to deliver a rectangular block of pixels for filtering.

public void setPixels (int x, int y, int width, int height, ColorModel model, int[] pixels, int offset, int scansize)

Parameters

x

x-coordinate of top-left corner of pixel data delivered with this method call.

y

y-coordinate of top-left corner of pixel data delivered with this method call.

width

Width of the rectangle of pixel data delivered with this method call.

height

Height of the rectangle of pixel data delivered with this method call.

model

Color model of image data.

pixels

Image data.

offset

Offset from beginning of the pixels array.

scansize

Size of each line of data in pixels array.

Overrides

`ImageFilter.setPixels(int, int, int, int, ColorModel, int[], int, int)`

Description

Called by the `ImageProducer` to deliver a rectangular block of pixels for filtering.

substituteColorModel

public void substituteColorModel (ColorModel oldModel, ColorModel newModel)

Parameters

oldModel

New value for `origmodel` variable.

newModel

New value for `newmodel` variable.

Description

Helper method for `setColorModel()` to initialize the protected variables `newmodel` and `origmodel`.

See Also

`ColorModel`, `ImageFilter`

PREVIOUS | HOME | NEXT
ReplicateScaleFilter ★ | BOOK INDEX | java.awt.peer Reference

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

CanvasPeer

Name

CanvasPeer

[Graphic: Figure from the text]

Description

`CanvasPeer` is an interface that defines the basis for canvases.

Interface Definition

```
public abstract interface java.awt.peer.CanvasPeer
   extends java.awt.peer.ComponentPeer {
}
```

See Also

`ComponentPeer`

ButtonPeer BOOK INDEX CheckboxMenuItemPeer

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

CheckboxMenuItemPeer

Name

CheckboxMenuItemPeer

[Graphic: Figure from the text]

Description

`CheckboxMenuItemPeer` is an interface that defines the basis for checkbox menu items.

Interface Definition

```
public abstract interface java.awt.peer.CheckboxMenuItemPeer
   extends java.awt.peer.MenuItemPeer {
  // Interface Methods
  public abstract void setState (boolean condition);
}
```

Interface Methods

setState

public abstract void setState (boolean condition)

Parameters

condition

New state for checkbox menu item's peer.

Description

Changes the state of checkbox menu item's peer.

See Also

`MenuComponentPeer`, `MenuItemPeer`

CheckboxPeer

Name

CheckboxPeer

[Graphic: Figure from the text]

Description

`CheckboxPeer` is an interface that defines the basis for checkbox components.

Interface Definition

```
public abstract interface java.awt.peer.CheckboxPeer
   extends java.awt.peer.ComponentPeer {
  // Interface Methods
  public abstract void setCheckboxGroup (CheckboxGroup group);
  public abstract void setLabel (String label);
  public abstract void setState (boolean state);
}
```

Interface Methods

setCheckboxGroup

public abstract void setCheckboxGroup (CheckboxGroup group)

Parameters

group

New group to put the checkbox peer in.

Description

Changes the checkbox group to which the checkbox peer belongs; implicitly removes the peer from its old group, if any.

setLabel

public abstract void setLabel (String label)

Parameters

label

New text for label of checkbox's peer.

Description

Changes the text of the label of the checkbox's peer.

setState

public abstract void setState (boolean state)

Parameters

state

New state for the checkbox's peer.

Description

Changes the state of the checkbox's peer.

See Also

`CheckboxGroup`, `ComponentPeer`, `String`

PREVIOUS HOME NEXT
CheckboxMenuItemPeer BOOK INDEX ChoicePeer

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

ChoicePeer

Name

ChoicePeer

[Graphic: Figure from the text]

Description

`ChoicePeer` is an interface that defines the basis for choice components.

Interface Definition

```
public abstract interface java.awt.peer.ChoicePeer
   extends java.awt.peer.ComponentPeer {
  // Interface Methods
  public abstract void add (String item, int index); ★
  public abstract void addItem (String item, int position); ☆
  public abstract void remove (int index); ★
  public abstract void select (int position);
}
```

Interface Methods

add

public abstract void add (String item, int index) ★

Parameters

item

Text of the entry to add.

index

Position in which to add the entry; position 0 is the first entry in the list.

Description

Adds a new entry to the available choices at the designated position.

addItem

public abstract void addItem (String item, int position) ☆

Parameters

item

Text of the entry to add.

position

Position in which to add the entry; position 0 is the first entry in the list.

Description

Adds a new entry to the available choices at the designated position.

remove

public abstract void remove (int index) ★

Parameters

index

Position of the item to remove.

Description

Removes an entry at the given position.

select

public abstract void select (int position)

Parameters

position

Position to make selected entry.

Description

Makes the given entry the selected one for the choice's peer.

See Also

`ComponentPeer`, `String`

PREVIOUS
CheckboxPeer

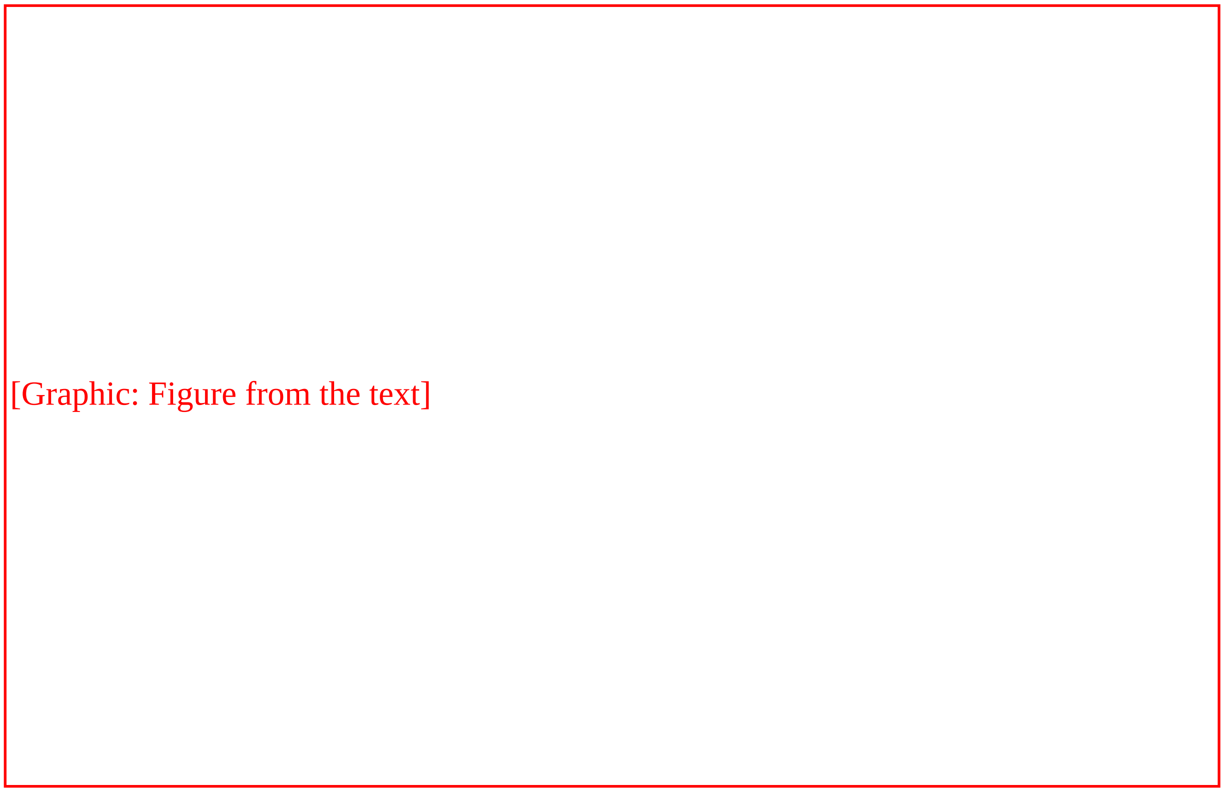
HOME
BOOK INDEX

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

ComponentPeer

Name

ComponentPeer



Description

`ComponentPeer` is an interface that defines the basis for all non-menu GUI peer interfaces.

Interface Definition

```
public abstract interface java.awt.peer.ComponentPeer {
  // Interface Methods
  public abstract int checkImage (Image image, int width, int height,
    ImageObserver observer);
  public abstract Image createImage (ImageProducer producer);
  public abstract Image createImage (int width, int height);
  public abstract void disable(); ☆
  public abstract void dispose();
  public abstract void enable(); ☆
  public abstract ColorModel getColorModel();
  public abstract FontMetrics getFontMetrics (Font f);
  public abstract Graphics getGraphics();
  public abstract Point getLocationOnScreen(); ★
  public abstract Dimension getMinimumSize(); ★
  public abstract Dimension getPreferredSize(); ★
  public abstract Toolkit getToolkit();
  public abstract boolean handleEvent (Event e);
  public abstract void hide(); ☆
  public abstract boolean isFocusTraversable(); ★
  public abstract Dimension minimumSize(); ☆
  public abstract void paint (Graphics g);
  public abstract Dimension preferredSize (); ☆
  public abstract boolean prepareImage (Image image, int width, int height,
    ImageObserver observer);
  public abstract void print (Graphics g);
  public abstract void repaint (long tm, int x, int y, int width, int height);
  public abstract void requestFocus();
  public abstract void reshape (int x, int y, int width, int height); ☆
  public abstract void setBackground (Color c);
  public abstract void setBounds (int x, int y, int width, int height); ★
  public abstract void setCursor (Cursor cursor); ★
  public abstract void setEnabled (boolean b); ★
  public abstract void setFont (Font f);
  public abstract void setForeground (Color c);
  public abstract void setVisible (boolean b); ★
  public abstract void show(); ☆
}
```

Interface Methods

checkImage

public abstract int checkImage (Image image, int width, int height, ImageObserver observer)

Parameters

image

`Image` to check.

width

Horizontal size to which the image will be scaled.

height

Vertical size to which the image will be scaled.

observer

An `ImageObserver` to monitor image loading; normally, the object on which the image will be rendered.

Returns

`ImageObserver` flags ORed together indicating status.

Description

Checks status of image construction.

createImage

public abstract Image createImage (ImageProducer producer)

Parameters

producer

An object that implements the `ImageProducer` interface to create a new image.

Returns

Newly created image instance.

Description

Creates an `Image` based upon an `ImageProducer`.

public abstract Image createImage (int width, int height)

Parameters

width

Horizontal size for in-memory `Image`.

height

Vertical size for in-memory `Image`.

Returns

Newly created image instance.

Description

Creates an in-memory `Image` for double buffering.

disable

public abstract void disable() ☆

Description

Disables component so that it is unresponsive to user interactions. Replaced by `setEnabled(false)`.

dispose

public abstract void dispose()

Description

Releases resources used by peer.

enable

public abstract void enable() ☆

Description

Enables component so that it is responsive to user interactions. Replaced by `setEnabled(true)`.

getColorModel

public abstract ColorModel getColorModel()

Returns

`ColorModel` used to display the current component.

getFontMetrics

public abstract FontMetrics getFontMetrics (Font f)

Parameters

f

A font whose metrics are desired.

Returns

Font sizing information for the desired font.

getGraphics

public abstract Graphics getGraphics()

Throws

InternalException

If acquiring a graphics context is unsupported

Returns

Component's graphics context.

getLocationOnScreen

public abstract Point getLocationOnScreen() ★

Returns

The location of the component in the screen's coordinate space.

getMinimumSize

public abstract Dimension getMinimumSize() ★

Returns

> The minimum dimensions of the component.

getPreferredSize

public abstract Dimension getPreferredSize() ★

Returns

> The preferred dimensions of the component.

getToolkit

public abstract Toolkit getToolkit()

Returns

> `Toolkit` of `Component`.

handleEvent

public abstract boolean handleEvent (Event e)

Parameters

> *e*
>
> > `Event` instance identifying what caused the method to be called.

Returns

> `true` if the peer handled the event, `false` to propagate the event to the parent container.

Description

> High-level event handling routine.

hide

public abstract void hide() ☆

Description

Hides the component. Replaced by `setVisible(false)`.

isFocusTraversable

public abstract boolean isFocusTraversable() ★

Returns

`true` if the peer can be tabbed onto, `false` otherwise.

Description

Determines if this peer is navigable using the keyboard.

minimumSize

public abstract Dimension minimumSize() ☆

Returns

The minimum dimensions of the component. Replaced by `getMinimumSize()`.

paint

public abstract void paint (Graphics g)

Parameters

g

Graphics context of the component.

Description

Draws something in graphics context.

preferredSize

public abstract Dimension preferredSize() ☆

Returns

The preferred dimensions of the component. Replaced by `getPreferredSize()`.

prepareImage

public abstract boolean prepareImage (Image image, int width, int height, ImageObserver observer)

Parameters

image

`Image` to load.

width

Horizontal size to which the image will be scaled.

height

Vertical size to which the image will be scaled.

observer

An `ImageObserver` to monitor image loading; normally, the object on which the image will be rendered.

Returns

`true` if the image has already loaded, `false` otherwise.

Description

Forces the image to start loading.

print

public abstract void print (Graphics g)

Parameters

g

Graphics context of component.

Description

Print something from the graphics context.

repaint

public abstract void repaint (long tm, int x, int y, int width, int height)

Parameters

tm

Millisecond delay allowed before repaint.

x

Horizontal origin of bounding box to redraw.

y

Vertical origin of bounding box to redraw.

width

Width of bounding box to redraw.

height

Height of bounding box to redraw.

Description

Requests scheduler to redraw portion of component within a time period.

requestFocus

public abstract void requestFocus()

Description

Requests this `Component` gets the input focus.

reshape

public abstract void reshape (int x, int y, int width, int height) ☆

Parameters

x

New horizontal position for component.

y

New vertical position for component.

width

New width for component.

height

New height for component.

Description

Relocates and resizes the component's peer. Replaced by `setBounds(int, int, int, int)`.

setBackground

public abstract void setBackground (Color c)

Parameters

c

New color for the background.

Description

Changes the background color of the component.

setBounds

public abstract void setBounds (int x, int y, int width, int height) ★

Parameters

x

New horizontal position for component.

y

New vertical position for component.

width

New width for component.

height

New height for component.

Description

Relocates and resizes the component's peer.

setCursor

public abstract void setCursor (Cursor cursor) ★

Parameters

cursor

New cursor.

Description

Changes the cursor of the component.

setEnabled

public abstract void setEnabled (boolean b) ★

Parameters

b

`true` to enable the peer; `false` to disable it.

Description

Enables or disables the peer.

setFont

public abstract void setFont (Font f)

Parameters

f

New font for the component.

Description

Changes the font used to display text in the component.

setForeground

public abstract void setForeground (Color c)

Parameters

c

New foreground color for the component.

Description

Changes the foreground color of the component.

setVisible

public abstract void setVisible (boolean b) ★

Parameters

b

`true` to show the peer; `false` to hide it.

Description

Shows or hides the peer.

show

public abstract void show() ☆

Description

Makes the peer visible. Replaced by `setVisible(true)`.

See Also

`ButtonPeer`, `CanvasPeer`, `CheckboxPeer`, `ChoicePeer`, `Color`, `ColorModel`, `ContainerPeer`, Cursor, `Dimension`, `Event`, `Font`, `FontMetrics`, `Graphics`, `Image`, `ImageObserver`, `ImageProducer`, `LabelPeer`, `ListPeer`, `ScrollbarPeer`, `TextComponentPeer`, `Toolkit`

PREVIOUS | HOME | NEXT
ChoicePeer | BOOK INDEX | ContainerPeer

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

ContainerPeer

Name

ContainerPeer

[Graphic: Figure from the text]

Description

`ContainerPeer` is an interface that defines the basis for containers.

Interface Definition

```
public abstract interface java.awt.peer.ContainerPeer
   extends java.awt.peer.ComponentPeer {
  // Interface Methods
  public abstract void beginValidate(); ★
  public abstract void endValidate(); ★
  public abstract Insets getInsets(); ★
  public abstract Insets insets(); ☆
}
```

Interface Methods

beginValidate

public abstract void beginValidate() ★

Description

> Notifies the peer that the `Container` is going to validate its contents.

endValidate

public abstract void endValidate() ★

Description

> Notifies the peer that the `Container` is finished validating its contents.

getInsets

public Insets getInsets() ★

Returns

> Current `Insets` of container's peer.

insets

public Insets insets() ☆

Returns

> Current `Insets` of container's peer. Replaced by `getInsets()`.

See Also

ComponentPeer, Insets, PanelPeer, ScrollPanePeer, WindowPeer

PREVIOUS HOME NEXT
ComponentPeer BOOK INDEX DialogPeer

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

DialogPeer

Name

DialogPeer

[Graphic: Figure from the text]

Description

`DialogPeer` is an interface that defines the basis for a dialog box.

Interface Definition

```
public abstract interface java.awt.peer.DialogPeer
   extends java.awt.peer.WindowPeer {
  // Interface Methods
  public abstract void setResizable (boolean resizable);
  public abstract void setTitle (String title);
}
```

Interface Methods

setResizable

public abstract void setResizable (boolean resizable)

Parameters

resizable

`true` if the dialog's peer should allow resizing; `false` to prevent resizing.

Description

Changes the resize state of the dialog's peer.

setTitle

public abstract void setTitle (String title)

Parameters

title

New title for the dialog's peer.

Description

Changes the title of the dialog's peer.

See Also

`FileDialogPeer`, `String`, `WindowPeer`

PREVIOUS HOME NEXT

FileDialogPeer

Name

FileDialogPeer

[Graphic: Figure from the text]

Description

`FileDialogPeer` is an interface that defines the basis for a file dialog box.

Interface Definition

```
public abstract interface java.awt.peer.FileDialogPeer
   extends java.awt.peer.DialogPeer {
  // Interface Methods
  public abstract void setDirectory (String directory);
  public abstract void setFile (String file);
  public abstract void setFilenameFilter (FilenameFilter filter);
}
```

Interface Methods

setDirectory

public abstract void setDirectory (String directory)

Parameters

directory

Initial directory for file dialog's peer.

Description

Changes the directory displayed in the file dialog's peer.

setFile

public abstract void setFile (String file)

Parameters

file

Initial filename for the file dialog's peer.

Description

Changes the default file selection for the file dialog's peer.

setFilenameFilter

public abstract void setFilenameFilter (FilenameFilter filter)

Parameters

filter

Initial filter for file dialog's peer.

Description

Changes the current filename filter of the file dialog's peer.

See Also

`DialogPeer`, `FilenameFilter`, `String`

FontPeer ★

Name

FontPeer ★

[Graphic: Figure from the text]

Description

`FontPeer` is an interface that defines the basis for fonts.

Interface Definition

```
public abstract interface java.awt.peer.FontPeer {
}
```

See Also

`ComponentPeer`

FramePeer

Name

FramePeer

[Graphic: Figure from the text]

Description

`FramePeer` is an interface that defines the basis for a frame.

Interface Definition

```
public abstract interface java.awt.peer.FramePeer
   extends java.awt.peer.WindowPeer {
  // Interface Methods
  public abstract void setIconImage (Image image);
  public abstract void setMenuBar (MenuBar bar);
  public abstract void setResizable (boolean resizable);
  public abstract void setTitle (String title);
}
```

Interface Methods

setIconImage

public abstract void setIconImage (Image image)

Parameters

image

New image to use for frame peer's icon.

Description

Changes the icon associated with the frame's peer.

setMenuBar

public abstract void setMenuBar (MenuBar bar)

Parameters

bar

New `MenuBar` to use for the frame's peer.

Description

Changes the menu bar of the frame.

setResizable

public abstract void setResizable (boolean resizable)

Parameters

resizable

`true` if the frame's peer should allow resizing, `false` to prevent resizing.

Description

Changes the resize state of the frame's peer.

setTitle

public abstract void setTitle (String title)

Parameters

title

New title to use for the frame's peer.

Description

Changes the title of the frame's peer.

See Also

`Image`, `MenuBar`, `String`, `WindowPeer`

PREVIOUS	HOME	NEXT
FontPeer ★	BOOK INDEX	LabelPeer

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

LabelPeer

Name

LabelPeer

[Graphic: Figure from the text]

Description

`LabelPeer` is an interface that defines the basis for label components.

Interface Definition

```
public abstract interface java.awt.peer.LabelPeer
   extends java.awt.peer.ComponentPeer {
  // Interface Methods
  public abstract void setAlignment (int alignment);
  public abstract void setText (String label);
}
```

Interface Methods

setAlignment

public abstract void setAlignment (int alignment)

Parameters

alignment

New alignment for label's peer.

Description

Changes the current alignment of label's peer.

setText

public abstract void setText (String label)

Parameters

label

New text for label's peer.

Description

Changes the current text of label's peer.

See Also

`ComponentPeer`, `String`

PREVIOUS | HOME | NEXT
FramePeer | BOOK INDEX | LightweightPeer ★

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

LightweightPeer ★

Name

LightweightPeer ★

[Graphic: Figure from the text]

Description

`LightweightPeer` is an interface that defines the basis for components that don't have a visual representation. When you directly subclass Component or Container, a `LightweightPeer` is used.

Interface Definition

```
public abstract interface java.awt.peer.LightweightPeer
   extends java.awt.peer.ComponentPeer {
}
```

See Also

`ComponentPeer`

PREVIOUS | HOME | NEXT
LabelPeer | BOOK INDEX | ListPeer

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

ListPeer

Name

ListPeer

[Graphic: Figure from the text]

Description

`ListPeer` is an interface that defines the basis for list components.

Interface Definition

```
public abstract interface java.awt.peer.ListPeer
   extends java.awt.peer.ComponentPeer {
  // Interface Methods
  public abstract void add (String item, int index); ★
  public abstract void addItem (String item, int index); ☆
  public abstract void clear(); ☆
  public abstract void delItems (int start, int end);
  public abstract void deselect (int index);
  public abstract Dimension getMinimumSize (int rows); ★
  public abstract Dimension getPreferredSize (int rows); ★
  public abstract int[] getSelectedIndexes();
  public abstract void makeVisible (int index);
  public abstract Dimension minimumSize (int rows); ☆
  public abstract Dimension preferredSize (int rows); ☆
  public abstract void removeAll(); ★
  public abstract void select (int position);
  public abstract void setMultipleMode (boolean b); ★
  public abstract void setMultipleSelections (boolean value); ☆
}
```

Interface Methods

add

public abstract void add (String item, int index) ★

Parameters

item

Text of an entry to add to the list.

index

Position in which to add the entry; position 0 is the first entry in the list.

Description

Adds a new entry to the available choices of the list's peer at the designated position.

addItem

public abstract void addItem (String item, int index) ☆

Parameters

item

Text of an entry to add to the list.

index

Position in which to add the entry; position 0 is the first entry in the list.

Description

Adds a new entry to the available choices of the list's peer at the designated position. Replaced by `add(String, int)`.

clear

public abstract void clear() ☆

Description

Clears all the entries out of the list's peer. Replaced by `removeAll()`.

delItems

public abstract void delItems (int start, int end)

Parameters

start

Starting position of entries to delete.

end

Ending position of entries to delete.

Description

Removes a set of entries from the list's peer.

deselect

public abstract void deselect (int index)

Parameters

index

Position to deselect.

Description

Deselects entry at designated position, if selected.

getMinimumSize

public abstract Dimension getMinimumSize (int rows) ★

Parameters

rows

Number of rows within list's peer to size.

Returns

The minimum dimensions of a list's peer of the given size.

getPreferredSize

public abstract Dimension getPreferredSize (int rows) ★

Parameters

rows

Number of rows within list's peer to size.

Returns

The preferred dimensions of a list's peer of the given size.

getSelectedIndexes

public abstract int[] getSelectedIndexes()

Returns

> Array of positions of currently selected entries in list's peer.

makeVisible

public abstract void makeVisible (int index)

Parameters

> *index*
>
> > Position to make visible on screen.

Description

> Ensures an item is displayed on the screen in the list's peer.

minimumSize

public abstract Dimension minimumSize (int rows) ☆

Parameters

> *rows*
>
> > Number of rows within list's peer to size.

Returns

> The minimum dimensions of a list's peer of the given size. Replaced by `getMinimumSize(int)`.

preferredSize

public abstract Dimension preferredSize (int rows) ☆

Parameters

rows

Number of rows within list's peer to size.

Returns

The preferred dimensions of a list's peer of the given size. Replaced by `getPreferredSize(int)`.

removeAll

public abstract void removeAll() ★

Description

Clears all the entries out of the list's peer.

select

public abstract void select (int position)

Parameters

position

Position to select; 0 indicates the first item in the list.

Description

Makes the given entry the selected item for the list's peer; deselects other selected entries if multiple selections are not enabled.

setMultipleMode

public abstract void setMultipleMode (boolean value) ★

Parameters

value

`true` to allow multiple selections within the list's peer; `false` to disallow multiple selections.

Description

Changes list peer's selection mode.

setMultipleSelections

public abstract void setMultipleSelections (boolean value) ☆

Parameters

value

`true` to allow multiple selections within the list's peer; `false` to disallow multiple selections.

Description

Changes list peer's selection mode. `Replaced by setMultipleMode(boolean)`.

See Also

`ComponentPeer`, `Dimension`, `String`

PREVIOUS | HOME | NEXT
LightweightPeer ★ | BOOK INDEX | MenuBarPeer

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

MenuBarPeer

Name

MenuBarPeer

[Graphic: Figure from the text]

Description

`MenuBarPeer` is an interface that defines the basis for menu bars.

Interface Definition

```
public abstract interface java.awt.peer.MenuBarPeer
   extends java.awt.peer.MenuComponentPeer {
  // Interface Methods
  public abstract void addHelpMenu (Menu m);
  public abstract void addMenu (Menu m);
  public abstract void delMenu (int index);
}
```

Interface Methods

addHelpMenu

public abstract void addHelpMenu (Menu m)

Parameters

m

Menu to designate as the help menu with the menu bar's peer.

Description

Sets a particular menu to be the help menu of the menu bar's peer.

addMenu

public abstract void addMenu (Menu m)

Parameters

m

Menu to add to the menu bar's peer

Description

Adds a menu to the menu bar's peer.

delMenu

public abstract void delMenu (int index)

Parameters

index

Menu position to delete from the menu bar's peer.

Description

Deletes a menu from the menu bar's peer.

See Also

`Menu`, `MenuComponentPeer`

MenuComponentPeer

Name

MenuComponentPeer

[Graphic: Figure from the text]

Description

`MenuComponentPeer` is an interface that defines the basis for all menu GUI peer interfaces.

Interface Definition

```
public abstract interface java.awt.peer.MenuComponentPeer {
  // Interface Methods
  public abstract void dispose();
}
```

Interface Methods

dispose

public abstract void dispose()

Description

Releases resources used by peer.

See Also

`MenuBarPeer`, `MenuItemPeer`

PREVIOUS MenuBarPeer | HOME BOOK INDEX | NEXT MenuItemPeer

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

MenuItemPeer

Name

MenuItemPeer

[Graphic: Figure from the text]

Description

`MenuBarPeer` is an interface that defines the basis for menu bars.

Interface Definition

```
public abstract interface java.awt.peer.MenuItemPeer
   extends java.awt.peer.MenuComponentPeer {
  // Interface Methods
  public abstract void disable(); ☆
  public abstract void enable(); ☆
  public abstract void setEnabled (boolean b); ★
  public abstract void setLabel (String label);
}
```

Interface Methods

disable

public abstract void disable() ☆

Description

> Disables the menu item's peer so that it is unresponsive to user interactions. Replaced by `setEnabled(false)`.

enable

public abstract void enable() ☆

Description

> Enables the menu item's peer so that it is responsive to user interactions. Replaced by `setEnabled(true)`.

setEnabled

public abstract void setEnabled (boolean b) ★

Parameters

> *b*
>
> > `true` to enable the peer; `false` to disable it.

Description

> Enables or disables the menu item's peer.

setLabel

public abstract void setLabel (String label)

Parameters

label

New text to appear on the menu item's peer.

Description

Changes the label of the menu item's peer.

See Also

`CheckboxMenuItemPeer`, `MenuComponentPeer`, `MenuPeer`, `String`

MenuPeer

Name

MenuPeer

[Graphic: Figure from the text]

Description

`MenuPeer` is an interface that defines the basis for menus.

Interface Definition

```
public abstract interface java.awt.peer.MenuPeer
   extends java.awt.peer.MenuItemPeer {
  // Interface Methods
  public abstract void addItem (MenuItem item);
  public abstract void addSeparator();
  public abstract void delItem (int index);
}
```

Interface Methods

addItem

public abstract void addItem (MenuItem item)

Parameters

item

`MenuItem` to add to the menu's peer

Description

Adds a menu item to the menu's peer.

addSeparator

public abstract void addSeparator()

Description

Adds a menu separator to the menu's peer.

delItem

public abstract void delItem (int index)

Parameters

index

`MenuItem` position to delete from the menu's peer.

Description

Deletes a menu item from the menu's peer.

See Also

MenuItem, MenuItemPeer

PanelPeer

Name

PanelPeer

[Graphic: Figure from the text]

Description

`PanelPeer` is an interface that defines the basis for a panel.

Interface Definition

```
public abstract interface java.awt.peer.PanelPeer
   extends java.awt.peer.ContainerPeer {
}
```

See Also

`ContainerPeer`

PopupMenuPeer ★

Name

PopupMenuPeer ★

[Graphic: Figure from the text]

Description

`PopupMenuPeer` is an interface that defines the basis for a popup menu.

Interface Definition

```
public abstract interface java.awt.peer.PopupMenuPeer
   extends java.awt.peer.MenuPeer {
  // Interface Methods
  public abstract void show (Event e);
}
```

Interface Methods

show

public abstract void show (Event e)

Parameters

e

A mouse down event that begins the display of the popup menu.

Description

Shows the peer at the location encapsulated in `e`.

See Also

`Event, MenuPeer`

ScrollbarPeer

Name

ScrollbarPeer

[Graphic: Figure from the text]

Description

`ScrollbarPeer` is an interface that defines the basis for scrollbar components.

Interface Definition

```
public abstract interface java.awt.peer.ScrollbarPeer
   extends java.awt.peer.ComponentPeer {
  // Interface Methods
  public abstract void setLineIncrement (int amount);
  public abstract void setPageIncrement (int amount);
  public abstract void setValues (int value, int visible, int minimum, int maximum);
}
```

Interface Methods

setLineIncrement

public abstract void setLineIncrement (int amount)

Parameters

amount

New line increment amount.

Description

Changes the line increment amount for the scrollbar's peer.

setPageIncrement

public abstract void setPageIncrement (int amount)

Parameters

amount

New paging increment amount.

Description

Changes the paging increment amount for the scrollbar's peer.

setValues

public abstract void setValues (int value, int visible, int minimum, int maximum)

Parameters

value

New value for the scrollbar's peer.

visible

New slider width.

minimum

New minimum value for the scrollbar's peer.

maximum

New maximum value for the scrollbar's peer.

Description

Changes the settings of the scrollbar's peer to the given amounts.

See Also

```
ComponentPeer
```

ScrollPanePeer ★

Name

ScrollPanePeer ★

[Graphic: Figure from the text]

Description

`ScrollPanePeer` is an interface that defines the basis for a scrolling container.

Interface Definition

```
public abstract interface java.awt.peer.ScrollPanePeer
   extends java.awt.peer.ContainerPeer {
  // Interface Methods
  public abstract void childResized (int w, int h);
  public abstract int getHScrollbarHeight();
  public abstract int getVScrollbarWidth();
  public abstract void setScrollPosition (int x, int y);
  public abstract void setUnitIncrement (Adjustable adj, int u);
  public abstract void setValue (Adjustable adj, int v);
}
```

Interface Methods

childResized

public abstract void childResized (int w, int h)

Parameters

w

The new child width.

h

The new child height.

Description

Tells the peer that the child has a new size.

getHScrollbarHeight

public abstract int getHScrollbarHeight()

Returns

Height that a horizontal scrollbar would occupy.

Description

The height is returned regardless of whether the scrollbar is showing or not.

getVScrollbarWidth

public abstract int getVScrollbarWidth()

Returns

Width that a vertical scrollbar would occupy.

Description

The width is returned regardless of whether the scrollbar is showing or not.

setScrollPosition

public abstract void setScrollPosition (int x, int y)

Parameters

x

The new horizontal position.

y

The new vertical position.

Description

Changes the coordinate of the child component that is displayed at the origin of the `ScrollPanePeer`.

setUnitIncrement

public abstract void setUnitIncrement (Adjustable adj, int u)

Parameters

adj

The `Adjustable` object to change.

u

The new value.

Description

Changes the unit increment of the given `Adjustable` object.

setValue

public abstract void setValue (Adjustable adj, int v)

Parameters

adj

The `Adjustable` object to change.

v

The new value.

Description

Changes the value of the given `Adjustable` object.

See Also

`Adjustable, ContainerPeer, Scrollbar`

PREVIOUS ScrollbarPeer | HOME BOOK INDEX | NEXT TextAreaPeer

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

TextAreaPeer

Name

TextAreaPeer

[Graphic: Figure from the text]

Description

`TextAreaPeer` is an interface that defines the basis for text areas.

Interface Definition

```
public abstract interface java.awt.peer.TextAreaPeer
   extends java.awt.peer.TextComponentPeer {
  // Interface Methods
  public abstract Dimension getMinimumSize (int rows, int columns); ★
  public abstract Dimension getPreferredSize (int rows, int columns); ★
  public abstract void insert (String string, int position); ★
  public abstract void insertText (String string, int position); ☆
  public abstract Dimension minimumSize (int rows, int columns); ☆
  public abstract Dimension preferredSize (int rows, int columns); ☆
  public abstract void replaceRange (String string, int startPosition, int
endPosition); ★
  public abstract void replaceText (String string, int startPosition, int
endPosition); ☆
}
```

Interface Methods

getMinimumSize

public abstract Dimension getMinimumSize (int rows, int columns) ★

Parameters

rows

Number of rows within the text area's peer.

columns

Number of columns within the text area's peer.

Returns

The minimum dimensions of a text area's peer of the given size.

getPreferredSize

public abstract Dimension getPreferredSize (int rows, int columns) ★

Parameters

rows

Number of rows within the text area's peer.

columns

Number of columns within the text area's peer.

Returns

The preferred dimensions of a text area's peer of the given size.

insert

public abstract void insert (String string, int position) ★

Parameters

string

Content to place within the text area's peer.

position

Location at which to insert the content.

Description

Places additional text within the text area's peer.

insertText

public abstract void insertText (String string, int position) ☆

Parameters

string

Content to place within the text area's peer.

position

Location at which to insert the content.

Description

Places additional text within the text area's peer. Replaced by `insert(String, int)`.

minimumSize

public abstract Dimension minimumSize (int rows, int columns) ☆

Parameters

rows

Number of rows within the text area's peer.

columns

Number of columns within the text area's peer.

Returns

The minimum dimensions of a text area's peer of the given size. Replaced by `getMinimumSize(int, int)`.

preferredSize

public abstract Dimension preferredSize (int rows, int columns) ☆

Parameters

rows

Number of rows within the text area's peer.

columns

Number of columns within the text area's peer.

Returns

The preferred dimensions of a text area's peer of the given size. Replaced by `getPreferredSize(int, int)`.

replaceRange

public abstract void replaceRange (String string, int startPosition, int endPosition) ★

Parameters

string

New content to place in the text area's peer.

startPosition

Starting position of the content to replace.

endPosition

Ending position of the content to replace.

Description

Replaces a portion of the text area peer's content with the given text.

replaceText

public abstract void replaceText (String string, int startPosition, int endPosition) ☆

Parameters

string

New content to place in the text area's peer.

startPosition

Starting position of the content to replace.

endPosition

Ending position of the content to replace.

Description

Replaces a portion of the text area peer's content with the given text. Replaced by `replaceRange(String, int, int)`.

See Also

`Dimension`, `String`, `TextComponentPeer`

PREVIOUS | HOME | NEXT
ScrollPanePeer ★ | BOOK INDEX | TextComponentPeer

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

TextComponentPeer

Name

TextComponentPeer

[Graphic: Figure from the text]

Description

`TextComponentPeer` is an interface that defines the basis for text components.

Interface Definition

```
public abstract interface java.awt.peer.TextComponentPeer
   extends java.awt.peer.ComponentPeer {
  // Interface Methods
  public abstract int getCaretPosition(); ★
  public abstract int getSelectionEnd();
  public abstract int getSelectionStart();
  public abstract String getText();
  public abstract void select (int selectionStart, int selectionEnd);
  public abstract void setCaretPosition (int pos); ★
  public abstract void setEditable (boolean state);
  public abstract void setText (String text);
}
```

Interface Methods

getCaretPosition

public abstract int getCaretPosition() ★

Returns

> The current position of the caret (text cursor).

getSelectionEnd

public abstract int getSelectionEnd()

Returns

> The ending cursor position of any selected text.

getSelectionStart

public abstract int getSelectionStart()

Returns

> The initial position of any selected text.

getText

public abstract String getText()

Returns

> The current contents of the text component's peer.

select

public abstract void select (int selectionStart, int selectionEnd)

Parameters

selectionStart

Beginning position of the text to select.

selectionEnd

Ending position of the text to select.

Description

Selects text in the text component's peer.

selectCaretPosition

public abstract void selectCaretPosition (int pos)

Parameters

pos

New caret position.

Description

Changes the position of the caret (text cursor).

setEditable

public abstract void setEditable (boolean state)

Parameters

state

`true` if the user can change the contents of the text component's peer (i.e., `true` to make the peer editable); `false` to make the peer read-only.

Description

Allows you to change the current editable state of the text component's peer.

setText

public abstract void setText (String text)

Parameters

text

New text for the text component's peer .

Description

Sets the content of the text component's peer.

See Also

`ComponentPeer`, `String`, `TextAreaPeer`, `TextFieldPeer`

PREVIOUS HOME NEXT
TextAreaPeer BOOK INDEX TextFieldPeer

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

TextFieldPeer

Name

TextFieldPeer

[Graphic: Figure from the text]

Description

`TextFieldPeer` is an interface that defines the basis for text fields.

Interface Definition

```
public abstract interface java.awt.peer.TextFieldPeer
   extends java.awt.peer.TextComponentPeer {
  // Interface Methods
  public abstract Dimension getMinimumSize (int rows, int columns); ★
  public abstract Dimension getPreferredSize (int rows, int columns); ★
  public abstract Dimension minimumSize (int rows, int columns); ☆
  public abstract Dimension preferredSize (int rows, int columns); ☆
  public abstract void setEchoChar (char echoChar); ★
  public abstract void setEchoCharacter (char c); ☆
}
```

Interface Methods

getMinimumSize

public abstract Dimension getMinimumSize (int rows) ★

Parameters

rows

Number of rows within the text field's peer.

Returns

The minimum dimensions of a text field's peer of the given size.

getPreferredSize

public abstract Dimension getPreferredSize (int rows) ★

Parameters

rows

Number of rows within the text field's peer.

Returns

The preferred dimensions of a text field's peer of the given size.

minimumSize

public abstract Dimension minimumSize (int rows) ☆

Parameters

rows

Number of rows within the text field's peer.

Returns

Replaced by `getMinimumSize(int)`.

preferredSize

public abstract Dimension preferredSize (int rows) ☆

Parameters

rows

Number of rows within the text field's peer.

Returns

Replaced by `getPreferredSize(int)`.

setEchoChar

public abstract void setEchoChar (char c) ★

Parameters

c

The character to display for all input.

Description

Changes the character that is displayed to the user for every character he or she types in the text field.

setEchoCharacter

public abstract void setEchoCharacter (char c) ☆

Parameters

c

The character to display for all input.

Description

Replaced by `setEchoChar(char)`.

See Also

`Dimension`, `TextComponentPeer`

PREVIOUS
TextComponentPeer

HOME
BOOK INDEX

NEXT
WindowPeer

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

WindowPeer

Name

WindowPeer

[Graphic: Figure from the text]

Description

`WindowPeer` is an interface that defines the basis for a window.

Interface Definition

```
public abstract interface java.awt.peer.WindowPeer
   extends java.awt.peer.ContainerPeer {
  // Interface Methods
  public abstract void toBack();
  public abstract void toFront();
}
```

Interface Methods

toBack

public abstract void toBack()

Description

> Puts the window's peer in the background of the display.

toFront

public abstract void toFront()

Description

> Brings the window's peer to the foreground of the display.

See Also

`ContainerPeer`, `DialogPeer`, `FramePeer`

PREVIOUS
TextFieldPeer

HOME
BOOK INDEX

NEXT
Using Properties and Resources

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

C.2 Test Program

The test program, `compList`, listed in Source Code shows the events peers pass along to the Java run-time system. You can then examine the output to see how the run-time system reacts to the different events. When you run `compList`, the screen looks something like the one in Figure C.1.

Figure C.1: Test program

[Graphic: Figure C-1]

How to Use the Program

Java does not have an automated record and playback feature, so the work is left for you to do. The program displays 10 components: `Label`, `Button`, `Scrollbar`, `List`, multiselection `List`, `Choice`, `Checkbox`, `TextField`, `TextArea`, and `Canvas` (the black box in Figure C.1). Basically, you must manually trigger every event for every component.

For *every* component on the screen (except Done), do the following:

With the mouse

Move the cursor over the object, press the mouse button and release, and drag the cursor over the object.

With the keyboard

Press and release an alphabetic key, press and release the Home and End keys, arrow keys, and function keys. Do this for every component, even for components like `Button` and `Label` that have no logical reason for using keyboard events.

For items with choices

Select and deselect a few choices; double-click and single-click selections.

For the scrollbar

Click on each arrow, drag the slider, and click in the paging area (the space between each arrow and the slider).

For the text field

Press Enter.

When finished

Press the Done button, and analyze the results. Run the program again (without exiting), and check the results again. Try to trigger any specific events that you expect but didn't appear in the output from the first pass. Generating some events requires a little work. For example, on a Macintosh, in order to get the `MOUSE_UP` and `MOUSE_DRAG` events, you must do a `MOUSE_DOWN` off the component; otherwise, the `MOUSE_DOWN`/`MOUSE_UP` combination turns into an `ACTION_EVENT`, if that component can generate it.

NOTE:

The SunTest business unit of Sun Microsystems has an early version of a record and playback Java GUI testing tool called JavaSTAR. Information about it is available at http://www.suntest.com/JavaSTAR/JavaSTAR.html. In the future, it may be possible to use JavaSTAR to help automate this process.

Source Code

The following is the source code for the test program:

```
import java.awt.*;
import java.util.*;
import java.applet.*;
public class compList extends Applet {
    Button done = new Button ("Done");
    Hashtable values = new Hashtable();
    public void init () {
        add (new Label ("Label"));
        add (new Button ("Button"));
        add (new Scrollbar (Scrollbar.HORIZONTAL, 50, 25, 0, 255));
        List l1 = new List (3, false);
        l1.addItem ("List 1");
        l1.addItem ("List 2");
        l1.addItem ("List 3");
        l1.addItem ("List 4");
        l1.addItem ("List 5");
        add (l1);
        List l2 = new List (3, true);
        l2.addItem ("Multi 1");
        l2.addItem ("Multi 2");
        l2.addItem ("Multi 3");
        l2.addItem ("Multi 4");
        l2.addItem ("Multi 5");
        add (l2);
        Choice c = new Choice ();
        c.addItem ("Choice 1");
        c.addItem ("Choice 2");
        c.addItem ("Choice 3");
        c.addItem ("Choice 4");
        c.addItem ("Choice 5");
        add (c);
        add (new Checkbox ("Checkbox"));
        add (new TextField ("TextField", 10));
        add (new TextArea ("TextArea", 3, 20));
        Canvas c1 = new Canvas ();
        c1.resize (50, 50);
        c1.setBackground (Color.blue);
        add (c1);
        add (done);
    }
    public boolean handleEvent (Event e) {
        if (e.target == done) {
            if (e.id == Event.ACTION_EVENT) {
                System.out.println (System.getProperty ("java.vendor"));
                System.out.println (System.getProperty ("java.version"));
                System.out.println (System.getProperty ("java.class.version"));
                System.out.println (System.getProperty ("os.name"));
                System.out.println (values);
            }
        }else {
            Vector v;
            Class c = e.target.getClass();
            v = (Vector)values.get(c);
            if (v == null)
                v = new Vector();
            Integer i = new Integer (e.id);
            if (!v.contains (i)) {
                v.addElement (i);
                values.put (c, v);
            }
        }
        return super.handleEvent (e);
    }
}
```

An HTML document to display the applet in a browser should look something like the following:

```
<APPLET code="compList.class" height=300 width=300>
</APPLET>
```

Examining Results

The results of the program are sent to standard output when you click on the Done button. What happens to the output depends on the platform. It may be sent to a log file (Internet Explorer), the Java Console (Netscape Navigator), or the command line (appletviewer). The following is sample output from Internet Explorer 3.0 on a Windows 95 platform.

```
Microsoft Corp.
1.0.2
45.3
Windows 95
{class java.awt.Canvas=[504, 503, 1004, 501, 506, 502, 505, 1005,
401, 402, 403, 404], class java.awt.Choice=[1001, 401, 402, 403,
404], class java.awt.Checkbox=[1001, 402, 401, 403, 404], class
compList=[504, 503, 501, 506, 502, 505, 1004, 1005], class java.
awt.TextField=[401, 402, 403, 404], class java.awt.List=[701,
1001, 401, 402, 403, 404, 702], class java.awt.Scrollbar=[602,
605, 604, 603, 601], class java.awt.TextArea=[401, 402, 403, 404],
class java.awt.Button=[1001, 401, 402, 403, 404]}
```

In addition to some identifying information about the run-time environment, the program displays a list of classes and the events they passed. The integers represent the event constants of the `Event` class; for example, `Canvas` received events with identifiers 504, 503, etc. The events are not sorted, so you can see the order in which they were sent. Unfortunately, you have to look up these constants in the source code yourself. The class listed as `compList` is the applet itself and shows you the events that the `Applet` class receives.

PREVIOUS | HOME | NEXT
The Results | BOOK INDEX | Image Loading

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

D.2 A Brief Tour of sun.awt.image

The classes in `sun.awt.image` do the behind-the-scenes work for rendering an image from a file or across the network. This information is purely for the curious; you should never have to work with these classes yourself.

`Image`

The `Image` class in this package represents a concrete `Image` instance. It contains the basis for the `Image` class that is actually used on the run-time platform, which exists in the package for the specific environment. For instance, the `sun.awt.win32` package includes the `W32Image` (Java 1.0), the `sun.awt.windows` package includes `WImage` (Java 1.1), while the `sun.awt.motif` package includes the `X11Image`, and the `sun.awt.macos` package includes the `MacImage`.

`ImageRepresentation`

The `ImageRepresentation` is the `ImageConsumer` that watches the creation of the image and notifies the `ImageObserver` when it is time to update the display. It plays an important part in the overall control of the `Image` production process.

Image sources

A Java image can come from three different sources: memory (through `createImage()`), local disk, or the network (through `getImage()`).

- `OffScreenImageSource` implements `ImageProducer` for a single framed image in memory. When an `Image` created from an `OffScreenImageSource` is drawn with `drawImage()`, the `ImageObserver` parameter can be `null` since all the image information is already in memory and there is no need for periodic updating as more is retrieved from disk. You can get the graphics context of `OffScreenImageSource` images and use the context to draw on the image area. This is how double buffering works.

- `InputStreamImageSource` implements `ImageProducer` for an image that comes from disk or across the network. When an `Image` created from an `InputStreamImageSource` is drawn with `drawImage()`, the `ImageObserver` parameter should be the component being drawn on (usually `this`) since the image information will be loaded periodically with the help of the `ImageObserver` interface). This class determines how to decode the image type and initializes the `ImageDecoder` to one of `GifImageDecoder`, `JPEGImageDecoder`, or `XbmImageDecoder`, although that can be overridden by a subclass. It can use a `ContentHandler` to work with unknown image types.

- `FileImageSource` is a subclass of `InputStreamImageSource` for images that come from the filesystem. It uses the filename to determine the type of image to decode and checks the security manager to ensure that access is allowed.

- `URLImageSource` is a subclass of `InputStreamImageSource` for images that are specified by a URL.

- `ByteArrayImageSource` (Java 1.1 only) is a subclass of `InputStreamImageSource` for images that are created by calling `Toolkit.createImage(byte[])`.

Image decoders

An `ImageDecoder` is utilized to convert the image source to an image object. If there is no decoder for an image type, it can be read in with the help of a `ContentHandler` or your own class that implements `ImageProducer`, like the `PPMImageDecoder` shown in *Chapter 12, Image Processing*.

- `GifImageDecoder` reads in an image file in the GIF format.

- `JPEGImageDecoder` reads in an image file in the JPEG format.

- `XbmImageDecoder` reads in an image file in the XBM format. Although XBM support is not required by the language specification, support is provided with Netscape Navigator, Internet Explorer, HotJava, and the Java Developer's Kit from Sun.

`ImageFetcher`

The `ImageFetcher` class fetches the actual image from its source. This class creates a separate daemon thread to fetch each image. The thread is run at a higher priority than the default but not at the maximum priority.

PREVIOUS
How Images are Loaded

HOME
BOOK INDEX

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

1.2 New Language Features in Java 1.1

Although Java 1.1 is a massive new release, there are relatively few changes to the Java language in this version. The new features of the language are quite significant, however, as they add useful functionality and make the Java language even more elegant. Here is a brief summary of the new features of the Java language in Java 1.1:

- The addition of inner classes is the largest change to the Java language in Java 1.1. With this new feature, classes can be defined as members of other classes, just like variables and methods. Classes can also be defined within blocks of Java code, just like local variables. A class that is declared inside of another class may have access to the instance variables of the enclosing class; a class declared within a block may have access to the local variables and/or formal parameters of that block.

  Inner classes include: nested top-level classes and interfaces, member classes, local classes, and anonymous classes. The various types of inner clases are described in Inner Classes. The syntax for nested top-level and member classes is covered in Nested Top-Level and Member Classes, while the syntax for nested top-level interfaces is covered in Nested Top-Level Interfaces. The syntax for local classes is described in Local Classes. The syntax for an anonymous class is part of an allocation expression, as covered in Allocation Expressions.

- Java 1.1 provides the ability to declare `final` local variables, method parameters, and `catch` clause parameters. `final` local variables, method parameters, and `catch` parameters are needed to allow local classes to access these entities within the scope of their blocks. The syntax for `final` local variables is described in Local Variables, while `final` method parameters are covered in Method formal parameters. The new syntax for the `catch` clause is described in The try Statement.

- Instance initializers are blocks of code that execute when an instance of a class is created. Instance initializers have been added in Java 1.1 to allow anonymous classes to perform any necessary initialization, since anonymous classes can not define any constructors. The syntax for instance initializers is covered in Instance Initializers.

- As of Java 1.1, `final` variable declarations do not have to include initializers. A `final` variable declaration that does not include an initializer is called a blank final. The functionality of blank finals is described in Variable modifiers and Final local variables.

- A class literal is a new kind of primary expression that can be used to obtain a `Class` object for a particular data type. Class literals have been added to support the new Reflection API in Java 1.1. The syntax for class literals is covered in Class Literals.

- An anonymous array is an array created and initialized without using a variable initializer. The syntax for an anonymous array is part of an allocation expression, as described in Allocation Expressions.

PREVIOUS | HOME | NEXT
A "Hello World" Program | BOOK INDEX | Compiling a Java Source File

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

1.3 Compiling a Java Source File

The interface for the Java compiler in Sun's Java Development Kit (JDK) is the command line. To compile a Java program, run the program *javac* with the name of the source file specified as a command-line argument. For example, to compile the "Hello World" program, issue the following command:

```
C:\> javac HelloWorld.java
```

The Java compiler, *javac*, requires that the name of a Java source file end with a *.java* extension. If the source file contains a class or interface that is declared with the keyword `public`, the filename must be the name of that class or interface. There can be at most one such class or interface in a source file.

In an environment such as Windows 95 that does not distinguish between uppercase and lowercase letters in a filename, you still need to be sure that the case of the filename exactly matches the case used in the `public` class or interface declaration. If you use a filename with the incorrect case, the compiler will be able to compile the file but it will complain about an incorrect filename.

The compiler produces a compiled class file with the same name as the `public` class or interface declaration; the file extension used for a compiled Java file is *.class*.

If the *javac* compiler complains that it is unable to find some classes, it may mean that an environment variable named `CLASSPATH` has not been set properly. The exact setting needed for `CLASSPATH` varies depending on the operating system and its directory structure. However, the value of `CLASSPATH` always specifies a list of directories in which the compiler should search for Java classes.

1.4 Running a Java Application

To run a Java application, you invoke the Java interpreter, *java*, with one or more arguments. The first argument is always the name of a Java class. Here is how to run the "Hello World" application:

```
C:\> java HelloWorld
```

The capitalization of the class name must match the name used in the class declaration in the source file. The interpreter loads the specified class and then calls its `main()` method.

A class can belong to a particular package. This allows the class to prevent classes in other packages from accessing its declared variables and methods. If a class is not specified as part of a package, it automatically becomes part of the default package. Because the `HelloWorld` class is part of the default package, you do not need to include the package name as part of the class name on the command line. If the `HelloWorld` class were part of a package called `student.language`, however, you would have to include the package name on the command line. For example, you would run the application as follows:

```
C:\> java student.language.HelloWorld
```

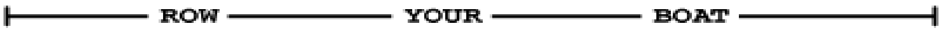
Any additional arguments specified on the command line are passed to the `main()` method in its `String[]` parameter. For the "Hello World" application, the `String[]` parameter is an empty array. If, however, there were command-line arguments, the first array element, `String[0]`, would correspond to the first command-line argument specified after the class name, `String[1]` would correspond to the next command-line element, and so on. The name of the class does not appear as an element in the array of parameters passed to the `main()` method. This is different than in C/C++, where the first element in the array of command-line arguments identifies the program name and the second element is the first command-line argument.

1.5 Notational Conventions

One of the topics of this manual is the *syntax* of Java: the way that identifiers such as `foobar`, operators such as `+`, and punctuation such as `;` can be put together to form a valid Java program. This book also talks about *lexical structure* : the sequences of characters that can be put together to form valid numbers, identifiers, operators, and the like.

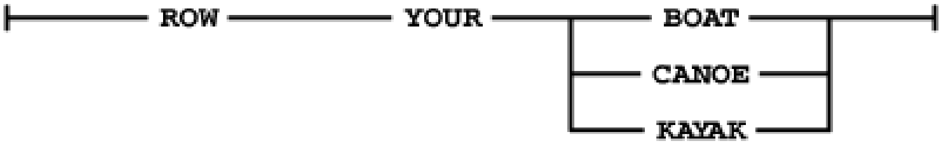
To describe syntax and lexical structure, many language reference manuals use a notation called BNF. BNF notation is very helpful to language implementors because it defines language constructs in a way that can easily be turned into a working language parser. Unfortunately, however, BNF can be difficult for human beings to understand. This reference manual uses a different notation, called a *railroad diagram*, to describe syntax and lexical structure. Railroad diagrams are much easier for people to understand. A railroad diagram provides a visual means of specifying the sequence of words, symbols, and punctuation that can be used to write a syntactic construct or a lexical structure.

Here is a simple example:



The idea is to follow the lines from left to right. The sequence of words or symbols that you pass along the way is the sequence of words or symbols that the railroad diagram specifies. The primary rule when navigating railroad diagrams is that you can follow lines from left to right only, unless there is an arrow pointing to the left. In the above example, there are no arrows, so there is only one way to navigate through the diagram. Therefore, the above railroad diagram specifies exactly one sequence of words: `ROW YOUR BOAT`.

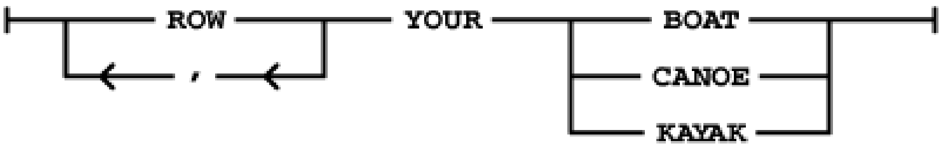
The next example provides you with a choice of sequences:



You can navigate the above diagram with one of three sequences:

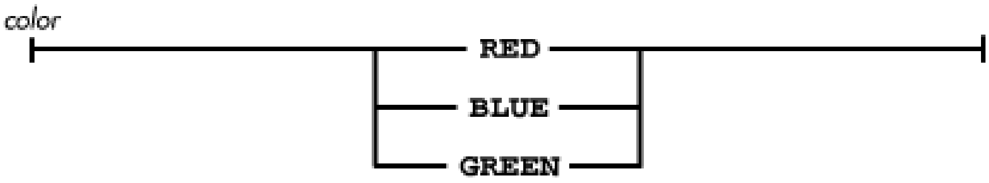
- `ROW YOUR BOAT`
- `ROW YOUR CANOE`
- `ROW YOUR KAYAK`

The following example contains an arrow:

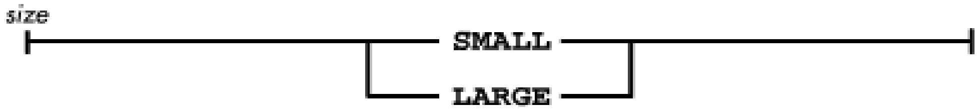


In the above diagram, there is a left-pointing arrow on the line under the word `ROW`. That arrow means that the line can only be traversed from right to left. The line with the arrow provides a loop that allows the word `ROW` to be repeated one or more times, separated by commas. This allows a sequence like: `ROW,ROW,ROW YOUR BOAT`.

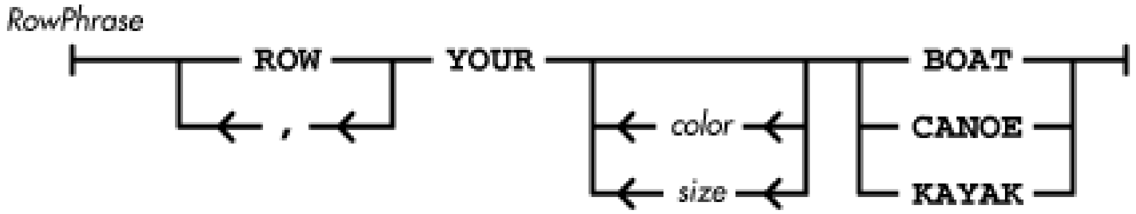
The railroad diagrams shown so far lack a feature that is typically needed to make them useful: a name. A name allows one railroad diagram to refer to another diagram. The following railroad diagram defines a construct named *color* :



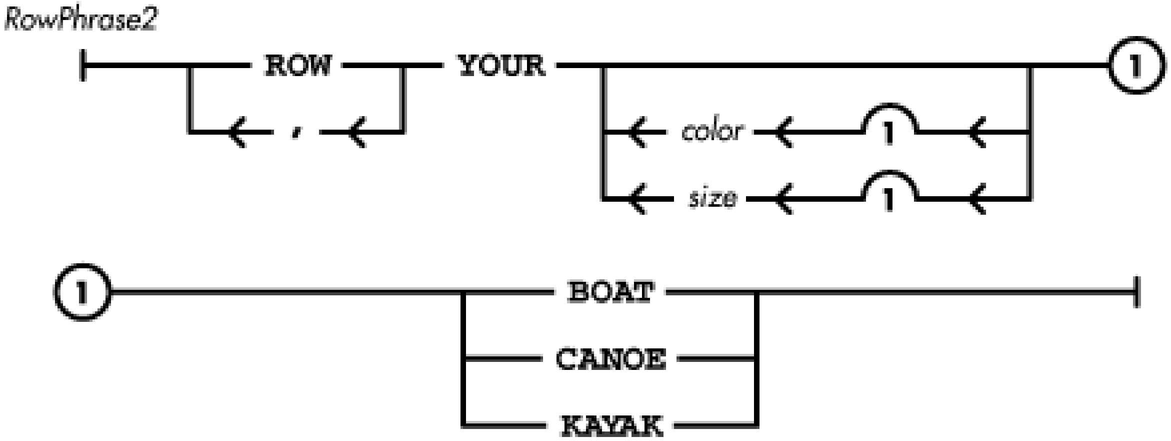
To further illustrate this point, let's look at two more railroad diagrams. The first diagram defines a construct named *size* :



The second railroad diagram is similar to previous ones except that now it allows an optional color or size to precede `BOAT`, `CANOE`, or `KAYAK`. The diagram does this by referring to the names of the railroad diagrams that define these things:



In the diagrams in this book, the font for words such as ROW that are directly contained in railroad diagrams is different from the font used for words like *color* that are names of railroad diagrams. The preceding railroad diagram allows *size* and *color* to occur more than once. The next diagram limits size and color to at most one occurrence:



The lines that refer to the *size* and *color* diagrams both have semi-circles with the number one under them. The semi-circles represent bridges that collapse if crossed more than a certain number of times. The number under the semi-circle is the number of times a bridge can be crossed. Adding bridges that can be crossed only once creates a railroad diagram that permits no more than one occurrence of *color* and *size*.

The other new feature introduced in the above railroad diagram is a circle enclosing a number. These circles are connectors used when a diagram does not fit across a page. The numbered connector at the right end of one part of a railroad diagram attaches to a connector with a matching number at the left end of another part of the railroad diagram.

2.2 Tokenization

The tokenization phase of lexical analysis in Java handles breaking down the lines of Unicode source code into comments, white space, and tokens. The rule that defines the overall lexical organization of Java programs is *TokenStream:*

[Graphic: Figure from the text]

References Comments; Identifiers; Keywords; Literals; Operators; Separators; White Space

Identifiers

An *identifier* is generally used as the name for a thing in a program. A few identifiers are reserved by Java for special uses; these are called *keywords*.

From the viewpoint of lexical analysis, an identifier is a sequence of one or more Unicode characters. The first character must be a letter, underscore, or dollar sign. The other characters must be letters, numbers, underscores, or dollar signs. An identifier can't have the same Unicode character sequence as a keyword:

[Graphic: Figure from the text]

For example, `foo21`, `_foo`, and `$foo` are all valid identifiers; `3foo` is not a valid identifier. There is no limit to the length of an identifier in Java. Although `$` is a legal character in an identifier, you should avoid using it to eliminate confusion with compiler-generated identifiers.

A *UnicodeDigit* is a Unicode character that is classified as a digit by `Character.isDigit()`.

A *UnicodeLetter* is a Unicode character code that is classified as a letter by `Character.isLetter()`.

Two identifiers are the same if they have the same length and if corresponding characters in each identifier have the same Unicode character code. It is possible, however, to have identifiers that are distinct to a Java compiler, but not to the human eye. For example, the Java compiler recognizes lowercase Latin `a' (`\u0061`) and lowercase Cyrillic `a' (`\u0430`) as different characters, although they may well be visually indistinguishable.

References Character; Keywords

Keywords

Keywords are identifiers that have a special meaning to Java. Because of their special meanings, keywords are not available for use as names of things defined in programs. A *Keyword* is one of the following:

```
abstract
boolean
break
byte
case
catch
char
class
default
do
double
else
extends
false
final
finally
goto
if
implements
import
instanceof
int
interface
long
null
package
private
protected
public
return
short
static
synchronized
this
throw
throws
transient
true
try
void
```

```
const    float   native    super    volatile
continue for     new       switch   while
```

The keywords `const` and `goto` are not currently used for any purpose in Java, although they may be assigned meaning in future versions of the Java language.

References Identifiers

Literals

A *literal* is a token that represents a constant value of a primitive data type or a `String` object:

[Graphic: Figure from the text]

References Boolean literals; Character literals; Floating-point literals; Integer literals; String literals

Integer literals

An integer literal represents an integer constant:

[Graphic: Figure from the text]

NonZeroDigit is defined as one of the following characters: `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, or `9`.

OctalDigit is defined as one of the following characters: `0`, `1`, `2`, `3`, `4`, `5`, `6`, or `7`.

Integer literals that begin with a non-zero digit are in base 10 and are called *decimal literals*. Integer literals that begin with `0x` are in base 16 and are called *hexadecimal literals*. Integer literals that begin with `0` followed by `0-7` are in base 8 and are called *octal literals*.

If an integer literal ends with `L` or `l`, its type is `long`; otherwise its type is `int`.

Integer literals cannot begin with a `+` or a `-`. If either of these characters precedes an integer literal, it is treated as a unary operator, a separate token in its own right.

Here are some examples of `int` literals:

```
0
92
0642
0xDeadBeef
```

Here are some examples of `long` literals:

```
0L
1414213562373l
0x2000000000L
075204l
```

Note that the preceding examples end with either an uppercase or lowercase "L". They do not end with the digit 1 (one).

Decimal literals of type `int` may not be greater than 2147483647, which represents 2^31-1. Decimal literals of type `long` may not be greater than 9223372036854775807L, which represents 2^63-1. Decimal literals cannot be used directly to represent negative values. To represent negative values using a decimal literal, you must use the decimal literal in conjunction with the unary minus operator. For example, representing -321 requires the use of a unary minus and a decimal literal. To represent the `int` -2147483648, use `0x80000000`. To represent the `long` -9223372036854775808L, use `0x8000000000000000L`.

Hexadecimal and octal literals may be positive or negative because they represent either a 32-bit (`int`) or 64-bit (`long`) two's-complement quantity. Two's complement is a binary encoding technique that represents both positive and negative values. The range of values that can be represented by `int` hexadecimal and octal literals is shown in Table 2-1.

Table 2.1: Minimum and Maximum int Literals

Representation	**Minimum Value**	**Maximum Value**
Hexadecimal	`0x80000000`	`0x7fffffff`
Octal	`020000000000`	`017777777777`
Base 10 equivalent	`-2147483648`	`2147483647`

The range of values that can be represented by `long` hexadecimal and octal literals is shown in Table 2-2.

Table 2.2: Minimum and Maximum long Literals

Representation	**Minimum Value**	**Maximum Value**
Hexadecimal	`0x8000000000000000L`	`0x7fffffffffffffffL`
Octal	`01000000000000000000000L`	`0777777777777777777777L`
Base 10 equivalent	`-9223372036854775808`	`9223372036854775807`

References **UNKNOWN XREF**; **UNKNOWN XREF**; Integer types; Conversion to Unicode; Unary Operators

Floating-point literals

A floating-point literal represents a constant value of type `float` or `double` :

[Graphic: Figure from the text]

A floating-point literal must minimally contain at least one digit and either a decimal point or an exponent.

The data type of a floating-point literal is `float` if and only if the suffix `f` or `F` appears at the end of the literal. If there is no suffix or the suffix is `d` or `D`, the data type is `double`.

Floating-point literals cannot begin with a `+` or a `-`. If either of these precedes a floating-point literal, it is treated as a separate token, a unary operator.

Here are some examples of `float` literals:

```
23e4f
1.E2f
.31416e1F
2.717f
7.63e+9f
```

Here are some examples of `double` literals:

```
23e4
1.E2
.31415e1D
2.717
7.53e+9d
```

The ranges of values that can be represented by `float` and `double` literals are shown in Table 2-3.

Table 2.3: Minimum and Maximum Floating-Point Literals

Representation	**Minimum Value**	**Maximum Value**
float	`1.40239846e-45f`	`3.40282347e38f`
double	`4.94065645841246544e-324`	`1.79769313486231570e308`

Floating-point literals that exceed these limits are treated as errors by the Java compiler. The special floating-point values positive infinity, negative infinity, and not-a-number are available as predefined constants in Java, as part of the `Float` and `Double` classes.

References **UNKNOWN XREF**; Floating-point types; Unary Operators; Double; Float

Boolean literals

There are two `boolean` literal values, represented by the keywords `true` and `false:`

[Graphic: Figure from the text]

References Boolean Type

Character literals

A character literal represents a constant value of type `char` (an unsigned 16-bit quantity). A character literal consists of either the character being represented, or an equivalent escape sequence, enclosed in single quotes:

[Graphic: Figure from the text]

[Graphic: Figure from the text]

Here are some examples of character literals:

```
'c'
'n'
'\\'
'\u0138'
```

The character sequence `\u`*xxxx* is not defined above as a valid *Escape*, even though it can be used as a legal character literal. This sequence of characters is defined as an *EscapedSourceCharacter*, which is handled during the pre-processing phase, before tokenization takes place. As a result, the tokenization phase never sees an *EscapedSourceCharacter*. Tokenization sees only the single Unicode character that replaces the *EscapedSourceCharacter* during pre-processing.

The translations of the different types of escape sequences supported in Java are shown in Table 2-4.

Table 2.4: Java Escape Sequences

Escape Sequence	**Unicode Equivalent**	**Meaning**
`\b`	`\u0008`	Backspace
`\t`	`\u0009`	Horizontal tab
`\n`	`\u000a`	Linefeed
`\f`	`\u000c`	Form feed
`\r`	`\u000d`	Carriage return
`\"`	`\u0022`	Double quote

\'	\u0027	Single quote
\\	\u005c	Backslash
\xxx	\u0000 to \u00ff	The character corresponding to the octal value *xxx*

A character literal representing a carriage return character can be written only as '\r'; a character literal representing a linefeed character can be written only as '\n'. During the pre-processing that precedes token recognition, these characters are classified as line terminators, so neither carriage return (\u000d) nor linefeed (\u000a) characters in Java source code can ever be seen by the Java compiler as being part of a character literal.

If a backslash that is not part of a legal *Escape* appears in a character literal, it is flagged as an error. This is different from languages like C++ that ignore backslashes in character literals that are not part of an escape.

References Conversion to Unicode; Integer types; **UNKNOWN XREF**

String literals

A string literal represents a constant string value and consists of the characters in the string or the equivalent escapes:

[Graphic: Figure from the text]

Here are some examples of string literals:

```
""                           // the empty string
"Hello World"
"This has \"escapes\"\n"     // a string literal with escapes
```

There is no primitive type for representing strings in Java. Instead, each string literal becomes a reference to a String object. If two or more string literals consist of the same sequence of characters, they refer to the same String object. Using one String object to represent multiple string literals works because, once created, the contents of a String object cannot be changed.

For a string literal to contain a carriage return or linefeed character, the carriage return or linefeed must be written as \r or \n. Neither carriage return (\u000d) nor linefeed (\u000a) characters in Java source code can ever be seen by the Java compiler as part of a string literal. These characters are classified as line terminators during the pre-processing phase that precedes token recognition. For the same reason, `\u` Unicode escapes for carriage return and linefeed characters cannot be directly used in string literals.

If a backslash that is not part of a legal *Escape* appears in a string literal it is flagged as an error. This is different from languages like C++ that ignore backslashes in string literals that are not part of an escape.

Because operations on strings are generally based on the length of the string, Java does not automatically supply a NUL character (`\u0000`) at the end of a string literal. For the same reason, it is not customary for Java programs to put a NUL character at the end of a string.

References *Escape* 2.2.3.4; Specially supported classes; String; StringBuffer; String Concatenation Operator +

Separators

A *separator* is any one of the punctuation tokens in the following railroad diagram:

[Graphic: Figure from the text]

Separator tokens are used to separate other types of tokens. Thus, separators are a part of a higher-level syntactic construct. Although separators have syntactic significance, they do not imply any operation on data.

Operators

An operator is a token that implies an operation on data. Java has both assignment and non-assignment operators:

[Graphic: Figure from the text]

A *NonAssignmentOperator* is one of the following:

```
+ - <= ^   ++
< * >= %   --
  / != ?   >>
! & == :   >>
~ | && >>>
```

An *AssignmentOperator* is one of the following:

```
=   -=  *=
/=  |=  &=
^=  +=  %=
<<= >>= >>>=
```

Unlike C/C++, Java does not have a comma operator. Java does allow a comma to be used as a separator in the header portion of `for` statements, however. Java also omits a number of other operators found in C and C++. Most notably, Java does not include operators for accessing physical memory as an array of bytes, such as `sizeof`, `unary & (address of)`, `unary * (contents of)`, or `-> (contents of field)`.

Comments

Java supports three styles of comments:

- A standard C-style comment, where all of the characters between `/*` and `*/` are ignored.
- A single-line comment, where all of the characters from `//` to the end of the line are ignored.
- A documentation comment that begins with `/**` and ends with `*/`. These comments are similar to standard C-style comments, but the contents of a documentation comment can be extracted to produce automatically generated documentation.

The formal definition of a comment is:

[Graphic: Figure from the text]

C-style comments and documentation comments do not nest. For example, consider the following arrangement of comments:

```
/*   ...   /*   ...   */   ...   */
```

The Java compiler interprets the first `*/` to be the end of the comment, so that what follows is a syntax error.

However, in a single-line comment (i.e., one that starts with `//`), the sequences `/*`, `/**`, and `*/` have no special meaning. Similarly, in a C-style comment or a documentation comment (i.e., comments that begin with `/*` or `/**`), the sequence `//` has no special meaning.

In order to comment out large chunks of code, you need to adopt a commenting style. The C/C++ practice of using `#if` to comment out multiple lines of code is not available for Java programs because Java does not have a conditional compilation mechanism. If you use C-style comments in your code, you'll need to use the `//` style of comment to comment out multiple lines of code:

```
///*
// * Prevent instantiation of RomanNumeral objects without
// * parameters.
// */
//    private RomanNumeral() {
//        super();
//    }
```

The `/*  */` style of comment cannot be used to comment out the lines in the above example because the example already contains that style of comment, and these comments do not nest.

If, however, you stick to using the `//` style of comment in your code, you can use C-style comments to comment out large blocks of code:

```
/*
```

```
*// Prevent instantiation of RomanNumeral objects without
*// parameters.
*    private RomanNumeral() {
*        super();
*     }
*/
```

Which style you choose is less important than using it consistently, so that you avoid inadvertently nesting comments in illegal ways.

References Documentation Comments; Division of the Input Stream into Lines

White Space

White space denotes characters such as space, tab, and form feed that do not have corresponding glyphs, but alter the position of following glyphs. White space and comments are discarded. The purpose of white space is to separate tokens from each other:

[Graphic: Figure from the text]

SpaceCharacter is equivalent to `\u0020`.

HorizontalTabCharacter is equivalent to `\u0009` or `\t`.

FormFeedCharacter is equivalent to `\u000C` or `\f`.

EndOf FileMarker is defined as `\u001A`. Also known as Control-Z, this is the last character in a pre-processed compilation unit. It is treated as white space if it is the last character in a file, to enhance compatibility with older MS-DOS programs and other operating environments that recognize `\u001A` as an end-of-file marker.

References Division of the Input Stream into Lines

PREVIOUS HOME NEXT

3.2 Reference Types

Java is an object-oriented language. An object is a collection of variables and associated methods that is described by a class. The concepts in this section that relate to objects are discussed in detail in Object-Orientation Java Style.

The name of a class can be used as a type, so you can declare an object-type variable or specify that a method returns an object. If you declare a variable using the name of a class for its type, that variable can contain a *reference* to an object of that class. Such a variable does not contain an actual object, but rather a reference to the class instance, or object, the variable refers to. Because using a class name as a type declares a reference to an object, such types are called *reference types*. Java also allows the use of an interface name to specify a reference type. In addition, array types in Java are reference types because Java treats arrays as objects.

The two main characteristics of objects in Java are that:

- Objects are always dynamically allocated. The lifetime of the storage occupied by an object is determined by the program's logic, not by the lifetime of a procedure call or the boundaries of a block. The lifetime of the storage occupied by an object refers to the span of time that begins when the object is created and ends at the earliest time it can be freed by the garbage collector.

- Objects are not contained by variables. Instead, a variable contains a reference to an object. A reference is similar to what is called a pointer in other languages. If there are two variables of the same reference type and one variable is assigned to the other, both variables refer to the same object. If the information in that object is changed, the change is visible through both variables.

Java references are very similar to pointers in C/C++, but they are not at all related to the C++ notion of a reference. The main difference between Java references and C++ pointers is that Java does not allow any arithmetic to be done with references. This, coupled with Java's lack of any way to explicitly deallocate the storage used by reference type values, guarantees that a reference can never point to an illegal address.

The formal definition of a reference type is:

[Graphic: Figure from the text]

It is possible to cause a reference variable to contain a reference to nothing by assigning the special value represented by the keyword `null` to the variable. The value `null` can be assigned to any reference variable without a type cast.

Java does not allow reference types to be cast to primitive data types or primitive data types to be type cast to reference types. In particular, unlike C/C++, there is no conversion between integer values and references.

The only operation that Java provides for reference-type variables is the ability to fetch the referenced object. However, Java does not provide an operator to fetch the object referenced by a reference variable. Instead, the object fetch operation is performed implicitly by the following operations:

- A field expression that accesses a variable or method of a class or interface object
- A field expression that accesses an element of an array object
- A type comparison operation that uses the `instanceof` operator

References Array Types; *ClassOrInterfaceName* 4.1.6; Class Types; Field Expressions; Interface Types; null; Object-Orientation Java Style; Primitive Types; The instanceof Operator

Class Types

The name of a class can be used to specify the type of a reference. If a variable is declared as a class type, the variable either contains `null` or a reference to an object of that class or a subclass of that class. It is not allowed to contain any other kinds of values. For example:

```
class Shape { ... }
class Triangle extends Shape { ... }
...
Shape s;
Triangle t;
...
s = t;
```

This example declares a class called `Shape` and a subclass of `Shape` called `Triangle`. The code later declares a reference variable called `s` that can contain a reference to a `Shape` object and another variable called `t` that can contain a reference to a `Triangle` object. The value of `s` can be assigned to the value of `t` because an object is not only an instance of its declared class, but also an instance of every superclass of its declared class. Since instances of the `Triangle` class are also instances of its superclass `Shape`, the Java compiler has no problem with `s = t`.

However, saying `t = s` generates an error message from the compiler. Java does not allow a reference variable declared as a class type to contain a reference to a superclass of the declared class. The assignment `t = s` is illegal because `Shape` is a superclass of `Triangle`. The assignment can be accomplished if `s` is first cast to a reference to `Triangle`:

```
t = (Triangle)s;
```

The cast operation ensures that the object referenced by `s` is a class type that is either `Triangle` or a descendant of `Triangle`. When you cast an object reference to a subclass of the reference type, you are saying that you want to treat the object being referenced as an instance of the specified subclass. If the compiler cannot determine whether the argument of a cast will be of the required type, the compiler generates runtime code that ensures that the argument really is an instance of the specified subclass. At runtime, if the class of the object being referenced is not an instance of the specified subclass, a `ClassCastException` is thrown.

References Casts; Classes; Class Declarations; Object Allocation Expressions; Runtime exceptions

Specially supported classes

Java provides special support for the `String` and `StringBuffer` classes. All string literals are compiled into `String` objects. The result of a string concatenation operation is a `String` object. An intermediate `StringBuffer` object is used to compute the result of a concatenation operation. Because operations on strings are generally based on the length of the string, Java does not automatically supply a NUL character (`\u0000`) at the end of a string literal. For the same reason, it is not customary for Java programs to put a NUL character at the end of a string.

Java also provides special support for the `Object` class. This class is the ultimate superclass of all other classes in Java. If a class is declared without its superclass being specified, the language automatically specifies `Object` as its superclass.

The `throw` statement in Java is special, in that it requires the use of a `Throwable` object.

References Object; String; StringBuffer; String Concatenation Operator +; String literals; The throw Statement; Throwable

Interface Types

The name of an interface can be used to specify the type of a reference. A reference variable declared using an interface name as its type can only reference instances of classes that implement that interface. For example, Java provides an interface called `Runnable`. Java also provides a class called `Thread` that implements `Runnable`. This means that the following assignment is allowed:

```
Runnable r;
r = new Thread();
```

The Java compiler does not allow a value to be assigned to a variable declared using an interface type unless the compiler can be sure that the object referenced by the value implements the specified interface. Casting a reference variable to an interface type allows the variable to be assigned to that interface type, because the cast operation provides its own guarantee that the object implements the specified interface. Unless the compiler is able to determine the actual class of the object that will be referenced at runtime, the cast produces code that verifies at runtime that the object being cast really does implement the specified interface. At runtime, if the object being cast does not implement the required interface, a `ClassCastException` is thrown.

References Casts; Interfaces; Interface Declarations; Object Allocation Expressions; Runtime exceptions

Array Types

An array is a special kind of object that contains values called *elements*. Array elements are similar to variables in that they contain values that can be used in expressions and set by assignment operations. Elements differ from variables, however, in that they do not have names. Instead, they are identified by non-negative integers. The elements in an array are identified by a contiguous range of integers from 0 to one less than the number of elements in the array. The elements of an array must all contain the same type of value; the type of the array is specified when the array is created.

An array-type variable is declared as follows:

```
int [] a;
```

This declaration specifies that the variable `a` refers to an array of `int` values. Java actually allows two styles of array declarations: the one shown above and a style that is more like that used in C/C++. In other words, you can put the square brackets after the variable name instead of after the type:

```
int a[];
```

Technically, all arrays in Java are one-dimensional. However, Java does allow you to declare an array of arrays, which is a more flexible data structure than a multi-dimensional array. The additional flexibility comes from the fact that the arrays in an array of arrays do not have to be the same length. Because arrays of arrays are typically used to represent multi-dimensional arrays, this book refers to them as multi-dimensional arrays, even though that is not precisely correct.

A multi-dimensional array is declared using multiple pairs of square brackets, as in the following examples:

```
int [][] d2;       // Refers to a 2-dimensional array
int [][][] d3;     // Refers to a 3-dimensional array
```

When you declare a variable to refer to a multi-dimensional array, the number of dimensions in the array is determined by the number of pairs of square brackets. Whether the brackets follow the type name or the variable name is not important. Thus, the above variables could have been declared like this:

```
int [] d2[],       // Refers to a 2-dimensional array
       d3[][];     // Refers to a 3-dimensional array
```

The actual length of each dimension of an array object is specified when the array object is created, not when the array variable is declared. An array object is not created at the same time that an array variable is declared. An array object is created with the `new` operator. Here are some examples:

```
int j[] = new int[10];          // An array of 10 ints
int k[][] = new float[3][4];  // An array of 3 arrays of 4 floats
```

The arrays contained in an array of arrays can also be of different lengths:

```
int a[][] = new int [3][];
a[0] = new int [5];
a[1] = new int [6];
a[2] = new int [7];
```

Although the first `new` operator is creating a two-dimensional array, only the length of one dimension is specified. In this case, just the array of arrays is created. The subarrays are created by the subsequent `new` operators.

The expression used to specify the length of an array does not have to be a constant. Consider the following example:

```
int[] countArray(int n){
    int[] a = new int[n];
    for (int i=0; i<n; i++) {
        a[i]=i+1;
    }
    return a;
}
```

The number of elements in an array object is fixed at the time that the array object is created and cannot be changed.[2] Every array object has a public variable called `length` that contains the number of elements in the array. The variable `length` is `final`, which means that its value cannot be changed by assignment.

> [2] The standard class `java.util.Vector` implements an array-like object with a length that can be changed.

The Java notion of arrays is fundamentally different than that of C/C++. Subscripting a Java array does not imply pointer arithmetic, so there is no danger of an out-of-range index accessing memory that shouldn't be accessed. Array objects in Java detect out-of-range subscripts; when they do they throw an `ArrayIndexOutOfBoundsException`. And unlike C/C++, arrays of type `char` are not strings in Java. Instead, Java uses the `String` class to support strings.

Although array objects are reference types, array objects are different from other kinds of objects. The `Object` class is the parent class of all array objects, but array objects do not really belong to a class of their own. An array object inherits all of the variables and methods of the `Object` class. Every array also defines the variable `length`, but there is no class declaration for an array type.

References Variable initializers; Array Allocation Expressions; Index Expressions; Object; String

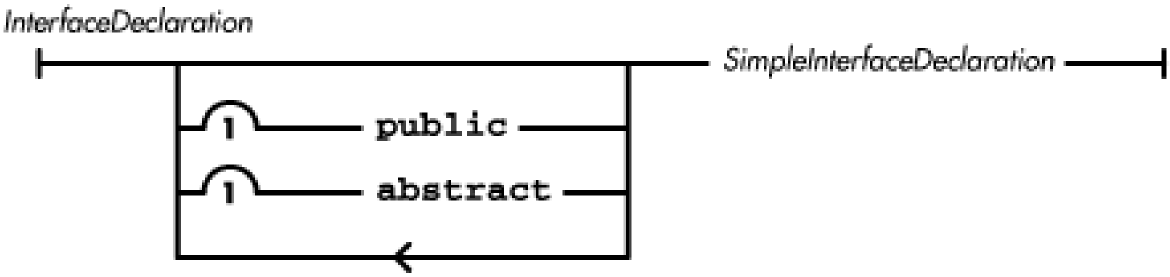
5.5 Interface Declarations

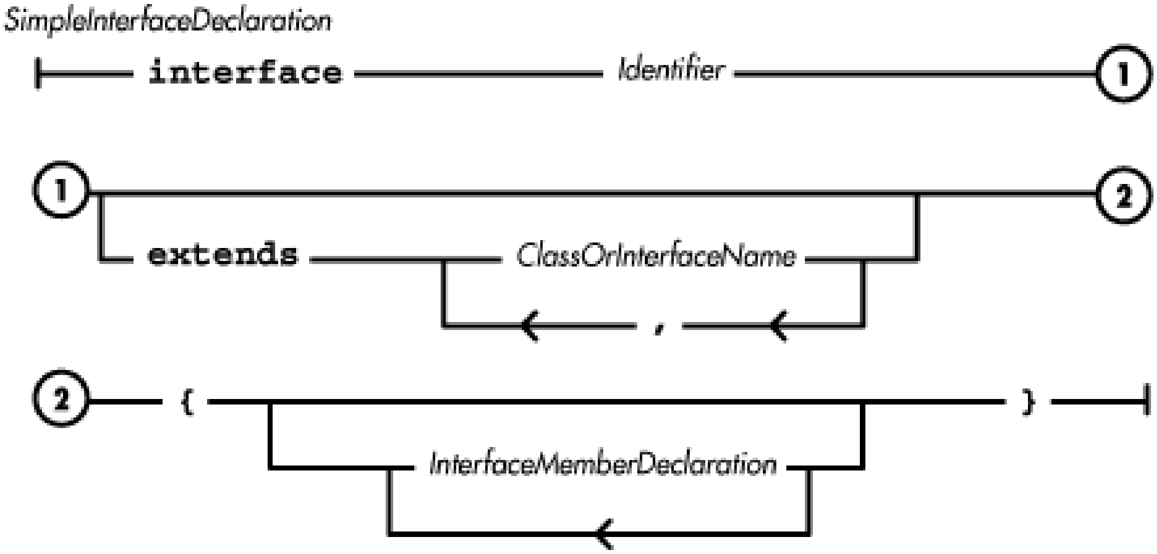
An interface declaration creates a reference type in Java. An interface declaration is similar to a class declaration, with the following two very important differences.

- All of the methods in an interface are implicitly `abstract`. Every method declaration in an interface specifies the formal parameters and return type of the method, but it does not include an implementation of the method.
- All of the variables in an interface are implicitly `static` and `final`.

Interfaces are most useful for declaring that an otherwise unrelated set of classes have a common set of methods. For example, if you want to store a variety of objects in a database, you might want all of those objects to have fetch and store methods. The fetch and store methods of each object require different implementations, so it makes sense to declare the fetch and store methods in an interface declaration. Then any class that needs fetch and store methods can implement the interface.

The formal definition for an interface declaration is:





While the above diagram may seem complicated, an interface declaration is really made up of five distinct things:

- Optional modifiers that specify attributes of the class
- The keyword `interface`
- An identifier that names the interface
- An optional `extends` clause that specifies the super interfaces of the declared interface
- Any number of interface member declarations, which can include variables and methods

Here are some sample interface declarations:

```
interface Dyn {
    double squeeze();
}
interface Press extends Dyn {
    double squeeze(double theta);
}
```

Here is an example of a class that implements `Press`:

```
class Clamp implements Press {
      ...
    double squeeze() {
        return squeeze(0);
    }
    double squeeze(double theta) {
        return force*Math.cos(theta);
    }
      ...
}
```

Since the `Press` interface extends the `Dyn` interface, the `Clamp` class must implement the methods declared in both `Dyn` and `Press`.

References Class Declarations; *ClassOrInterfaceName* 4.1.6; Identifiers; Interfaces; Interface Members

Interface Modifiers

The keywords `public` and `abstract` can appear as modifiers at the beginning of an interface declaration. In this situation, these modifiers have the following meanings:

`public`

> If an interface is declared `public`, it can be referenced by any class or interface. If the `public` modifier is not used, however, the interface can only be referenced by classes and interfaces in the same package. A single source file, or compilation unit, can only declare one `public` class or interface (see Compilation Units for an exception to this rule).

`abstract`

> An interface is implicitly `abstract`; so all of the methods in an interface are implicitly `abstract`. Including the `abstract` modifier in an interface declaration is permitted, but it does not change the meaning of the interface declaration.

References Compilation Units; Inner interface modifiers; Interface method modifiers; Interface variable modifiers

Interface Name

The identifier that follows the keyword `interface` is the name of the interface. This identifier can be used as a reference type wherever the interface is accessible.

References Interface Types

Interface Inheritance

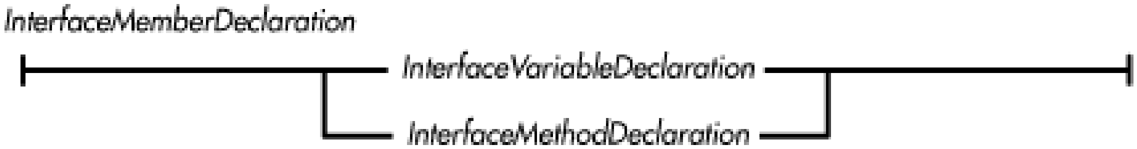
The extends clause specifies any super-interfaces of the interface being declared; the extends keyword can be followed by the names of one or more interfaces. If an interface has an extends clause, the clause can only name other interfaces.

Including an interface in the extends clause of another interface means that the declared interface inherits the variables and methods declared in the super-interface. A class that implements the declared interface must implement all of the methods in the declared interface, as well as all of the methods inherited from the super-interface.

If an interface declaration does not include an extends clause, the interface does not extend any other interfaces.

Interface Members

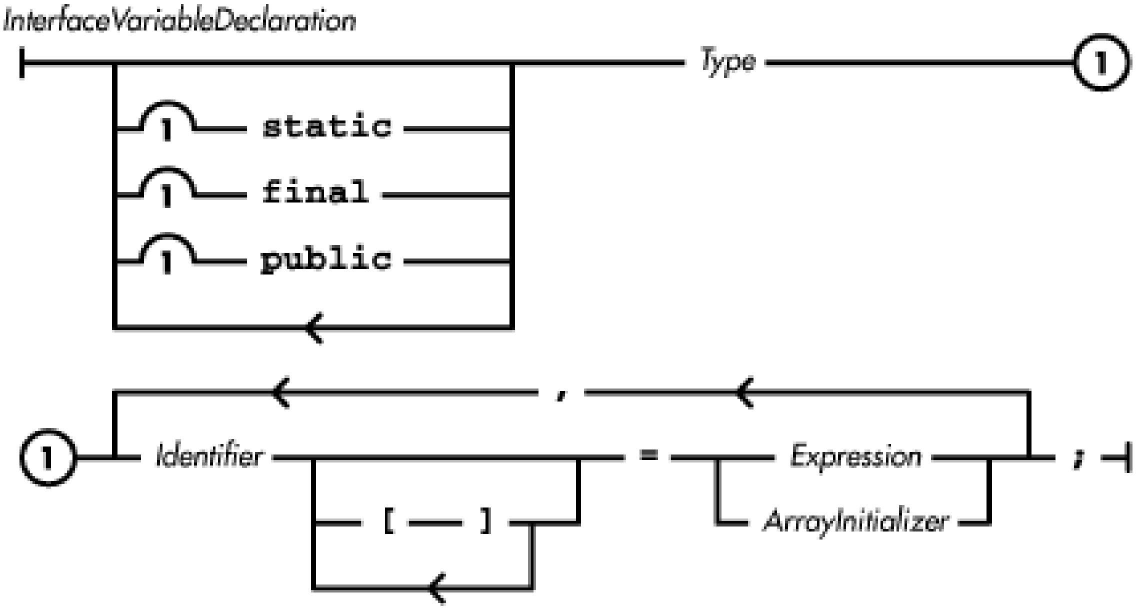
The members of an interface can be variables or methods; an interface cannot have constructors, static initializers, instance initializers, nested top-level classes or interfaces, or member classes:


InterfaceMemberDeclaration
InterfaceVariableDeclaration
InterfaceMethodDeclaration

References Interface Methods; Interface Variables

Interface Variables

Any field variables declared in an interface are implicitly static and final. In other words, field variables in an interface are named constants. Every field variable declaration in an interface must contain an initializer that sets the value of the named constant:



A variable declaration in an interface is made up of three distinct things:

- Optional modifiers that specify attributes of the variable.
- A type, which can be either a primitive type or a reference type.
- Any number of identifiers that name variables. Each name must be followed by an initializer that sets the value of the constant.

References Variable initializers; *Expression* 4; Identifiers; *Type* 3

Interface variable modifiers

Variables in an interface are implicitly `static` and `final`. Including these modifiers in a variable declaration is permitted, but it is not necessary and it does not change the meaning of the variable declaration. Thus, by definition, all variables in an interface are named constants.

If an interface is declared `public`, a field variable declared in the interface is `public`, even if it is declared with the `private` or `protected` modifier. If an interface is not declared `public`, however, any field variables in the interface have the default accessibility, which means that they are only accessible in classes and interfaces in the same package.

It is an error to declare a field variable in an interface with the `transient` or `volatile` modifier.

References Interface Modifiers; Variable modifiers

Interface variable type

If the interface variable declaration uses a primitive type, the variable contains a constant value of the specified primitive type. If the declaration uses a reference type, the variable contains a constant reference to the specified type of object. The presence of square brackets in a variable declaration, after either the type or variable name, indicates that the variable contains a reference to an array.

References Array Types; Primitive Types; Reference Types

Interface variable name

The identifier that follows the variable type is the name of the variable. This identifier can be used anywhere that the variable is accessible.

It is an error to declare two field variables with the same name in the same interface. It is also an error to declare a field variable with the same name as a method declared in the same interface or any of its super-interfaces.

An interface that extends another interface inherits all of the variables in its super-interface. Any class that implements an interface has access to all of the variables defined in that interface, as well as the variables inherited from super-interfaces.

If a field variable is declared with the same name as a variable declared in a super-interface, the variable in the super-interface is considered to be shadowed. If a variable is shadowed in an interface, it cannot be accessed as a field of that interface. However, a shadowed variable can be accessed by casting a reference to an object that implements the interface to a reference to the appropriate super-interface in which the variable is not shadowed. For example:

```
interface A {
    int x = 4;
}
interface B extends A {
    int x = 7;
}
class Z implements B {
    Z() {
        int i = x;              // i gets the value of B's x
        int h = ((A)this).x;    // h gets the value of A's x
    }
}
```

The variable x in interface A is shadowed by the variable x in interface B. Class Z implements interface B, so a reference to x produces the value 7, as defined in interface B. However, it is possible to access the shadowed variable by casting this to a reference to interface A.

In some situations, an interface may inherit multiple field variables with the same name. This leads to a single, ambiguous variable name. For example:

```
interface A {
    int x = 4;
}
interface B {
    int x = 43;
}
interface C extends A, B {
    int y = 22;
}
class Z implements C {
    public static void main (String[] argv) {
        System.out.println(x);        // Ambiguous
    }
}
```

In this example, the interface `C` inherits two variables named `x`. This is fine, as long as `C` does not refer to the variable `x` by its simple name in any of its declarations. If `C` needs to use `x`, it must qualify the name with the appropriate interface name (e.g., `A.x`). Class `Z` implements interface `C`, so it also has access to two variables named `x`. As a result, the use of `x` in `main()` is ambiguous. This problem can be resolved by qualifying the variable name with the appropriate interface name (e.g., `B.x`).

A class that implements multiple interfaces can also inherit multiple field variables with the same name. Again, this leads to a single, ambiguous variable name:

```
interface A {
    int x = 4;
}
interface B {
    int x = 43;
}
class Z implements A, B {
    public static void main (String[] argv) {
        System.out.println(x);        // Ambiguous
    }
}
```

The class `Z` implements both interface `A` and interface `B`, so it inherits two variables named `x`. As a result, the use of `x` in `main()` is ambiguous. This problem can again be resolved by qualifying the variable name with the appropriate interface name (e.g., `B.x`).

References Field Expressions; Identifiers; Interface method name

Interface variable initializers

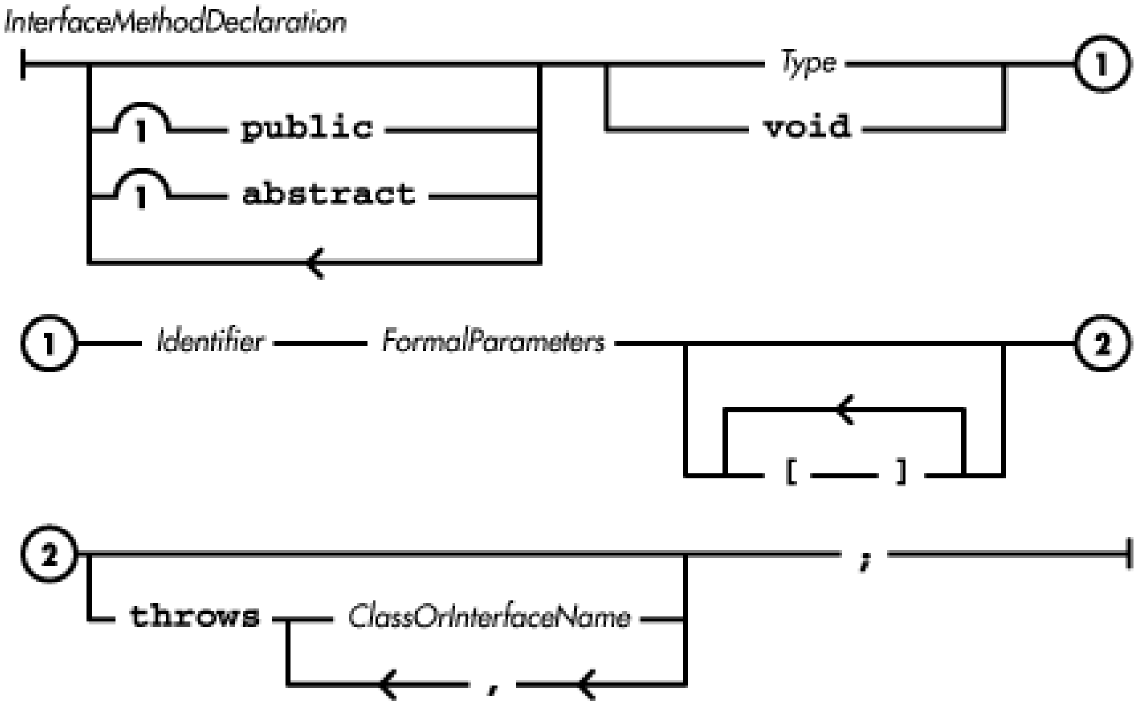
Every variable declaration in an interface must include an initializer that sets the value of the constant. The initializer does not, however, have to be a constant expression. If the variable is of a non-array type, the expression in the initializer is evaluated and the variable is set to the result of the expression, as long as the result is assignment-compatible with the variable. If the variable is of an array type, the initializer must be an array initializer.

The initializer for a variable in an interface cannot refer to any variables that are declared after its own declaration.

References Variable initializers; Array Types; Assignment Operators; Constant Expressions; *Expression* 4

Interface Methods

Any methods declared in an interface are implicitly `abstract`. In other words, methods in an interface do not have a specified implementation:



A method declaration in an interface is made up of six distinct things:

- Optional modifiers that specify attributes of the method
- A type that specifies the type of value returned by the method
- An identifier that names the method
- A list of formal parameters that specifies the values that are passed to the method
- An optional `throws` clause that specifies any exceptions that can be thrown by the method
- A semicolon, since the method declaration does not include an implementation

References *ClassOrInterfaceName* 4.1.6; Exception Handling 9; Method formal parameters; Identifiers; *Type* 3

Interface method modifiers

Methods in an interface are implicitly `abstract`. Including this modifier in a method declaration is permitted, but it is not necessary and it does not change the meaning of the method declaration. Thus, by definition, none of the methods in an interface has a specified implementation.

If an interface is declared `public`, a method declared in the interface is `public`, even if it is declared with the `private` or `protected` modifier. If the interface is not declared `public`, however, any methods in the interface have the default accessibility, which means that they are only accessible in classes and interfaces in the same package.

It is an error to declare a method in an interface with the `static`, `final`, `native`, or `synchronized` modifier. These modifiers are not allowed because defining a method in an interface is not meant to imply anything about the nature of the implementation, other than the return type of the method and the types of the formal parameters. A class that implements the interface has control over the implementation of the methods and can use any of these modifiers when they are appropriate for the implementation.

References Interface Modifiers; Method modifiers

Interface method return type

A method declaration in an interface must specify the type of value returned by the method. The return value can be of a primitive type or of a reference type. The presence of square brackets in a method declaration, after either the return type or the formal parameters, indicates that the method returns a reference to the specified type of array. If the method does not return a value, the declaration uses `void` to indicate that. The return type comes before the name of the method in the method declaration.

References Array Types; Method return type; Primitive Types; Reference Types

Interface method name

The identifier that follows the return type is the name of the method. This identifier can be used anywhere that the method is accessible.

It is an error to declare two methods that have the same name, the same number of parameters, and the same type for each corresponding parameter in the same interface. It is also an error to declare a method with the same name as a variable declared in the same interface or any of its super-interfaces.

An interface that extends another interface inherits all of the methods in its super-interface. Any class that implements an interface must provide an implementation for each of the methods defined in that interface, as well as each of the methods inherited from super-interfaces.

If an interface inherits methods from multiple super-interfaces that have the same name, formal parameters, and return type, there is no problem. The various super-interfaces are in agreement about the method. The interface can also override the inherited methods by declaring a method with the same name, formal parameters, and return type. In any case, a class that implements the interface has to provide a single implementation for the method.

However, if an interface inherits methods from multiple super-interfaces that have the same name and same formal parameters, but different return types, a compile-time error results. By the same token, if the interface attempts to override an inherited method with a method that has the same name and same formal parameters, but a different return type, a compile-time error results.

If an interface inherits methods from multiple super-interfaces that have the same name but different formal parameters, there is no problem. The methods are simply considered overloaded in the interface. The interface can even declare additional methods that have the same name but different formal parameters. A class that implements the interface simply has to provide an implementation for each of the overloaded methods.

References Identifiers; Interface variable name; Method Call Expression

Interface method formal parameters

The formal parameters in a method declaration specify a list of variables to which values are assigned when the method is called. If a method has no formal parameters, the parentheses must still appear in the method declaration. The presence of square brackets in a formal parameter declaration, either as part of a reference type or after the name of a formal parameter, indicates that the formal parameter is an array type.

References Array Types; Method formal parameters; Method formal parameters; *Type* 3

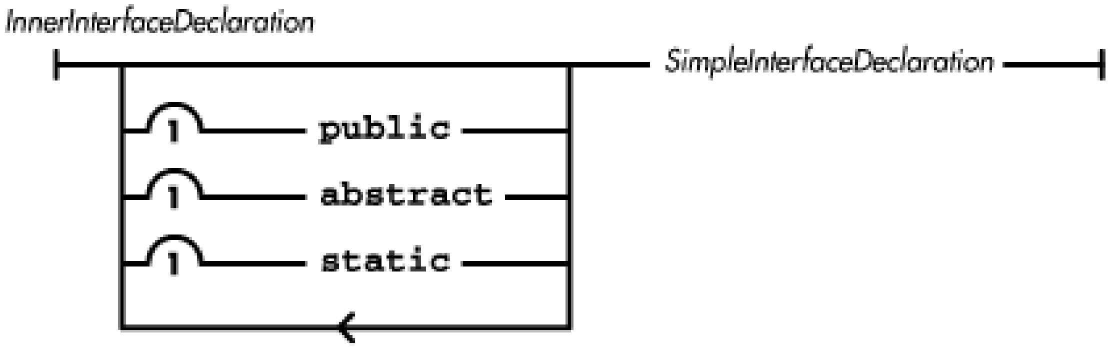
Interface method throws clause

If a method is expected to throw any exceptions, the method declaration must declare that fact in a `throws` clause. If the declaration of a method in an interface contains a `throws` clause, any method in a sub-interface that overrides that method cannot include any classes in its `throws` clause that are not declared in the overridden method.

References Exception Handling 9; Method throws clause

Nested Top-Level Interfaces

Nested top-level interfaces are interfaces that are declared inside of another class. Just as with a top-level interface declaration, the declaration of a nested top-level interface creates a reference type in Java. Here's the formal definition of a nested top-level interface:



A nested top-level interface can be accessed outside of its enclosing class by qualifying its name with the name of its enclosing class, as follows:

`EnclosingClass.InnerInterface`

The syntax for declaring nested top-level interfaces is not supported prior to Java 1.1.

References Nested top-level classes and interfaces; *SimpleInterfaceDeclaration* 5.5

Inner interface modifiers

The keywords `public`, `abstract`, and `static` can be used in the declaration of a nested top-level interface. In this situation, these modifiers have the following meanings:

`public`

> If a nested top-level interface is declared `public`, it is accessible from any class or interface that can access the enclosing class. If the `public` modifier is not used, however, the nested top-level interface can only be referenced by classes and interfaces in the same package as the enclosing class.

`abstract`

> A nested top-level interface is implicitly `abstract`; thus, all of the methods in the interface are implicitly `abstract`. Including the `abstract` modifier in a nested top-level interface declaration is permitted, but it does not change the meaning of the interface declaration.

`static`

> A nested top-level interface is implicitly `static`. Including the `static` modifier in a nested top-level interface declaration is permitted, but it does not change the meaning of the interface declaration.

References Interface Modifiers

Inner interface members

The remainder of a nested top-level interface declaration is the same as that for a top-level interface declaration, which is described in Interface Declarations.

References Interface Declarations; Interface Methods; Interface Variables

PREVIOUS
Class Declarations

HOME
BOOK INDEX

NEXT
Statements and Control Structures

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

5.4 Class Declarations

A class declaration creates a reference type in Java. The class declaration also specifies the implementation of the class, including its variables, constructors, and methods. The formal definition of a class declaration is:

[Graphic: Figure from the text]

[Graphic: Figure from the text]

[Graphic: Figure from the text]

While the above diagram may seem complicated, a class declaration is really made up of six distinct things:

- Optional modifiers that specify attributes of the class
- The keyword `class`
- An identifier that names the class
- An optional `extends` clause that specifies the superclass of the declared class
- An optional `implements` clause that specifies the interfaces implemented by the declared class
- Any number of member declarations, which can include variables, methods, constructors, static initializers, instance initializers, nested top-level classes and interfaces, and member classes

References Classes; *ClassOrInterfaceName* 4.1.6; Class Members; Identifiers

Class Modifiers

The keywords public, abstract, and final can appear as modifiers at the beginning of a class declaration. In this situation, these modifiers have the following meanings:[3]

> [3] Version 1.0 of Java included a private protected access specification; this specification has been removed as of version 1.0.2 of the language.

public

> If a class is declared public, it can be referenced by any other class. If the public modifier is not used, however, the class can only be referenced by other classes in the same package. A single source file, or compilation unit, can only declare one public class or interface (see Compilation Units for an exception to this rule).

abstract

> If a class is declared abstract, no instances of the class may be created. A class declared abstract may contain abstract methods. Classes not declared abstract may not contain abstract methods and must override any abstract methods they inherit with methods that are not abstract. Furthermore, classes that implement an interface and are not declared abstract must contain or inherit methods that are not abstract that have the same name, number of parameters, and corresponding parameter types as the methods declared in the interfaces that the class implements.

final

> If a class is declared final, it cannot be subclassed. In other words, it cannot appear in the extends clause of another class.
>
> You want to declare a class final if it is important to ensure the exact properties and behavior of that class. Many of the classes in the java.lang package are declared final for that reason. In addition, the compiler can often optimize operations on final classes. For example, the compiler can optimize operations involving the String class because it can safely assume the exact logic of String methods. The compiler does not have to account for the possibility of methods of a final class being overridden in a subclass.

References Compilation Units; Inner class modifiers; Local class modifiers; Method modifiers; Variable modifiers

Class Name

The identifier that follows the keyword class is the name of the class. This identifier can be used as a reference type wherever the class is accessible.

References Class Types

Class Inheritance

The `extends` clause specifies the superclass of the class being declared. If a class is declared without an `extends` clause, the class `Object` is its implicit superclass. The class inherits all of the accessible methods and variables of its superclass.

If a class is declared `final`, it cannot appear in an `extends` clause for any other class.

The `implements` clause specifies any interfaces implemented by the class being declared. Unless it is an `abstract` class, the class (or one of its superclasses) must define implementations for all of the methods declared in the interfaces.

References Inheritance; Interfaces; Interface Declarations; Object

Class Members

Fields are the variables, methods, constructors, static (load-time) initializers, instance initializers, nested top-level classes and interfaces, and member classes that are declared as part of a class:

[Graphic: Figure from the text]

A member declaration causes the member to be defined throughout the entire class and all of its subclasses. This means that it is not a problem to have forward references to members, or in other words, you can use members in a class before you have defined them. For example:

```
class foo {
    void doIt() {
        countIt();
    }
    void countIt() {
        i++;
    }
    int i;
}
```

References Constructors; Nested Top-Level and Member Classes; Nested Top-Level Interfaces; Instance Initializers; Methods; Static Initializers; Variables

Variables

A variable that is declared as a member in a class is called a *field variable*. A field variable is different from a local variable, which is declared within a method or a block. The formal definition of a variable declaration is:

[Graphic: Figure from the text]

While the above diagram may seem complicated, a variable declaration is really made up of four distinct things:

- Optional modifiers that specify attributes of the variable.

- A type, which can be either a primitive type or a reference type.
- Any number of identifiers that name variables. Each name can be followed by pairs of square brackets to indicate an array variable.
- An optional initializer for each variable declared.

Here are some examples of variable declarations:

```
int x;
public static final double[] k, m[];
```

References Variable initializers; *Expression* 4; Identifiers; Local Variables; *Type* 3

Variable modifiers

The modifiers `public`, `protected`, and `private` can be used in the declaration of a field variable to specify the accessibility of the variable. In this situation, the modifiers have the following meanings:[4]

> [4] Version 1.0 of Java included a `private protected` access specification; this specification has been removed as of version 1.0.2 of the language.

`public`

> A field variable that is declared `public` is accessible from any class.

`protected`

> A field variable that is declared `protected` is accessible to any class that is part of the same package as the class in which the variable is declared. Such a field variable is also accessible to any subclass of the class in which it is declared; this occurs regardless of whether or not the subclass is part of the same package.

`private`

> A field variable that is declared `private` is only accessible in the class in which it is declared. Such a field variable is not accessible to other classes. In particular, a field variable that is declared `private` is not accessible in subclasses of the class in which it is declared.

If a field variable is not declared with any of the access modifiers, the variable has the default accessibility. Default access is often called "friendly" access because it is similar to `friendly` access in

C++. A variable with default access is accessible in any class that is part of the same package as the class in which the variable is declared. However, a friendly variable is not accessible to classes outside of the package in which it is declared, even if the desired classes are subclasses of the class in which it is declared.

The keywords `static`, `final`, `transient`, and `volatile` can also be used in the declaration of a field variable. These modifiers have the following meanings:

`static`

> A field variable that is declared with the `static` modifier is called a *class variable*. There is exactly one copy of each class variable associated with the class; every instance of the class shares the single copy of the class's `static` variables. Thus, setting the value of a class variable changes the value of the variable for all objects that are instances of that class or any of its subclasses.
>
> For example, if you want to count how many instances of a class have been instantiated, you can write:

```
class Foo {
    ...
    static int fooCount = 0;
    Foo() {
        fooCount++;
    }
    ...
}
```

> A field variable that is not declared with the `static` modifier is called an *instance variable*. There is a distinct copy of each instance variable associated with every instance of the class. Thus, setting the value of an instance variable in one object does not affect the value of that instance variable in any other object.

`final`

> If a field variable is declared with the `final` modifier, the variable is a named constant value. As such, it must be assigned an initial value. Any assignment to a `final` variable, other than the one that provides its initial value, is a compile-time error. The initial value for a `final` variable is typically provided by an initializer that is part of the variable's declaration. For example:

```
final int X = 4;
```

A `final` field variable that is not initialized in its declaration is called a *blank final*. Blank finals are not supported prior to Java 1.1. A blank final that is declared `static` must be assigned a value exactly once in a static initializer. A blank final that is not declared `static` must be assigned a value exactly once in an instance initializer or exactly once in each constructor. The compiler uses flow analysis that takes `if` statements and iteration statements into account to ensure that a blank final is assigned a value exactly once. Thus, it is possible to have multiple assignments to a blank final, so long as exactly one of them can be executed. For example, here is an instance initializer that sets the value of a blank final:

```
{
    final int DAYS_IN_YEAR;
    if (isLeapYear(new Date()))
        DAYS_IN_YEAR = 366;
    else
        DAYS_IN_YEAR = 365;
    ...
}
```

Note that the meaning of `final` in a variable declaration is very different from the meaning of `final` in a method or class declaration. In particular, if a class contains a `final` variable, you can declare a variable with the same name in a subclass of that class without causing an error.

`transient`

The `transient` modifier is used to indicate that a field variable is not part of the persistent state of an object. The `java.io.ObjectOutputStream` class defines `write()` methods that output a representation of an object that can be read later to create a copy of the object. These `write()` methods do not include field variables that are declared `transient` in the representation of an object.

`volatile`

The `volatile` modifier is used to tell the compiler that a field variable will be modified asynchronously by methods that are running in different threads. Each time the variable is accessed or set, it is fetched from or stored into global memory in a way that avoids the assumption that a version of the variable in a cache or a register is consistent with the version in global memory.

References Class Modifiers; Inner class modifiers; Local class modifiers; Method modifiers

Variable type

A field variable declaration must always specify the type of the variable. If the declaration of a field variable uses a primitive type, the variable contains a value of the specified primitive type. If the declaration uses a reference type, the variable contains a reference to the specified type of object.

The presence of square brackets in a variable declaration, after either the type or variable name, indicates that the variable contains a reference to an array. For example:

```
int a[];        // a is an array of int
int[] b;        // b is also an array of int
```

It is also possible to declare a variable to contain an array of arrays, or more generally, arrays nested to any level. Each pair of square brackets in the declaration corresponds to a dimension of the array; it makes no difference whether the brackets appear after the type or the variable name. For example:

```
int[][][] d3;      // Each of these is an array of
int[][] f3[];      // arrays of arrays of integers
int[] g3[][];
int h3[][][];
int[] j3, k3[];    // An array and an array of arrays
```

References Array Types; Primitive Types; Reference Types

Variable name

The identifier that follows the variable type is the name of the variable. This identifier can be used anywhere that the variable is accessible.

It is an error to declare two field variables with the same name in the same class. It is also an error to declare a field variable with the same name as a method declared in the same class or any of its superclasses.

If a field variable is declared with the same name as a variable declared in a superclass, the variable in the superclass is considered to be *shadowed*. If a variable is shadowed in a class, it cannot be accessed as a field of that class. However, a shadowed variable can be accessed by casting a reference to an object of that class to a reference to the appropriate superclass in which the variable is not shadowed. For example:

```
class A {
    int x = 4;
}
class B extends A {
    int x = 7;
    B () {
        int i = x;             // i gets the value of B's x
        int h = ((A)this).x;   // h gets the value of A's x
    }
}
```

Alternatively, if a variable is shadowed in a class but not in its immediate superclass, the methods of the class can access the shadowed variable using the keyword `super`. In the above example, this would look as follows:

```
int h = super.x;        // h gets the value of A's x
```

If a method is declared with the same name and parameters as a method in a superclass, the method in the superclass is considered to be overridden. Note that variable shadowing is different than method overriding. The most important difference is that using a reference to an instance of an object's superclass does not provide access to overridden methods. Overriding is described in detail in Method name.

References Field Expressions; Identifiers; Inheritance; Method name

Variable initializers

A variable declaration can contain an initializer. However, if a variable is declared to be `final`, it must either have an initializer or be initialized exactly once in a static initializer, instance initializer, or constructor. If the variable is of a non-array type, the expression in the initializer is evaluated and the variable is set to the result of the expression, as long as the result is assignment-compatible with the variable. If the variable is of an array type, the initializer must be an array initializer:

[Graphic: Figure from the text]

Each expression or array initializer in an array initializer is evaluated and becomes an element of the array produced by the initializer. The variable is set to the array produced by the initializer, as long as the assignment is assignment-compatible. Here are some examples of actual array initializers:

```
short a[] = {2,5,8,2,11};    // array of 5 shorts
int s[][] = { {3,45,8},      // array of 4 arrays
              {12,9,33},     // of 3 ints
```

```
               {7,22,53},
               {33,1,2} };
```

Note that a trailing comma is allowed within an array initializer. For example, the following is legal:

```
int x[] = {2,23,4,};
```

Any initializers for class variables (i.e., `static` variables) are evaluated when the class is loaded. The initializer for a class variable cannot refer to any instance variables in the class. An initializer for a `static` variable cannot refer to any `static` variables that are declared after its own declaration. The initial value of a class variable can also be set in a static initializer for the class; static initializers are described in Static Initializers.

Any initializers for instance variables are evaluated when a constructor for the class is called to create an instance of the class. Every class has at least one constructor that explicitly or implicitly calls one of the constructors of its immediate superclass before it does anything else. When the superclass's constructor returns, any instance variable initializers (and instance initializers) are evaluated before the constructor does anything else. The initial value of an instance variable can also be set in an instance initializer; instance initializers are described in Instance Initializers. Of course, it is also possible to set the initial values of instance variables explicitly in a constructor. Constructors are described in Constructors.

If a variable declaration does not contain an initializer, the variable is set to a default value. The actual value is determined by the variable's type. Table 5-1 shows the default values used for the various types in Java.

Table 5.1: Default Values for Field Variables

Type	**Default Value**
`byte`	`0`
`char`	`'\u0000'`
`short`	`0`
`int`	`0`
`long`	`0L`
`float`	`0.0F`
`double`	`0.0`
`boolean`	`false`
Object reference	`null`

For an array, every element of the array is set to the appropriate default value, based on the type of elements in the array.

Here are some examples of variable declarations, with and without initializers:

```
int i,j;                 // initialized to zero
long k = 243L;
double d = k*1.414;
String s;                // initialized to null
char c[] = new char[123];
float f[] = { 3.2f, 4.7f, 9.12f, 345.9f};
Double dbl = new Double(382.3748);
java.io.File fl = new File("/dev/null");
Object o = fl;
```

References Array Types; Assignment Operators; Constructors; *Expression* 4; Instance Initializers; Static Initializers; Variable modifiers

Methods

A method is a piece of executable code that can be called as a subroutine or a function. A method can be passed parameters by its caller; the method can also return a result to its caller. In Java, a method can only be declared as a field in a class. The formal definition of a method declaration is:

[Graphic: Figure from the text]

While the above diagram may seem complicated, a method declaration is really made up of six distinct things:

- Optional modifiers that specify attributes of the method

- A type that specifies the type of value returned by the method
- An identifier that names the method
- A list of formal parameters that specifies the values that are passed to the method
- An optional `throws` clause that specifies any exceptions that can be thrown by the method
- A block that defines the functionality of the method

Here are some examples of method declarations:

```
public static void main(String[] argv) {
    System.out.println( argv[0] );
}
int readSquare(DataInputStream d) throws IOException {
    int i = d.readInt();
    return i*i;
}
int filledArray(int length, int value) [] {
    int [] array = new int [length];
    for (int i = 0; i < length; i++ ) {
        array[i] = value;
    }
    return array;
}
```

Unlike C/C++, Java only allows method declarations that fully specify the type and number of parameters that the method can be called with.

References Blocks; *ClassOrInterfaceName* 4.1.6; Exception Handling 9; Method formal parameters; Identifiers; *Type* 3

Method modifiers

The modifiers `public`, `protected`, and `private` can be used in the declaration of a method to specify the accessibility of the method. In this situation, the modifiers have the following meanings:

`public`

> A method that is declared `public` is accessible from any class.

`protected`

> A method that is declared `protected` is accessible in any class that is part of the same package as the class in which the method is declared. Such a method is also accessible to any subclass of the class in which it is declared, regardless of whether or not the subclass is part of the same package.

`private`

> A method that is declared `private` is only accessible in the class in which it is declared. Such a method is not accessible in other classes. In particular, a method that is declared `private` is not accessible in subclasses of the class in which it is declared. A method cannot be declared both `private` and `abstract`.

If a method is not declared with any of the access modifiers, it has the default accessibility. Default access is often called "friendly" access because it is similar to `friendly` access in C++. A method with default access is accessible in any class that is part of the same package as the class in which the method is declared. However, a friendly method is not accessible to classes outside of the package in which it is declared, even if the classes are subclasses of the class in which it is declared.

The keywords `static`, `final`, `abstract`, `native`, and `synchronized` can also be used in the declaration of a method. These modifiers have the following meanings:

`static`

> A method that is declared with the `static` modifier is called a *class method*. Class methods are not associated with an instance of a class. This means that a class method cannot directly refer to other, non-`static` methods or variables in its class, unless the method or variable is accessed through an explicit object reference. In addition, the keywords `this` and `super` are treated as undefined variables within `static` methods. A method that is declared `static` is also implicitly `final`, or in other words, `static` methods cannot be overridden. A method that is declared `static` cannot also be declared `abstract`.
>
> Because `static` methods are not associated with a class instance, you do not need an instance of a class to invoke such a method. For example, the `Math` class contains a collection of mathematical methods that can be called using the class name:
>
> ```
> Math.tan(x)
> ```
>
> A method that is not declared with the `static` modifier is called an *instance method*. Instance methods are associated with an instance of a class, so an instance method may contain direct

references to any other methods or variables in its class.

`final`

A method that is declared with the `final` modifier cannot be overridden. In other words, if a method in a class is declared `final`, no subclass of that class can declare a method with the same name, number of parameters, and parameter types as the `final` method. Although `final` methods cannot be overridden, declaring a method to be `final` in no way prevents it from being overloaded.

`abstract`

If a method is declared with the `abstract` modifier, the declaration must end with a semicolon rather than a block. An `abstract` method declaration specifies the name, number and type of parameters, and return type of the method; it does not specify the implementation of the method. If a class contains an `abstract` method, the class must also be declared `abstract`. If a non-`abstract` class inherits an `abstract` method, the class must override the method and provide an implementation.

An `abstract` method cannot also be declared either `private` or `static` because neither `private` nor `static` methods can be overridden. A `private` method cannot be overridden because it is not inherited by its subclasses; a `static` method cannot be overridden because it is implicitly `final`.

`native`

If a method is declared with the `native` modifier, the declaration must end with a semicolon rather than a block. A `native` method is implemented in a platform-specific way using a language other than Java, such as C++. Because the implementation of a `native` method is not done in Java, Java requires the semicolon in place of an implementation.

Because the implementation of a `native` method is platform-specific, you should avoid using native methods in classes that are expected to run on different kinds of clients. Native methods also require an installation process, which is another reason to avoid them for use on clients.

`synchronized`

If a method is declared with the `synchronized` modifier, a thread must obtain a lock before it can invoke the method. If the method is not declared `static`, the thread must obtain a lock associated with the object used to access the method. If the method is declared `static`, the thread must obtain a lock associated with the class in which the method is declared.

> A synchronized method is one of two mechanisms for providing single-threaded access to the contents of a class or object. The other mechanism is the `synchronized` statement. Of the two, a synchronized method is usually the preferred mechanism. If all access to instance data that is shared by multiple threads is through `synchronized` methods, the integrity of the instance data is guaranteed, no matter what the callers of the methods do. On the other hand, if instance data shared by multiple threads is directly accessible outside of the class that defines it or its subclasses, providing single-threaded access to the data requires the use of `synchronized` statements.

References Class Modifiers; Inner class modifiers; Local class modifiers; Variable modifiers

Method return type

A method declaration must always specify the type of value returned by the method. The return value can be of a primitive type or of a reference type. If the method does not return a value, it should be declared with its return type specified as `void`. The return type comes before the name of the method in the method declaration.

The presence of square brackets in a method declaration, after either the return type or the formal parameters, indicates that the method returns a reference to the specified type of array.

For example:

```
int a()[] {...};        // a returns an array of int
int[] b() {...};        // b also returns an array of int
```

It is also possible to declare that a method returns a reference to an array of arrays, or more generally, arrays nested to any level. Each pair of square brackets in the declaration corresponds to a dimension of the array; it makes no difference whether the brackets appear after the return type or the formal parameters. For example:

```
int[][][] d3() {...};    // Each of these returns an array of
int[][] f3()[] {...};    // arrays of arrays of integers
int[] g3()[][] {...};
int h3()[][][] {...};
```

If a method is declared with the `void` return type, any `return` statement that appears within the method must not contain a return value. Because a method with a `void` return type does not return a value, such a method can only be called from an expression statement that consists of a method call expression.

On the other hand, if a method is declared with a return type other than `void`, it must return through an explicit `return` statement that contains a return value that is assignment-compatible with the return type of the method.

References Array Types; Expression Statements; Primitive Types; Reference Types; The return Statement

Method name

The identifier that follows the return type is the name of the method. This identifier can be used anywhere that the method is accessible.

It is an error to declare two methods that have the same name, the same number of parameters, and the same type for each corresponding parameter in the same class. It is also an error to declare a method with the same name as a variable declared in the same class or any of its superclasses.

A method is said to be *overloaded* if there is more than one accessible method in a class with the same name, but with parameters that differ in number or type.[5] This situation can arise if two or more such methods are declared in the same class. It can also occur when at least one of the methods is defined in a superclass and the rest are in a subclass.

> [5] Although Java supports overloaded methods, it does not allow programs to define overloaded operators. While it is true that the `+` operator is defined in an overloaded way, that operator is part of the language specification and it is the only overloaded operator.

Overloaded methods aren't required to have the same return type. For example:

```
int max(int x, int y){return x>y ? x : y;}
double max(double x, double y){return x>y ? x : y;}
```

A method that is inherited from a superclass is said to be *overridden* if a method in the inheriting class has the same name, number of parameters, and types of parameters as the inherited method. If the overridden method returns `void`, the overriding method must also return `void`. Otherwise, the return type of the overriding method must be the same as the type of the overridden method.

An overriding method can be more accessible than the overridden method, but it cannot be less accessible. In other words, a subclass cannot hide things that are visible in its superclass, but it can make visible things that are hidden. An object is considered to be an instance of its own class, as well as an instance of each of its superclasses. As a result, you can use an object reference to call a method in an object and not worry about whether the object is actually an instance of a subclass of the type of the reference. If a subclass were allowed to override methods of its superclass with methods that were less accessible, you would no longer be able to use a reference without regard to the actual type of the object being referenced.

For example, `Object` is the superclass of `String`. This means that a variable declared to contain a reference to an `Object` may actually refer to a `String`. The `Object` class defines a `public` method called `hashCode()`, so a reference to the `Object` class can be used to call the `hashCode()` method of whatever subclass of `Object` it refers to. Allowing a subclass of `Object` to declare a `private hashcode()` method would be inconsistent with this usage.

Table 5-2 shows the access modifiers that are permitted for an overriding method, based on the access allowed for the overridden method.

Table 5.2: Permitted Access Modifiers for Overriding Methods

		Access declared for overridden method		
		no modifier	**`protected`**	**`public`**
Access for overriding method	`private`	not allowed	not allowed	not allowed
	no modifier	allowed	not allowed	not allowed
	`protected`	allowed	allowed	not allowed
	`public`	allowed	allowed	allowed

If a method in the superclass is declared `private`, it is not inherited by the subclass. This means that a method in the subclass that has the same name, number of parameters, and types of parameters does not override the `private` method in the superclass. As a result, the method in the subclass can have any return type and there are no restrictions on its accessibility.

Non-`static` methods must be called through an object reference. If a non-`static` method is called with no explicit object reference, it is implicitly called using the object reference `this`. At compile-time, the type of the object reference is used to determine the combinations of method names and parameters that are accessible to the calling expression (see Method Call Expression). At runtime, however, the actual type of the object determines which of the methods is called. If the actual object is an instance of a subclass of the referenced class and the subclass overrides the method being called, the overriding method in the subclass is invoked.

In other words, the actual type of the object is used to determine which method to call, not the type of the reference to that object. This means that you cannot simply cast an object reference to a superclass of the class of the actual object to call to an overridden method. Instead, you use the keyword `super` to access an overridden method in the superclass. For example:

```
class A {
    void doit() {
        ...
    }
}
class B extends A {
    void doit() {
        super.doit();      // calls overridden A.doit()
    }
    public static void main(String argv[]) {
        B b = new B();
        ((A)b).doit();     // calls B.doit()
    }
}
```

The `doit()` method in class `B` calls the overridden `doit()` method in class `A` using the `super` construct. But, in `main()`, the `doit()` method in class `B` is invoked because casting a reference does not provide access to overridden methods.

References Identifiers; Inheritance; Method Call Expression; Variable name

Method formal parameters

The formal parameters in a method declaration specify a list of variables to which values are assigned when the method is called:

[Graphic: Figure from the text]

Within the block that contains the implementation of the method, the method's formal parameters are treated as local variables; the name of each formal parameter is available as an identifier in the method's implementation. Formal parameters differ from local variables only in that their declaration and value come from outside the method's block.

If a formal parameter is declared `final`, any assignment to that parameter generates an error. The syntax for declaring `final` parameters is not supported prior to Java 1.1.

If a method has no formal parameters, the parentheses must still appear in the method declaration.

Here's an example of a method declaration with formal parameters:

```
abstract int foo(DataInputStream d, Double[] values, int weights[]) ;
```

The presence of square brackets in a formal parameter declaration, either as part of a reference type or after the name of a formal parameter, indicates that the formal parameter is an array type. For example:

```
foo(int a[],    // a is an array of int
    int[] b)    // b is also an array of int
```

It is also possible to declare that a formal parameter is an array of arrays, or more generally, arrays nested to any level. Each pair of square brackets in the declaration corresponds to a dimension of the array; it makes no difference whether the brackets appear with the type or after the name of the formal parameter. For example:

```
int[][][] d3        // Each of these is an array of
int[][] f3[]        // arrays of arrays of integers
int[] g3[][]
int h3[][][]
```

References Array Types; Blocks; Identifiers; Local Variables; *Type* 3

Method throws clause

If a method is expected to throw any exceptions, the method declaration must declare that fact in a `throws` clause. Java requires that most types of exceptions either be caught or declared, so bugs caused by programmers forgetting to handle particular types of exceptions are uncommon in Java programs.

If a method implementation contains a `throw` statement, or if the method calls another method declared with a `throws` clause, there is the possibility that an exception will be thrown from within the method. If the exception is not caught, it will be thrown out of the method to its caller. Any exception that can be thrown out of a method in this way must be listed in a `throws` clause in the method declaration, unless the exception is an instance of `Error`, `RuntimeException`, or a subclass of one of those classes. Subclasses of the `Error` class correspond to situations that are not easily predicted, such as the system running out of memory. Subclasses of `RuntimeException` correspond to many common runtime problems, such as illegal casts and array index problems. The classes listed in a `throws` clause must be `Throwable` or any of its subclasses; the `Throwable` class is the superclass of all objects that can be thrown in Java.

Consider the following example:

```
import java.io.IOException;
class throwsExample {
    char[] a;
```

```
int position;
...
// Method explicitly throws an exception
int read() throws IOException {
    if (position >= a.length)
        throw new IOException();
    return a[position++];
}
// Method implicitly throws an exception
String readUpTo(char terminator) throws IOException {
    StringBuffer s = new StringBuffer();
    while (true) {
        int c = read(); // Can throw IOException
        if (c == -1 || c == terminator) {
            return s.toString();
        }
        s.append((char)c);
    }
    return s.toString():
}
// Method catches an exception internally
int getLength() {
    String s;
    try {
        s = readUpTo(':');
    } catch (IOException e) {
        return 0;
    }
    return s.length();
}
// Method can throw a RuntimeException
int getAvgLength() {
    int count = 0;
    int total = 0;
    int len;
    while (true){
        len = getLength();
        if (len == 0)
            break;
        count++;
        total += len;
    }
    return total/count; // Can throw ArithmeticException
    }
}
```

The method `read()` can throw an `IOException`, so it declares that fact in its `throws` clause. Without that `throws` clause, the compiler would complain that the method must either declare `IOException` in its `throws` clause or catch it. Although the `readUpTo()` method does not explicitly throw any exceptions, it calls the `read()` method that does throw an `IOException`, so it declares that fact in its `throws` clause. Whether explicitly or implicitly thrown, the requirement to catch or declare an exception is the same. The `getLength()` method catches the `IOException` thrown by `readUpTo()`, so it does not have to declare the exception. The final method, `getAvgLength()`, can throw an `ArithmeticException` if `count` is zero. Because `ArithmeticException` is a subclass of `RuntimeException`, the fact that it can be thrown out of `getAvgLength()` does not need to be declared.

If a method overrides another method, the overriding method cannot throw anything that the overridden method does not throw. Specifically, if the declaration of a method contains a `throws` clause, any method that overrides that method cannot include any classes in its `throws` clause that are not declared in the overridden method. This restriction avoids surprises. When a method is called, the Java compiler requires that all of the objects listed the method's `throws` clause are either caught by the calling method or declared in the calling method's `throws` clause. The requirement that an overriding method cannot include any class in its `throws` clause that is not in the overridden method's `throws` clause ensures that the guarantee made by the compiler is respected by the runtime environment.

References Exception Handling 9; The throw Statement; The try Statement

Method implementation

A method declaration must end with either a block or a semicolon. If either the `abstract` or `native` modifier is used in the declaration, the declaration must end with a semicolon. All other method declarations must end with a block that defines the implementation of the method.

References Blocks; Method modifiers

Constructors

A constructor is a special kind of method that is designed to set the initial values of an object's instance variables and do anything else that is necessary to create an object. Constructors are only called as part of the object creation process. The declaration of a constructor does not include a return type. The name of a constructor is always the same as the name of the class:

[Graphic: Figure from the text]

A constructor declaration is really made up of five distinct things:

- Optional modifiers that specify attributes of the constructor
- An identifier that names the constructor; this identifier must be the same as the name of the class
- A list of formal parameters that specifies the values that are passed to the constructor
- An optional `throws` clause that specifies any exceptions that can be thrown by the constructor
- A block that defines the functionality of the constructor

Here is an example that shows a class with some constructors:

```
class Construct {
    private Construct(Double[] values, int weights[]) {
    }
    public Construct(OutputStream o, Double[] values, int weights[])
                    throws IOException {
        this(values, weights);
        o.write(weights[0]);
    }
    public Construct() {
    }
}
```

References Blocks; *ClassOrInterfaceName* 4.1.6; Exception Handling 9; Method formal parameters; Identifiers; Object Creation

Constructor modifiers

The modifiers `public`, `protected`, and `private` can be used in the declaration of a constructor to specify the accessibility of the constructor. In this situation, the modifiers have the following meanings:

`public`

> A constructor that is declared `public` is accessible from any class.

`protected`

A constructor that is declared `protected` is accessible in any class that is part of the same package as the class in which the constructor is declared. Such a constructor is also accessible to any subclass of the class in which it is declared, regardless of whether or not the subclass is part of the same package.

`private`

A constructor that is declared `private` is accessible in the class in which it is declared. Such a constructor is not accessible in other classes. In particular, a constructor that is declared `private` is not accessible in subclasses of the class in which it is declared.

If a class is declared with at least one constructor, to prevent Java from providing a default `public` constructor, and all of the constructors are declared `private`, no other class can create an instance of the class. It makes sense to prevent the instantiation of a class if the class exists only to provide a collection of `static` methods. An example of this type of class is `java.lang.Math`.

`private` constructors can be used by `static` methods in the same class.

If a constructor is not declared with any of the access modifiers, the constructor has the default accessibility. Default access is often called "friendly" access because it is similar to `friendly` access in C++. A constructor with default access is accessible in any class that is part of the same package as the class in which the constructor is declared. However, a friendly constructor is not accessible in subclasses of the class in which it is declared.

References Class Modifiers; Inner class modifiers; Local class modifiers

Constructor name

A constructor has no name of its own. The identifier that appears in a constructor declaration must be the same as the name of the class in which the constructor is declared. This identifier can be used anywhere that the constructor is accessible.

References Class Types

Constructor return type

A constructor has no declared return type; it always returns an object that is an instance of its class. A `return` statement in a constructor is treated the same as it is in a method declared to return `void`; the `return` statement must not contain a return value. Note that it is not possible to explicitly declare a constructor to have the return type `void`.

References The return Statement

Constructor formal parameters

The formal parameters in a constructor declaration specify a list of variables to which values are assigned when the constructor is called. Within the block that contains the implementation of the constructor, the constructor's formal parameters are treated as local variables; the name of each formal parameter is available as an identifier in the constructor's implementation. Formal parameters differ from local variables only in that their declaration and value come from outside the constructor's block.

If a formal parameter is declared `final`, any assignment to that parameter generates an error. The syntax for declaring `final` parameters is not supported prior to Java 1.1.

If a constructor has no formal parameters, the parentheses must still appear in the constructor declaration.

The presence of square brackets in a formal parameter declaration, either as part of a reference type or after the name of a formal parameter, indicates that the formal parameter is an array type. For example:

```
Foo(int a[],    // a is an array of int
    int[] b)    // b is also an array of int
```

It is also possible to declare that a formal parameter is an array of arrays, or more generally, arrays nested to any level. Each pair of square brackets in the declaration corresponds to a dimension of the array; it makes no difference whether the brackets appear with the type or after the name of the formal parameter. For example:

```
int[][][] d3        // Each of these is an array of
int[][] f3[]        // arrays of arrays of integers
int[] g3[][]
int h3[][][]
```

References Array Types; Blocks; Method formal parameters; Local Variables

Constructor throws clause

If a constructor is expected to throw any exceptions, the constructor declaration must declare that fact in a `throws` clause. If a constructor implementation contains a `throw` statement, or if the constructor calls another constructor or method declared with a `throws` clause, there is the possibility that an exception will be thrown from within the constructor.

If the exception is not caught, it will be thrown out of the constructor to its caller. Any exception that can be thrown out of a constructor in this way must be listed in a `throws` clause in the constructor declaration, unless the exception is an instance of `Error`, `RuntimeException`, or a subclass of one of those classes.

Subclasses of the `Error` class correspond to situations that are not easily predicted, such as the system running out of memory. Subclasses of `RuntimeException` correspond to many common runtime problems, such as illegal casts and array index problems. The classes listed in a `throws` clause must be `Throwable` or any of its subclasses; the `Throwable` class is the superclass of all objects that can be thrown in Java.

References Exception Handling 9; The throw Statement; The try Statement

Constructor implementation

The block at the end of a constructor declaration contains the implementation of the constructor. The block is called the constructor body. The first statement in a constructor body is special; it is the only place that Java allows an explicit call to a constructor outside of an allocation expression. An explicit call to a constructor has a special form:

[Graphic: Figure from the text]

In an explicit constructor call, the keyword `this` can be used to specify a call to a constructor in the same class. The keyword `super` can be used to specify a call to a constructor in the immediate superclass.

For example:

```
class Square extends RegularPolygon {
    // Construct a square without specifying the length of the sides
    Square() {
        this(5);
    }
    // Construct a square with sides of a specified length
    Square(int len) {
        super(4,len);
    }
}
```

The first constructor simply calls the second constructor with the argument 5. The second constructor calls a constructor in the immediate superclass to create a four-sided regular polygon with sides of the given length.

Except for the constructors in the class `Object`, a constructor always begins by calling another constructor in the same class or in its immediate superclass. If the first statement in a constructor is not an explicit call to another constructor using `this` or `super` and the class is not `Object`, the compiler inserts a call to `super()` before the first statement in the constructor.

In other words, if a constructor does not begin with an explicit call to another constructor, it begins with an implicit call to the constructor of its immediate superclass that takes no argument. The result is constructor chaining: a constructor for each superclass of a class is called before the constructor of the class executes any of its own code. After all of the calls to the superclasses' constructors (explicit or implicit) have returned, any instance variables that have initializers are initialized, and finally the constructor executes its own code.

Constructor chaining places a restriction on the arguments that can be passed to a constructor in an explicit constructor call. The expressions provided as arguments must not refer to any instance variables of the object being created because these instance variables are not initialized until the superclass's constructor returns.

References *ArgumentList* 4.1.8; Object Allocation Expressions; this; super

The default constructor

If a class declaration does not contain any constructor declarations, Java supplies a default constructor for the class. The default constructor is `public`, it takes no arguments, and it simply calls the constructor of its class's superclass that takes no arguments. The default constructor is approximately equivalent to:

```
public MyClass() {
    super();
}
```

Because Java creates a default constructor only for a class that does not have any explicitly declared constructors, it is possible for the superclass of that class not to have a constructor that takes no arguments. If a class declaration does not contain a constructor declaration and its immediate superclass does not have a constructor that takes no arguments, the compiler issues an error message because the default constructor references a non-existent constructor in the superclass. The default constructor for the class `Object` does not contain a call to another constructor because class `Object` has no superclass.

References Object

Constructor inheritance

A subclass does not inherit constructors from its superclass, as it does normal methods. This is one important difference between regular methods and constructors: constructors are not inherited. However, a subclass can access a constructor in its superclass, as long as the constructor is accessible, based on any access modifiers used in its declaration.

This example illustrates the difference between inheritance and accessibility:

```
public class A {
    public A (int q) {
    }
}
public class B extends A {
    public B () {
        super(5);
    }
}
```

Although class `B` is a subclass of class `A`, `B` does not inherit the public constructor in `A` that takes a single argument. This means that if you try to create a new instance of `B` using an allocation expression with a single argument, you'll get an error message from the compiler. Here's an erroneous call:

```
B b1 = new B(9);
```

However, as shown in the example, the constructor in `B` can access the constructor in `A` using the keyword `super`.

The finalize method

A class declaration can include a special method that is called before an instance of the class is destroyed by the garbage collector. This method is called a *finalizer* ; it has the name `finalize()`. The `finalize()` method for a class must be declared with no parameters, a `void` return type, and no modifiers:

```
void finalize() {...}
```

If a class has a `finalize()` method, it is normally called by the garbage collector before an object of that class type is destroyed. A program can also explicitly call an object's `finalize()` method, but in this case, the garbage collector does not call the method during the object destruction process. If the garbage collector does call an object's `finalize()` method, the garbage collector does not immediately destroy the object because the `finalize()` method might do something that results in a reference to the object. Thus the garbage collector waits to destroy the object until it can again prove it is safe to do so. The next time the garbage collector decides it is safe to destroy the object, it does so without calling the finalizer again. In any case, a `finalize()` method is never called more than once by the garbage collector for a particular object.

A superclass of the class may also define a `finalize()` method, but Java does not provide a mechanism that automatically calls the superclass's `finalize()` method. If a class contains a `finalize()` method, it is a good idea for that method to call `super.finalize()` as the very last thing that it does. This technique ensures that the `finalize()` method of the superclass gets called. The technique even works if the superclass does not explicitly define a `finalize()` method, since every class inherits a default `finalize()` method from the `Object` class. This default `finalize` method does not do anything.

References Object Destruction

Static Initializers

A *static initializer* is a piece of code that is executed when a class is loaded. A static initializer is simply a block of code in a class declaration that is preceded by the keyword `static`:

[Graphic: Figure from the text]

A class is loaded when its definition is needed by another class. You can specifically request that a class be loaded by calling the `forName()` method of the `Class` class on the class you want to load. Alternatively, you can use the `loadClass()` method of a `ClassLoader` object to load a class directly.

When a class is loaded, a `Class` object is created to represent it and storage for the class's `static` variables is allocated. When a class is initialized, its static initializers and `static` variable initializers are evaluated in the order in which they appear in the class declaration. For example, here is a class that contains both static initializers and `static` variable initializers:

```
class foo {
    static int i = 4;
    static {
        i += 2;
        j = 5 * i;
    }
    static int j = 7;
    static double d;
    static frame f = new Frame();
    static { d = Math.tan(Math.PI/j); }
}
```

When the `foo` class is loaded, here is what happens. First, the variable `i` is set to 4. Then the first static initializer is executed. It increments `i` by 2, which makes it 6, and sets `j` to `5*i`, which is 30. Next, the variable `j` is set to 7 by its initializer; this overwrites the value that was set in the static initializer. The variable `f` is then set to the new `Frame` object created by its initializer. Finally, the second static initializer is executed. It sets the variable `d` to , which is .

Notice that the first static initializer uses the variable `j`, even though the variable is not declared until after the static initializer. A static initializer can refer to a `static` variable that is declared after the static initializer. However, the same is not true for `static` variable initializers. A `static` variable initializer cannot refer to any variables that are declared after its own declaration, or the compiler generates an error message. The following class declaration is erroneous:

```
class foo {
    static int x = y*3;    // error because y defined after x
    static int y;
}
```

If an exception is thrown out of a static initializer, the method that caused the class to be defined throws an `ExceptionInInitializerError`. This `ExceptionInInitializerError` contains a reference to the original exception that can be fetched by calling its `getException()` method.

References Blocks; Class; Errors; Variables

Instance Initializers

An *instance initializer* is a piece of code that is executed when an instance of a class is created.

Specifically, it is executed after the object's immediate superclass constructor returns, but before the constructor of the class itself runs. An instance initializer is simply a block of code in a class that is not in any method. Here is the formal syntax:

[Graphic: Figure from the text]

Every class has at least one constructor that explicitly or implicitly calls one of the constructors of its immediate superclass before it does anything else. When the superclass's constructor returns, any instance initializers and instance variable initializers are evaluated before the constructor does anything else. The instance initializers and instance variable initializers are evaluated in the order in which they appear in the class declaration. If an instance initializer throws an exception, the exception appears to have come from the constructor that called the superclass's constructor.

References Blocks; Constructors; Variable initializers

Nested Top-Level and Member Classes

Nested top-level classes and member classes are classes that are declared inside of another class. Just as with a top-level class declaration, the declaration of a nested top-level class or member class creates a reference type in Java. Here's the formal definition of a nested top-level or member class declaration:

[Graphic: Figure from the text]

A class declared inside of another class has access to all of the variables, methods, and other inner classes of the enclosing class. If a nested top-level or member class is not `private`, it can also be accessed outside of its enclosing class by qualifying its name with the name of its enclosing class, as follows:

`EnclosingClass.InnerClass`

The syntax for declaring nested top-level classes and member classes is not supported prior to Java 1.1.

References Nested top-level classes and interfaces; Member classes; Class Declarations

Inner class modifiers

The keywords `public`, `protected`, and `private` can be used in the declaration of a nested top-level or member class to specify the accessibility of the inner class. In this situation, the modifiers have the following meanings:

`public`

> A nested top-level or member class that is declared `public` is accessible from any class that can access the enclosing class.

`protected`

> A nested top-level or member class that is declared `protected` is accessible from any class that is part of the same package as the enclosing class. Such an inner class is also accessible to any

subclass of the enclosing class, regardless of whether or not the subclass is part of the same package.

`private`

A nested top-level or member class that is declared `private` is only accessible from its enclosing class and other classes declared within the enclosing class. In particular, an inner class that is declared `private` is not accessible in subclasses of its enclosing class.

If a nested top-level or member class is not declared with any of the access modifiers, the class has the default accessability. Default access is often called "friendly" access because it is similar to `friendly` access in C++. An inner class with default access is accessible in any class that is part of the same package as the enclosing class. However, a friendly inner class is not accessible to classes outside of the package of the enclosing class, even if the desired classes are subclasses of the enclosing class.

The keywords `abstract`, `final`, and `static` can also be used in the declaration of a nested top-level or member class. These modifiers have the following meanings:

`abstract`

If a nested top-level or member class is declared `abstract`, no instances of the class may be created. An inner class declared `abstract` may contain `abstract` methods; classes not declared `abstract` may not contain `abstract` methods and must override any `abstract` methods they inherit with methods that are not `abstract`. Furthermore, classes that implement an interface and are not declared `abstract` must contain or inherit methods that are not `abstract`, that have the same name, have the same number of parameters, and have corresponding parameter types as the methods declared in the interfaces that the class implements.

`final`

If a nested top-level or member class is declared `final`, it cannot be subclassed. In other words, it connot appear in the `extends` clause of another class.

`static`

An inner class that is declared with the `static` modifier is called a nested top-level class. A class can only be declared with the `static` modifier if its enclosing class is a top-level class (i.e., it is not declared within another class). The code within a nested top-level class cannot directly access non-`static` variables and methods of its enclosing class.

An inner class that is not declared with the `static` modifier is called a member class. The code within a member class can access all of the variables and methods of its enclosing class, including private variables and methods.

References Class Modifiers; Local class modifiers

Inner class members

The body of a nested top-level or member class cannot declare any `static` variables, `static` methods, `static` classes, or static initializers. Beyond those restrictions, the remainder of the declaration is the same as that for a top-level class declaration, which is described in Class Declarations.

References Class Declarations; Constructors; Instance Initializers; Methods; Nested Top-Level and Member Classes; Static Initializers; Variables

4.6 Additive Operators

The additive operators in Java are binary operators that are used for addition and string concatenation (`+`) and subtraction (`-`). The additive operators appear in additive expressions.

[Graphic: Figure from the text]

The additive operators are equal in precedence and are evaluated from left to right.

References Multiplicative Operators; Order of Operations

Arithmetic Addition Operator +

The binary arithmetic addition operator `+` produces a pure value that is the sum of its operands. The arithmetic `+` operator may appear in an additive expression. The arithmetic addition operator never throws an exception.

Here is an example that uses the arithmetic addition operator:

```
int addThree (int x, int y, int z) {
    return x + y + z;
}
```

If the type of either operand of `+` is a reference to a `String` object, the operator is the string concatenation operator, not the arithmetic addition operator. The string concatenation operator is described in String Concatenation Operator +.

Otherwise, the types of both operands of the arithmetic addition operator must be arithmetic types, or a compile-time error occurs. The `+` operator may perform type conversions on its operands:

- If either operand is of type `double`, the other operand is converted to `double` and the operation produces a `double` value.
- Otherwise, if either operand is of type `float`, the other operand is converted to `float` and the operation produces a `float` value.
- Otherwise, if either operand is of type `long`, the other operand is converted to `long` and the operation produces a `long` value.
- Otherwise, both operands are converted to `int` and the operation produces an `int` value.

If the addition of integer data overflows, the value produced by `+` contains the low order bits of the sum and the sign of the value is the opposite of the mathematically correct sign, due to the limitations of the two's complement representation used for integer data.

The addition of floating-point data is governed by the following rules:

- If either operand is not-a-number (NaN), the sum is NaN.
- If one operand is positive infinity and the other is negative infinity, the sum is NaN.
- If both of the operands are positive infinity, the sum is positive infinity.
- If both of the operands are negative infinity, the sum is negative infinity.
- If one of the operands is an infinity value and the other operand is a finite value, the sum is the same infinity value as the operand.
- If one operand is positive zero and the other is negative zero; the sum is positive zero.

- If both operands are positive zero, the sum is positive zero.

- If both operands are negative zero, the sum is negative zero.

- If neither operand is NaN nor an infinity value, the sum is rounded to the nearest representable value. If the magnitude of the sum is too large to be represented, the operation overflows and an infinity value of the appropriate sign is produced. If the magnitude of the sum is too small to be represented, the operation underflows and a zero value of the appropriate sign is produced.

References Arithmetic Types; String Concatenation Operator +

Arithmetic Subtraction Operator -

The binary subtraction operator `-` produces a pure value that is the difference between its operands; it subtracts its right operand from its left operand. The arithmetic `-` operator may appear in an additive expression. The arithmetic subtraction operator never throws an exception.

Here is an example that uses the arithmetic subtraction operator:

```
int subThree (int x, int y, int z) {
    return x - y - z;
}
```

The types of both operands of the arithmetic subtraction operator must be arithmetic types, or a compile-time error occurs. The `-` operator may perform type conversions on its operands:

- If either operand is of type `double`, the other operand is converted to `double` and the operation produces a `double` value.

- Otherwise, if either operand is of type `float`, the other operand is converted to `float` and the operation produces a `float` value.

- Otherwise, if either operand is of type `long`, the other operand is converted to `long` and the operation produces a `long` value.

- Otherwise, both operands are converted to `int` and the operation produces an `int` value.

For both integer and floating-point data, `a-b` always produces the same result as `a-(+b)`.

If the subtraction of integer data overflows, the value produced by `-` contains the low order bits of the difference and the sign of the value is the opposite of the mathematically correct sign, due to the limitations of the two's complement representation used for integer data.

The subtraction of floating-point data is governed by the following rules:

- If either operand is not-a-number (NaN), the difference is NaN.
- If the left operand is positive infinity and the right operand is negative infinity, the difference is positive infinity.
- If the left operand is negative infinity and the right operand is positive infinity, the difference is negative infinity.
- If both operands are positive infinity, the difference is NaN.
- If both operands are negative infinity, the difference is NaN.
- If the left operand is an infinity value and the right operand is a finite value, the difference is the same infinity value as the left operand.
- If the left operand is a finite value and the right argument is an infinity value, the difference is the opposite infinity value of the right operand.
- If both operands are either positive zero or negative zero, the difference is positive zero.
- If the left operand is positive zero and the right operand is negative zero, the difference is positive zero.
- If the left operand is negative zero and the right operand is positive zero, the difference is negative zero.
- If neither operand is NaN nor an infinity value, the difference is rounded to the nearest representable value. If the magnitude of the difference is too large to be represented, the operation overflows and an infinity value of the appropriate sign is produced. If the magnitude of the difference is too small to be represented, the operation underflows and a zero value of the appropriate sign is produced.

References Arithmetic Types

String Concatenation Operator +

The string concatenation operator `+` produces a pure value that is a reference to a `String` object that it creates. The `String` object contains the concatenation of the operands; the characters of the left operand precede the characters of the right operand in the newly created string. The string concatenation + operator may appear in an additive expression.

Here is an example of some code that uses the string concatenation operator:

```
// format seconds into hours, minutes, and seconds
String formatTime(int t) {
    int minutes, seconds;
    seconds = t%60;
    t /= 60;
    minutes = t%60;
    return t/60 + ":" + minutes + ":" + seconds;
}
```

If neither operand of + is a reference to a `String` object, the operator is the arithmetic addition operator, not the string concatenation operator. Note that Java does not allow a program to define overloaded operators. However, the language defines the + operator to have a meaning that is fundamentally different from arithmetic addition if at least one of its operands is a `String` object.

The way in which Java decides if + means arithmetic addition or string concatenation means that the use of parentheses can alter the meaning of the + operator. Consider the following code:

```
int x = 3, y = 4;
System.out.println("x = " + x + ", y = " + y);
System.out.println("\"??\" + x + y ==> " + "??" + x + y);
System.out.println("\"??\" + (x + y) ==> " + "??"+ (x + y));
```

In the output from this code, you can see that the addition of parentheses changes the meaning of the last + from string concatenation to arithmetic addition:

```
x = 3, y = 4
"??" + x + y ==> ??34
"??" + (x + y) ==> ??7
```

If one of the operands of + is a `String` object and the other is not, the operand that is not a `String` object is converted to one using the following rules:

- If the operand is an object reference that is `null`, it is converted to the string literal `"null"`.
- If the operand is a non-`null` reference to an object that is not a string, the object's `toString()` method is called. The result of the conversion is the string value returned by the object's

`toString()` method, unless the return value is `null`, in which case the result of the conversion is the string literal `"null"`. Since the `Object` class defines a `toString()` method, every class in Java has such a method.

- If the type of the operand is `char`, the operand is converted to a reference to a `String` object that has a length of one and contains that character.

- If the type of the operand is an integer type other than `char`, the operand is converted to a base 10 string representation of its value. If the value is negative, the string value starts with a minus sign; if it is positive there is no sign character. If the value is zero, the result of the conversion is `"0"`. Otherwise, the string representation of the integer does not have any leading zeros.

- If the type of the operand is a floating-point type, the exact string representation depends on the value being converted. If the absolute value of `d` is greater than or equal to 10^-3 or less than or equal to 10^7, it is converted to a string with an optional minus sign (if the value is negative) followed by up to eight digits before the decimal point, a decimal point, and the necessary number of digits after the decimal point (but no trailing zero if there is more than one significant digit). There is always a minimum of one digit after the decimal point.

  Otherwise, the value is converted to a string with an optional minus sign (if the value is negative), followed by a single digit, a decimal point, the necessary number of digits after the decimal point (but no trailing zero if there is more than one significant digit), and the letter `E` followed by a plus or a minus sign and a base 10 exponent of at least one digit. Again, there is always a minimum of one digit after the decimal point.

  In addition, the values `NaN`, `NEGATIVE_INFINITY`, `POSITIVE_INFINITY`, `-0.0`, and `+0.0` are represented by the strings `"NaN"`, `"--Infinity"`, `"Infinity"`, `"--0.0"`, and `"0.0"`, respectively.

  Note that the specification for this conversion has changed as of Java 1.1. Prior to that release, the conversion provided a string representation that was equivalent to the `%g` format of the `printf` function in C. In addition, the string representations of the infinity values, the zero values, and NaN are not specified prior to Java 1.1.

- If the type of the operand is `boolean`, the value is converted to a reference to either the string literal `"true"` or the string literal `"false"`.

Java uses the `StringBuffer` object to implement string concatenation. Consider the following code:

```
String s, s1,s2;
s = s1 + s2;
```

To compute the string concatenation, Java compiler generates code equivalent to:

```
s = new StringBuffer().append(s1).append(s2).toString();
```

Consider another expression:

```
s = 1 + s1 + 2
```

In this case, the Java compiler generates code equivalent to:

```
s = new StringBuffer().append(1).append(s1).append(2).toString()
```

No matter how many strings are being concatenated in an expression, the expression always produces exactly one `StringBuffer` object and one `String` object.[3] From an efficiency standpoint, if you concatenate more than two strings, it may be more efficient to do so in a single expression, rather than in multiple expressions.

[3] Although an optimizing compiler should be smart enough to combine multiple concatenation expressions when it is advantageous, the compiler provided with Sun's reference implementation of Java does not do this.

References Arithmetic Addition Operator +; Object; String; StringBuffer

PREVIOUS | HOME | NEXT
Multiplicative Operators | BOOK INDEX | Shift Operators

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

4.13 Assignment Operators

Assignment operators set the values of variables and array elements. An assignment operator may appear in an assignment expression:

[Graphic: Figure from the text]

The actual assignment operator in an assignment expression can be the simple assignment operator `=` or one of the compound assignment operators shown below. All of the assignment operators are equal in precedence. Assignment operators are evaluated from right to left, so `a=b=c` is equivalent to `a=(b=c)`.

The left operand of an assignment operator must be an expression that produces a variable or an array element. The left operand of an assignment operator cannot be an expression that evaluates to a pure value, or a compile-time error occurs. So, for example, the left operand cannot be a `final` variable, since a `final` variable evaluates to a pure value, not a variable.

The assignment operator itself produces a pure value, not a variable or an array element. The pure value produced by an assignment operator is the value of the variable or array element after it has been set by the assignment operation. The type of this pure value is the type of the variable or array element.

The simple assignment operator `=` just sets the value of a variable or array element. It does not imply any other computation. The right operand of the simple assignment operator can be of any type, as long as that type is assignment-compatible with the type of the left operand, as described in the next section. If the right operand is not assignment-compatible, a compile-time error occurs.

The compound assignment operators are:

```
+=  -=   *=
/=  |=   &=
^=  %=   <<=
>>= >>>=
```

Both of the operands of a compound assignment operator must be of primitive types, or a compile-time error occurs. The one exception is if the left operand of the `+=` operator is of type `String`; in this case the right operand can be of any type.

A compound assignment operator combines a binary operator with the simple assignment operator `=`. Thus, to be assignment-compatible, the right operand of a compound assignment operator must be of a type that complies with the rules for the indicated binary operation. Otherwise, a compile-time error occurs. An assignment expression of the form:

```
e1 op = e2
```

is approximately equivalent to:

```
e1 = (type) ((e1) op (e2))
```

where *type* is the type of the expression `e1`. The only difference is that `e1` is only evaluated once in the expression that uses the compound assignment operator.

For example, consider the following code fragment:

```
j = 0;
a[0] = 3;
a[1]=6;
a[j++] += 2;
```

After this code is executed, `j` equals 1 and `a[0]` is 5. Contrast this with the following code:

```
j = 0;
a[0] = 3;
a[1]=6;
a[j++] = a[j++] + 2;
```

After this code is executed, `j` equals 2 and `a[0]` is 8 because `j++` is evaluated twice.

References Array Types; **UNKNOWN XREF**; Conditional Operator; Interface Variables; Local Variables; Order of Operations; Primitive Types; Reference Types; String; Unary Operators; Variables

Assignment Compatibility

Saying that one type of value is *assignment-compatible* with another type of value means that a value of the first type can be assigned to a variable of the second type. Here are the rules for assignment compatibility in Java:

- Every type is assignment-compatible with itself.
- The `boolean` type is not assignment-compatible with any other type.
- A value of any integer type can be assigned to a variable of any other integer type if the variable is of a type that allows it to contain the value without any loss of information.
- A value of any integer type can be assigned to a variable of any floating-point type, but a value of any floating-point type cannot be assigned to a variable of any integer type.
- A `float` value can be assigned to a `double` variable, but a `double` value cannot be assigned to a `float` variable.
- With a type cast, a value of any arithmetic type can be assigned to a variable of any other arithmetic type.
- Any reference can be assigned to a variable that is declared of type `Object`.
- A reference to an object can be assigned to a class-type reference variable if the class of the variable is the same class or a superclass of the class of the object.
- A reference to an object can be assigned to an interface-type reference variable if the class of the object implements the interface.
- A reference to an array can be assigned to an array variable if either of the following conditions is true:
  - Both array types contain elements of the same type.
  - Both array types contain object references and the type of reference contained in the elements of the array reference can be assigned to the type of reference contained in the elements of the variable.

Here's an example that illustrates the rules about assignment compatibility of arrays:

```
class Triangle extends Shape {...}
...
int[] i = new int[8];
int j[];
long l[];
short s[];
Triangle[] t;
Shape[] sh;
j = i;    // Okay
s = i;    // Error
l = i;    // Error
sh = t;   // Okay
t = sh;   // Error
```

Assigning `i` to `j` is fine because both variables are declared as references to arrays that contain `int` values. On the other hand, assigning `i` to `s` is an error because the variables are declared as references to arrays that contain different kinds of elements and these elements are not object references. Assigning `i` to `l` is an error for the same reason. Assigning `t` to `sh` is fine because the variables are declared as references to arrays that contain object references, and `sh[0]=t[0]` is legal. However, assigning `sh` to `t` is an error because `t[0]=sh[0]` is not legal.

It is not always possible for the compiler to determine if an assignment to an array element is legal; in these cases the assignment compatibility is checked at runtime. This situation can occur when a variable contains a reference to an array whose type of elements is specified by a class or interface name. In this case, it may not be possible to determine the actual type of the array elements until runtime. Consider the following example:

```
void foo (InputStream a[]) {
    a[0] = new FileInputStream("/dev/null");
}
```

Figure 4.1 shows the `InputStream` class and some of its subclasses in the `java.io` package.

Figure 4.1: InputStream and some of its classes

[Graphic: Figure 4-1]

Any array with elements that contain references to objects of class InputStream or any of its subclasses can be passed to the method foo() shown in the above example. For example:

```
FileInputStream f[] = new FileInputStream[3];
foo(f);
```

Since FileInputStream is a subclass of InputStream, the call to foo() does not cause any problems at runtime.

However, the following call to foo() is problematic:

```
DataInputStream f[] = new DataInputStream[3];
foo(f);
```

This call causes an ArrayStoreException to be thrown at runtime. Although DataInputStream is a subclass of InputStream, it is not a superclass of FileInputStream, so the array element assignment in foo() is not assignment-compatible.

References Arithmetic Types; Array Types; Boolean Type; Class Types; Interface Types

4.10 Bitwise/Logical Operators

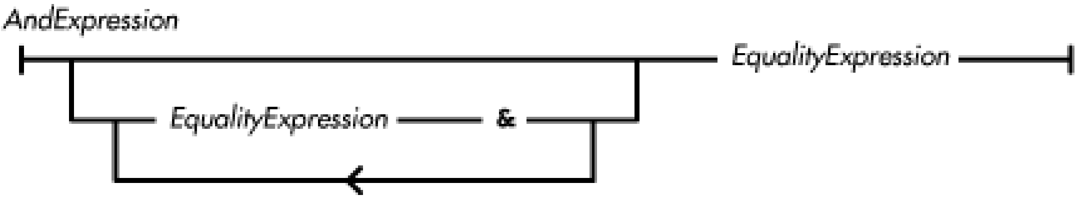
The bitwise/logical operators in Java are used for bitwise and logical AND (&), bitwise and logical exclusive OR (^), and bitwise and logical inclusive OR (|) operations. These operators have different precedence; the & operator has the highest precedence of the group and the | operator has the lowest. All of the operators are evaluated from left to right.

The unary operator ~ provides a bitwise negation operation.

References Bitwise Negation Operator ~; Order of Operations

Bitwise/Logical AND Operator &

The bitwise/logical AND operator & produces a pure value that is the AND of its operands. The & operator may appear in a bitwise or logical AND expression:



The bitwise/logical AND operator is evaluated from left to right. The operator never throws an exception.

Here is a code example that shows the use of the bitwise AND operator:

```
boolean isOdd(int x) {
    return (x & 1) == 1;
}
```

The operands of the bitwise/logical AND operator must both be of either an integer type or the type `boolean`, or a compile-time error occurs.

If both operands are of integer types, the operator performs a bitwise AND operation. The operator may perform type conversions on the operands:

- If either operand is of type `long`, the other operand is converted to `long` and the operation produces a `long` value.
- Otherwise, both operands are converted to `int` and the operation produces an `int` value.

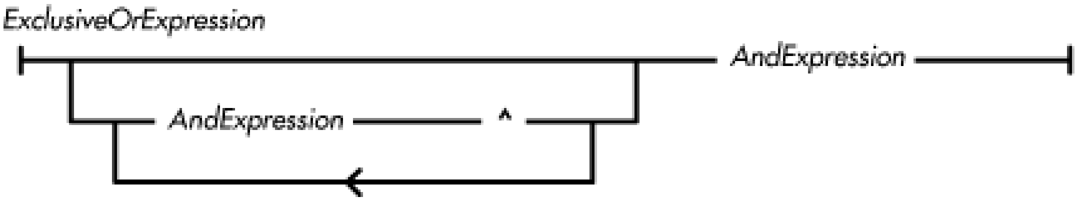
The bitwise AND operator produces a pure value that is the bitwise AND of its operands. If the corresponding bits in both of the converted operands are 1s, the corresponding bit in the result is a 1; otherwise the corresponding bit in the result is a 0.

If both operands are of type `boolean`, the operator performs a logical AND operation. The logical AND operation produces a pure value of type `boolean`. If both operands are `true`, the operation produces `true`; otherwise the operation produces `false`. This operator differs from the conditional AND operator (&&) because it always evaluates both of its operands, even if its left operand evaluates to `false`.

References Boolean AND Operator &&; Boolean Type; Equality Comparison Operators; Integer types; Order of Operations

Bitwise/Logical Exclusive OR Operator ^

The bitwise/logical exclusive OR operator ^ produces a pure value that is the exclusive OR of its operands. The ^ operator may appear in a bitwise or logical exclusive OR expression:



The bitwise/logical exclusive OR operator is evaluated from left to right. The operator never throws an exception.

The operands of the bitwise/logical exclusive OR operator must both be of either an integer type or the type `boolean`, or a compile-time error occurs.

If both operands are of integer types, the operator performs a bitwise exclusive OR operation. The operator may perform type conversions on the operands:

- If either operand is of type `long`, the other operand is converted to `long` and the operation produces a `long` value.
- Otherwise, both operands are converted to `int` and the operation produces an `int` value.

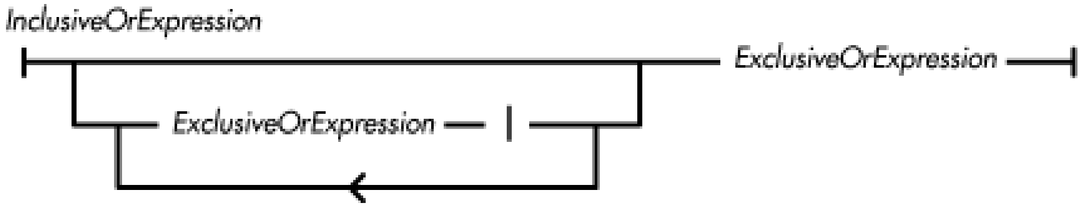
The bitwise exclusive OR operator produces a pure value that is the bitwise exclusive OR of its operands. If the corresponding bits in the converted operands are both 0 or both 1, the corresponding bit in the result is a 0; otherwise the corresponding bit in the result is a 1.

If both operands are of type `boolean`, the operator performs a logical exclusive OR operation. The logical exclusive OR operation produces a pure value of type `boolean`. If either, but not both, operands are `true`, the operation produces `true`; otherwise the operation produces `false`.

References Bitwise/Logical AND Operator &; Boolean Type; Integer types; Order of Operations

Bitwise/Logical Inclusive OR Operator |

The bitwise/logical inclusive OR operator `|` produces a pure value that is the inclusive OR of its operands. The `|` operator may appear in a bitwise or logical inclusive OR expression:



The bitwise/logical inclusive OR operator is evaluated from left to right. The operator never throws an exception.

Here is a code example that shows the use of the bitwise inclusive OR operator:

```
setFont("Helvetica", Font.BOLD | Font.ITALIC, 18);
```

The operands of the bitwise/logical inclusive OR operator must both be of either an integer type or the type `boolean`, or a compile-time error occurs.

If both operands are of integer types, the operator performs a bitwise inclusive OR operation. The operator may perform type conversions on the operands:

- If either operand is of type `long`, the other operand is converted to `long` and the operation produces a `long` value.
- Otherwise, both operands are converted to `int` and the operation produces an `int` value.

The bitwise inclusive OR operator produces a pure value that is the bitwise inclusive OR of its operands. If the corresponding bits in either or both of the converted operands are 1s, the corresponding bit in the result is a 1; otherwise the corresponding bit in the result is a 0.

If both operands are of type `boolean`, the operator performs a logical inclusive OR operation. The logical inclusive OR operation produces a pure value of type `boolean`. If either or both operands are `true`, the operation produces `true`; otherwise the operation produces `false`. This operator differs from the conditional OR operator (`||`) because it always evaluates both of its operands, even if its left operand evaluates to `true`.

References Boolean OR Operator ||; Boolean Type; Bitwise/Logical Exclusive OR Operator ^; Integer types; Order of Operations

PREVIOUS	HOME	NEXT
Equality Comparison Operators	BOOK INDEX	Boolean Operators

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Byte

Name

Byte

Synopsis

Class Name:

> `java.lang.Byte`

Superclass:

> `java.lang.Number`

Immediate Subclasses:

> None

Interfaces Implemented:

> None

Availability:

> New as of JDK 1.1

Description

The `Byte` class provides an object wrapper for a `byte` value. This is useful when you need to treat a `byte` value as an object. For example, there are a number of utility methods that take a reference to an `Object` as one of their arguments. You cannot specify a `byte` value for one of these arguments, but you can provide a reference to a `Byte` object that encapsulates the byte value. Furthermore, the Byte class is necessary as of JDK 1.1 to support the Reflection API and class literals.

The Byte class also provides a number of utility methods for converting byte values to other primitive types and for converting byte values to strings and vice versa.

Class Summary

```
public final class java.lang.Byte extends java.lang.Number {
  // Constants
  public static final byte MAX_VALUE;
  public static final byte MIN_VALUE;
  public static final Class TYPE;
  // Constructors
  public Byte(byte value);
  public Byte(String s);
  // Class Methods
  public static Byte decode(String nm);
  public static byte parseByte(String s);
  public static byte parseByte(String s, int radix);
  public static String toString(byte b);
  public static Byte valueOf(String s, int radix);
  public static Byte valueOf(String s);
  // Instance Methods
  public byte byteValue();
  public double doubleValue;
  public boolean equals(Object obj);
  public float floatValue
  public int hashCode();
  public int intValue();
  public long longValue();
  public short shortValue();
  public String toString();
}
```

Constants

MAX_VALUE

public static final byte MAX_VALUE= 127

The largest value that can be represented by a byte.

MIN_VALUE

public static final byte MIN_VALUE= -128

The smallest value that can be represented by a `byte`.

TYPE

public static final Class TYPE

The `Class` object that represents the primitive type `byte`. It is always true that `Byte.TYPE` == `byte.class`.

Constructors

Byte

public Byte(byte value)

Parameters

`value`

The `byte` value to be encapsulated by this object.

Description

Creates a `Byte` object with the specified byte value.

public Byte(String s) throws NumberFormatException

Parameters

`s`

The string to be made into a `Byte` object.

Throws

`NumberFormatException`

If the sequence of characters in the given `String` does not form a valid `byte` literal.

Description

Constructs a `Byte` object with the value specified by the given string. The string should consist of one or more digit characters. The digit characters can be preceded by a single `-' character. If the string contains any other characters, the constructor throws a `NumberFormatException`.

Class Methods

decode

public static Byte decode(String nm) throws NumberFormatException

Parameters

`nm`

A `String` representation of the value to be encapsulated by a `Byte` object. If the string begins with # or `0x`, it is a radix 16 representation of the value. If the string begins with `0`, it is a radix 8 representation of the value. Otherwise, it is assumed to be a radix 10 representation of the value.

Returns

A `Byte` object that encapsulates the given value.

Throws

`NumberFormatException`

If the `String` contains any non-digit characters other than a leading minus sign or the value represented by the `String` is less than `Byte.MIN_VALUE` or greater than `Byte.MAX_VALUE`.

Description

This method returns a `Byte` object that encapsulates the given value.

parseByte

public static byte parseByte(String s) throws NumberFormatException

Parameters

`s`

The `String` to be converted to a `byte` value.

Returns

The numeric value of the byte represented by the `String` object.

Throws

`NumberFormatException`

If the `String` does not contain a valid representation of a `byte` or the value represented by the `String` is less than `Byte.MIN_VALUE` or greater than `Byte.MAX_VALUE`.

Description

This method returns the numeric value of the `byte` represented by the contents of the given `String` object. The `String` must contain only decimal digits, except that the first character may be a minus sign.

`public static byte parseByte(String s, int radix) throws NumberFormatException`

Parameters

`s`

The `String` to be converted to a `byte` value.

`radix`

The radix used in interpreting the characters in the `String` as digits. This value must be in the range `Character.MIN_RADIX` to `Character.MAX_RADIX`. If `radix` is in the range 2 through 10, only characters for which the `Character.isDigit()` method returns `true` are considered to be valid digits. If `radix` is in the range 11 through 36, characters in the ranges `A' through `Z' and `a' through `z' may be considered valid digits.

Returns

The numeric value of the byte represented by the `String` object in the specified radix.

Throws

`NumberFormatException`

If the `String` does not contain a valid representation of a `byte`, `radix` is not in the appropriate range, or the value represented by the `String` is less than `Byte.MIN_VALUE` or greater than `Byte.MAX_VALUE`.

Description

This method returns the numeric value of the `byte` represented by the contents of the given `String` object in the specified radix. The `String` must contain only valid digits of the specified radix, except that the first character may be a minus sign. The digits are parsed in the specified radix to produce the numeric value.

toString

public String toString(byte b)

Parameters

`b`

The `byte` value to be converted to a string.

Returns

The string representation of the given value.

Description

This method returns a `String` object that contains the decimal representation of the given value.

This method returns a string that begins with `-' if the given value is negative. The rest of the string is a sequence of one or more of the characters `0', `1', `2', `3', `4', `5', `6', `7', `8', and `9'. This method returns "0" if its argument is `0`. Otherwise, the string returned by this method does not begin with "0" or "-0".

valueOf

public static Byte valueOf(String s) throws NumberFormatException

Parameters

`s`

The string to be made into a `Byte` object.

Returns

The `Byte` object constructed from the string.

Throws

`NumberFormatException`

If the `String` does not contain a valid representation of a `byte` or the value represented by the `String` is less than `Byte.MIN_VALUE` or greater than `Byte.MAX_VALUE`.

Description

Constructs a `Byte` object with the value specified by the given string. The string should consist of one or more digit characters. The digit characters can be preceded by a single `-' character. If the string contains any other characters, the method throws a `NumberFormatException`.

`public static Byte valueOf(String s, int radix) throws NumberFormatException`

Parameters

`s`

The string to be made into a `Byte` object.

`radix`

The radix used in converting the string to a value. This value must be in the range `Character.MIN_RADIX` to `Character.MAX_RADIX`.

Returns

The `Byte` object constructed from the string.

Throws

`NumberFormatException`

If the `String` does not contain a valid representation of a `byte`, radix is not in the appropriate range, or the value represented by the `String` is less than `Byte.MIN_VALUE` or greater than `Byte.MAX_VALUE`.

Description

Constructs a `Byte` object with the value specified by the given string in the specified radix. The string should consist of one or more digit characters or characters in the range `A' to `Z' or `a' to `z' that are considered digits in the given radix. The digit characters can be preceded by a single `-' character. If the string contains any other characters, the method throws a `NumberFormatException`.

Instance Methods

byteValue

public byte byteValue()

Returns

The value of this object as a `byte`.

Overrides

`Number.byteValue()`

Description

This method returns the value of this object as a `byte`.

doubleValue

public double doubleValue()

Returns

The value of this object as a `double`.

Overrides

`Number.doubleValue()`

Description

This method returns the value of this object as a `double`.

equals

public boolean equals(Object obj)

Parameters

`obj`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `obj` is an instance of `Byte` and it contains the same value as the object this method is associated with.

floatValue

public float floatValue()

Returns

The value of this object as a `float`.

Overrides

`Number.floatValue()`

Description

This method returns the value of this object as a `float`.

hashCode

public int hashCode()

Returns

A hashcode based on the `byte` value of the object.

Overrides

`Object.hashCode()`

Description

This method returns a hashcode computed from the value of this object.

intValue

public int intValue()

Returns

The value of this object as an `int`.

Overrides

`Number.intValue()`

Description

This method returns the value of this object as an `int`.

longValue

public long longValue()

Returns

The value of this object as a `long`.

Overrides

`Number.longValue()`

Description

This method returns the value of this object as a `long`.

shortValue

public short shortValue()

Returns

The value of this object as a `short`.

Overrides

`Number.shortValue()`

Description

This method returns the value of this object as a `short`.

toString

public String toString()

Returns

The string representation of the value of this object.

Overrides

`Object.toString()`

Description

This method returns a `String` object that contains the decimal representation of the value of this object.

This method returns a string that begins with `-` if the given value is negative. The rest of the string is a sequence of one or more of the characters `0', `1', `2', `3', `4', `5', `6', `7', `8', and `9'. This method returns "0" if its argument is `0`. Otherwise, the string returned by this method does not begin with "0" or "-0".

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

Character; Class; Double; Exceptions; Float; Integer literals; Integer types; Integer; Long; Number; Short; String

Character

Name

Character

Synopsis

Class Name:

```
java.lang.Character
```

Superclass:

```
java.lang.Object
```

Immediate Subclasses:

None

Interfaces Implemented:

```
java.io.Serializable
```

Availability:

JDK 1.0 or later

Description

The `Character` class provides an object wrapper for a `char` value. This is useful when you need to treat a char value as an object. For example, there are a number of utility methods that take a reference to an Object as one of their arguments. You cannot specify a char value for one of these arguments, but you can provide a reference to a Character object that encapsulates the char value. Furthermore, as of JDK 1.1, the Character class is necessary to support the Reflection API and class literals.

In Java, Character objects represent values defined by the Unicode standard. Unicode is defined by an organization called the Unicode Consortium. The defining document for Unicode is *The Unicode Standard, Version 2.0* (ISBN 0-201-48345-9). More recent information about Unicode is available at *http://unicode.org*. Appendix a, *The Unicode 2.0 Character Set*, contains a table that lists the characters defined by the Unicode 2.0 standard.

The Character class provides some utility methods, such as methods for determining the type (e.g., uppercase or lowercase, digit or letter) of a character and for converting from uppercase to lowercase. The logic for these utility methods is based on a Unicode attribute table that is part of the Unicode standard. That table is available at *ftp://unicode.org/pub/2.0-Update/UnicodeData-2.0.14.txt*.

Some of the methods in the Character class are concerned with characters that are digits; these characters are used by a number of other classes to convert strings that contain numbers into actual numeric values. The digit-related methods all use a radix value to interpret characters. The *radix* is the numeric base used to represent numbers as characters or strings. Octal is a radix 8 representation, while hexadecimal is a radix 16 representation. The methods that require a radix parameter use it to determine which characters should be treated as valid digits. In radix 2, only the characters `0' and `1' are valid digits. In radix 16, the characters `0' through `9', `a' through `z', and `A' through `Z' are considerd valid digits.

Class Summary

```
public final class java.lang.Character extends java.lang.Object
                                       implements java.io.Serializable {
    // Constants
    public final static byte COMBINING_SPACING_MARK;        // New in 1.1
    public final static byte CONNECTOR_PUNCTUATION;         // New in 1.1
    public final static byte CONTROL;                       // New in 1.1
    public final static byte CURRENCY_SYMBOL;               // New in 1.1
    public final static byte DASH_PUNCTUATION;              // New in 1.1
    public final static byte DECIMAL_DIGIT_NUMBER;          // New in 1.1
    public final static byte ENCLOSING_MARK;                // New in 1.1
    public final static byte END_PUNCTUATION;               // New in 1.1
    public final static byte FORMAT;                        // New in 1.1
    public final static byte LETTER_NUMBER;                 // New in 1.1
    public final static byte LINE_SEPARATOR;                // New in 1.1
    public final static byte LOWERCASE_LETTER;              // New in 1.1
    public final static byte MATH_SYMBOL;                   // New in 1.1
    public final static int MAX_RADIX;
    public final static char MAX_VALUE;
    public final static int MIN_RADIX;
    public final static char MIN_VALUE;
    public final static byte MODIFIER_LETTER;               // New in 1.1
    public final static byte MODIFIER_SYMBOL;               // New in 1.1
    public final static byte NON_SPACING_MARK;              // New in 1.1
    public final static byte OTHER_LETTER;                  // New in 1.1
    public final static byte OTHER_NUMBER;                  // New in 1.1
    public final static byte OTHER_PUNCTUATION;             // New in 1.1
    public final static byte OTHER_SYMBOL;                  // New in 1.1
    public final static byte PARAGRAPH_SEPARATOR;           // New in 1.1
    public final static byte PRIVATE_USE;                   // New in 1.1
    public final static byte SPACE_SEPARATOR;               // New in 1.1
    public final static byte START_PUNCTUATION;             // New in 1.1
    public final static byte SURROGATE;                     // New in 1.1
    public final static byte TITLECASE_LETTER;              // New in 1.1
    public final static Class TYPE;                         // New in 1.1
    public final static byte UNASSIGNED;                    // New in 1.1
    public final static byte UPPERCASE_LETTER;              // New in 1.1
    // Constructors
    public Character(char value);
    // Class Methods
    public static int digit(char ch, int radix);
    public static char forDigit(int digit, int radix);
    public static int getNumericValue(char ch);             // New in 1.1
    public static int getType(char ch);                     // New in 1.1
    public static boolean isDefined(char ch);
    public static boolean isDigit(char ch);
    public static boolean isIdentifierIgnorable(char ch);   // New in 1.1
    public static boolean isISOControl(char ch);            // New in 1.1
    public static boolean isJavaIdentifierPart(char ch);    // New in 1.1
    public static boolean isJavaIdentifierStart(char ch);   // New in 1.1
    public static boolean isJavaLetter(char ch);        // Deprecated in 1.1
    public static boolean isJavaLetterOrDigit(char ch); // Deprecated in 1.1
    public static boolean isLetter(char ch);
    public static boolean isLetterOrDigit(char ch);
    public static boolean isLowerCase(char ch);
    public static boolean isSpace(char ch);             // Deprecated in 1.1
    public static boolean isSpaceChar(char ch);             // New in 1.1
    public static boolean isTitleCase(char ch);
    public static boolean isUnicodeIdentifierPart(char ch); // New in 1.1
    public static boolean isUnicodeIdentifierStart(char ch);// New in 1.1
    public static boolean isUpperCase(char ch);
    public static boolean isWhitespace(char ch);            // New in 1.1
    public static char toLowerCase(char ch);
    public static char toTitleCase(char ch);
    public static char toUpperCase(char ch);
    // Instance Methods
    public char charValue();
```

```
    public boolean equals(Object obj);
    public int hashCode();
    public String toString();
}
```

Constants

COMBINING_SPACING_MARK

`public final static byte COMBINING_SPACING_MARK`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

CONNECTOR_PUNCTUATION

`public final static byte CONNECTOR_PUNCTUATION`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

CONTROL

`public final static byte CONTROL`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

CURRENCY_SYMBOL

`public final static byte CURRENCY_SYMBOL`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

DASH_PUNCTUATION

`public final static byte DASH_PUNCTUATION`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

DECIMAL_DIGIT_NUMBER

`public final static byte DECIMAL_DIGIT_NUMBER`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

ENCLOSING_MARK

`public final static byte ENCLOSING_MARK`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

END_PUNCTUATION

`public final static byte END_PUNCTUATION`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

FORMAT

`public final static byte FORMAT`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

LETTER_NUMBER

`public final static byte LETTER_NUMBER`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

LINE_SEPARATOR

`public final static byte LINE_SEPARATOR`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

LOWERCASE_LETTER

`public final static byte LOWERCASE_LETTER`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

MATH_SYMBOL

`public final static byte MATH_SYMBOL`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

MAX_RADIX

`public static final int MAX_RADIX = 36`

Description

The maximum value that can be specified for a radix.

MAX_VALUE

`public final static char MAX_VALUE = '\ufff'f`

Description

The largest value that can be represented by a `char`.

MIN_RADIX

`public static final int MIN_RADIX = 2`

Description

The minimum value that can be specified for a radix.

MIN_VALUE

`public final static char MIN_VALUE = '\u0000'`

Description

The smallest value that can be represented by a `char`.

MODIFIER_LETTER

`public final static byte MODIFIER_LETTER`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

MODIFIER_SYMBOL

`public final static byte MODIFIER_SYMBOL`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

NON_SPACING_MARK

`public final static byte NON_SPACING_MARK`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

OTHER_LETTER

`public final static byte OTHER_LETTER`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

OTHER_NUMBER

`public final static byte OTHER_NUMBER`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

OTHER_PUNCTUATION

`public final static byte OTHER_PUNCTUATION`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

OTHER_SYMBOL

`public final static byte OTHER_SYMBOL`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

PARAGRAPH_SEPARATOR

`public final static byte PARAGRAPH_SEPARATOR`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

PRIVATE_USE

`public final static byte PRIVATE_USE`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

SPACE_SEPARATOR

`public final static byte SPACE_SEPARATOR`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

START_PUNCTUATION

`public final static byte START_PUNCTUATION`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

SURROGATE

`public final static byte SURROGATE`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

TITLECASE_LETTER

`public final static byte TITLECASE_LETTER`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

TYPE

`public static final Class TYPE`

Availability

New as of JDK 1.1

Description

The `Class` object that represents the type `char`. It is always true that `Character.TYPE == char.class`.

UNASSIGNED

`public final static byte UNASSIGNED`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

UPPERCASE_LETTER

`public final static byte UPPERCASE_LETTER`

Availability

New as of JDK 1.1

Description

This constant can be returned by the `getType()` method as the general category of a Unicode character.

Constructors

Character

`public Character(char value)`

Parameters

`value`

The `char` value to be encapsulated by this object.

Description

Creates a `Character` object with the given `char` value.

Class Methods

digit

`public static int digit(char ch, int radix)`

Parameters

`ch`

A `char` value that is a legal digit in the given radix.

`radix`

The radix used in interpreting the specified character as a digit. If `radix` is in the range 2 through 10, only characters for which the `isDigit()` method returns `true` are considered to be valid digits. If `radix` is in the range 11 through 36, characters in the ranges `A' through `Z' and `a' through `z' may be considered valid digits.

Returns

The numeric value of the digit. This method returns `-1` if the value of `ch` is not considered a valid digit, if `radix` is less than `MIN_RADIX`, or if `radix` is greater than `MAX_RADIX`.

Description

Returns the numeric value represented by a digit character. For example, `digit('7',10)` returns 7. If the value of `ch` is not a valid digit, the method returns `-1`. For example, `digit('7',2)` returns `-1` because `'7'` is not a valid digit in radix 2. A number of methods in other classes use this method to convert strings that contain numbers to actual numeric values. The `forDigit()` method is an approximate inverse of this method.

If `radix` is greater than 10, characters in the range `A' to `A'+`radix`-11 are treated as valid digits. Such a character has the numeric value `ch`-`A'+10. By the same token, if `radix` is greater than 10, characters in the range `a' to `a'+`radix`-11 are treated as valid digits. Such a character has the numeric value `ch`-`a'+10.

forDigit

`public static char forDigit(int digit, int radix)`

Parameters

`digit`

The numeric value represented as a digit character.

`radix`

The radix used to represent the specified value.

Returns

The character that represents the digit corresponding to the specified numeric value. The method returns `\ 0' if `digit` is less than 0, if `digit` is equal to or greater than `radix`, if `radix` is less than `MIN_RADIX`, or if `radix` is greater than `MAX_RADIX`.

Description

This method returns the character that represents the digit corresponding to the specified numeric value. If `digit` is in the range 0 through 9, the method returns `0'+`digit`. If `digit` is in the range 10 through `MAX_RADIX`-1, the method returns `a'+`digit`-10. The method returns `\ 0' if `digit` is less than 0, if `digit` is equal to or greater than `radix`, if `radix` is less than `MIN_RADIX`, or if `radix` is greater than `MAX_RADIX`.

getNumericValue

`public static int getNumericValue(char ch)`

Availability

New as of JDK 1.1

Parameters

`ch`

A `char` value.

Returns

The Unicode numeric value of the character as a nonnegative integer. This method returns `-1` if the character has no numeric value; it returns `-2` if the character has a numeric value that is not a nonnegative integer, such as `1/2`.

Description

This method returns the Unicode numeric value of the specified character as a nonnegative integer.

getType

`public static int getType(char ch)`

Availability

New as of JDK 1.1

Parameters

`ch`

A `char` value.

Returns

An `int` value that represents the Unicode general category type of the character.

Description

This method returns the Unicode general category type of the specified character. The value corresponds to one of the general category constants defined by `Character`.

isDefined

`public static boolean isDefined(char ch)`

Parameters

`ch`

A `char` value to be tested.

Returns

`true` if the specified character has an assigned meaning in the Unicode character set; otherwise `false`.

Description

This method returns `true` if the specified character value has an assigned meaning in the Unicode character set.

isDigit

`public static boolean isDigit(char ch)`

Parameters

`ch`

A `char` value to be tested.

Returns

`true` if the specified character is defined as a digit in the Unicode character set; otherwise `false`.

Description

This method determines whether or not the specified character is a digit, based on the definition of the character in Unicode.

isIdentifierIgnorable

`public static boolean isIdentifierIgnorable(char ch)`

Availability

New as of JDK 1.1

Parameters

`ch`

A `char` value to be tested.

Returns

`true` if the specified character is ignorable in a Java or Unicode identifier; otherwise `false`.

Description

This method determines whether or not the specified character is ignorable in a Java or Unicode identifier. The following characters are ignorable in a Java or Unicode identifier:

`\u0000 - \u0008 \u000E - \u001B \u007F - \u009F`	ISO control characters that aren't whitespace
`\u200C - \u200F`	Join controls
`\u200A - \u200E`	Bidirectional controls
`\u206A - \u206F`	Format controls
`\uFEFF`	Zero-width no-break space

isISOControl

`public static boolean isISOControl(char ch)`

Availability

New as of JDK 1.1

Parameters

`ch`

A `char` value to be tested.

Returns

`true` if the specified character is an ISO control character; otherwise `false`.

Description

This method determines whether or not the specified character is an ISO control character. A character is an ISO control character if it falls in the range `\u0000` through `\u001F` or `\u007F` through `\u009F`.

isJavaIdentifierPart

`public static boolean isJavaIdentifierPart(char ch)`

Availability

New as of JDK 1.1

Parameters

`ch`

A `char` value to be tested.

Returns

`true` if the specified character can appear after the first character in a Java identifier; otherwise `false`.

Description

This method returns `true` if the specified character can appear in a Java identifier after the first character. A character is considered part of a Java identifier if and only if it is a letter, a digit, a currency symbol (e.g., $), a connecting punctuation character (e.g., _), a numeric letter (e.g., a Roman numeral), a combining mark, a nonspacing mark, or an ignorable control character.

isJavaIdentifierStart

`public static boolean isJavaIdentifierStart(char ch)`

Availability

New as of JDK 1.1

Parameters

`ch`

A `char` value to be tested.

Returns

`true` if the specified character can appear as the first character in a Java identifier; otherwise `false`.

Description

This method returns `true` if the specified character can appear in a Java identifier as the first character. A character is considered a start of a Java identifier if and only if it is a letter, a currency symbol (e.g., $), or a connecting punctuation character (e.g., _).

isJavaLetter

`public static boolean isJavaLetter(char ch)`

Availability

Deprecated as of JDK 1.1

Parameters

`ch`

A `char` value to be tested.

Returns

`true` if the specified character can appear as the first character in a Java identifier; otherwise `false`.

Description

This method returns `true` if the specified character can appear as the first character in a Java identifier. A character is considered a Java letter if and only if it is a letter, the character $, or the character _ . This method returns `false` for digits because digits are not allowed as the first character of an identifier.

This method is deprecated as of JDK 1.1. You should use `isJavaIdentifierStart()` instead.

isJavaLetterOrDigit

`public static boolean isJavaLetterOrDigit(char ch)`

Availability

Deprecated as of JDK 1.1

Parameters

`ch`

A `char` value to be tested.

Returns

`true` if the specified character can appear after the first character in a Java identifier; otherwise `false`.

Description

This method returns `true` if the specified character can appear in a Java identifier after the first character. A character is considered a Java letter or digit if and only if it is a letter, a digit, the character $, or the character _.

This method is deprecated as of JDK 1.1. You should use `isJavaIdentifierPart()` instead.

isLetter

`public static boolean isLetter(char ch)`

Parameters

`ch`

A `char` value to be tested.

Returns

`true` if the specified character is defined as a letter in the Unicode character set; otherwise `false`.

Description

This method determines whether or not the specified character is a letter, based on the definition of the character in Unicode. This method does not consider character values in ranges that have not been assigned meanings by Unicode to be letters.

isLetterOrDigit

`public static boolean isLetterOrDigit(char ch)`

Parameters

`ch`

A `char` value to be tested.

Returns

`true` if the specified character is defined as a letter in the Unicode character set; otherwise `false`.

Description

This method determines whether or not the specified character is a letter or a digit, based on the definition of the character in Unicode. There are some ranges that have not been assigned meanings by Unicode. If a character value is in one of these ranges, this method does not consider the character to be a letter.

isLowerCase

`public static boolean isLowerCase (char ch)`

Parameters

`ch`

A `char` value to be tested.

Returns

`true` if the specified character is defined as lowercase in the Unicode character set; otherwise `false`.

Description

This method determines whether or not the specified character is lowercase. Unicode defines a number of characters that do not have case mappings; if the specified character is one of these characters, the method returns `false`.

isSpace

`public static boolean isSpace(char ch)`

Availability

Deprecated as of JDK 1.1

Parameters

`ch`

A char value to be tested.

Returns

true if the specified character is defined as whitespace in the ISO-Latin-1 character set; otherwise false.

Description

This method determines whether or not the specified character is whitespace. This method recognizes the whitespace characters shown in the following table.

\u0009	Horizontal tab
\u000A	Newline
\u000C	Formfeed
\u000D	Carriage return
\u0020 ` `	Space

This method is deprecated as of JDK 1.1. You should use isWhitespace() instead.

isSpaceChar

public static boolean isSpaceChar(char ch)

Availability

New as of JDK 1.1

Parameters

ch

A char value to be tested.

Returns

true if the specified character is a Unicode 2.0 space characters; otherwise false.

Description

This method determines if the specified character is a space character according to the Unicode 2.0 specification. A character is considered to be a Unicode space character if and only if it has the general category "Zs", "Zl", or "Zp" in the Unicode specification.

isTitleCase

`public static boolean isTitleCase(char ch)`

Parameters

`ch`

A `char` value to be tested.

Returns

`true` if the specified character is defined as titlecase in the Unicode character set; otherwise `false`.

Description

This method determines whether or not the specified character is a titlecase character. Unicode defines a number of characters that do not have case mappings; if the specified character is one of these characters, the method returns `false`.

Many characters are defined by the Unicode standard as having upper- and lowercase forms. There are some characters defined by the Unicode standard that also have a titlecase form. The glyphs for these characters look like a combination of two Latin letters. The titlecase form of these characters has a glyph that looks like a combination of an uppercase Latin character and a lowercase Latin character; this case should be used when the character appears as the first character of a word in a title. For example, one of the Unicode characters that has a titlecase form looks like the letter `D' followed by the letter `Z'. Here is what the three forms of this letter look like:

Uppercase	`DZ'
Titlecase	`Dz'
Lowercase	`dz'

isUnicodeIdentifierPart

`public static boolean isUnicodeIdentifierPart(char ch)`

Availability

New as of JDK 1.1

Parameters

`ch`

A `char` value to be tested.

Returns

`true` if the specified character can appear after the first character in a Unicode identifier; otherwise `false`.

Description

This method returns `true` if the specified character can appear in a Unicode identifier after the first character. A character is considered part of a Unicode identifier if and only if it is a letter, a digit, a connecting punctuation character (e.g., _), a numeric letter (e.g., a Roman numeral), a combining mark, a nonspacing mark, or an ignorable control character.

isUnicodeIdentifierStart

```
public static boolean isUnicodeIdentifierStart(char ch)
```

Availability

New as of JDK 1.1

Parameters

`ch`

A `char` value to be tested.

Returns

`true` if the specified character can appear as the first character in a Unicode identifier; otherwise `false`.

Description

This method returns `true` if the specified character can appear in a Unicode identifier as the first character. A character is considered a start of a Unicode identifier if and only if it is a letter.

isUpperCase

```
public static boolean isUpperCase(char ch)
```

Parameters

ch

A `char` value to be tested.

Returns

`true` if the specified character is defined as uppercase in the Unicode character set; otherwise `false`.

Description

This method determines whether or not the specified character is uppercase. Unicode defines a number of characters that do not have case mappings; if the specified character is one of these characters, the method returns `false`.

isWhitespace

`public static boolean isWhitespace(char ch)`

Availability

New as of JDK 1.1

Parameters

ch

A `char` value to be tested.

Returns

`true` if the specified character is defined as whitespace according to Java; otherwise `false`.

Description

This method determines whether or not the specified character is whitespace. This method recognizes the following as whitespace:

Unicode category "Zs" except `\u00A0` and `\uFEFF`	Unicode space separators except no-break spaces
Unicode category "Zl"	Unicode line separators
Unicode category "Zp"	Unicode paragraph separators
`\u0009`	Horizontal tab

`\u000A`	Linefeed
`\u000B`	Vertical tab
`\u000C`	Formfeed
`\u000D`	Carriage return
`\u001C`	File separator
`\u001D`	Group separator
`\u001E`	Record separator
`\u001F`	Unit separator

toLowerCase

`public static char toLowerCase(char ch)`

Parameters

`ch`

A `char` value to be converted to lowercase.

Returns

The lowercase equivalent of the specified character, or the character itself if it cannot be converted to lowercase.

Description

This method returns the lowercase equivalent of the specified character value. If the specified character is not uppercase or if it has no lowercase equivalent, the character is returned unmodified. The Unicode attribute table determines if a character has a mapping to a lowercase equivalent.

Some Unicode characters in the range `\u2000` through `\u2FFF` have lowercase mappings. For example, `\u2160` (Roman numeral one) has a lowercase mapping to `\u2170` (small Roman numeral one). The `toLowerCase()` method maps such characters to their lowercase equivalents even though the method `isUpperCase()` does not return `true` for such characters.

toTitleCase

`public static char toTitleCase(char ch)`

Parameters

`ch`

A `char` value to be converted to titlecase.

Returns

The titlecase equivalent of the specified character, or the character itself if it cannot be converted to titlecase.

Description

This method returns the titlecase equivalent of the specified character value. If the specified character has no titlecase equivalent, the character is returned unmodified. The Unicode attribute table is used to determine the character's titlecase equivalent.

Many characters are defined by the Unicode standard as having upper- and lowercase forms. There are some characters defined by the Unicode standard that also have a titlecase form. The glyphs for these characters look like a combination of two Latin letters. The titlecase form of these characters has a glyph that looks like a combination of an uppercase Latin character and a lowercase Latin character; this case should be used when the character appears as the first character of a word in a title. For example, one of the Unicode characters that has a titlecase form looks like the letter `D' followed by the letter `Z'. Here is what the three forms of this letter look like:

Uppercase	`DZ'
Titlecase	`Dz'
Lowercase	`dz'

toUpperCase

`public static char toUpperCase(char ch)`

Parameters

`ch`

A `char` value to be converted to lowercase.

Returns

The uppercase equivalent of the specified character, or the character itself if it cannot be converted to uppercase.

Description

This method returns the uppercase equivalent of the specified character value. If the specified character is not lowercase or if it has no uppercase equivalent, the character is returned unmodified. The Unicode attribute table determines if a character has a mapping to an uppercase equivalent.

Some Unicode characters in the range `\u2000` through `\u2FFF` have uppercase mappings. For example, `\u2170` (small Roman numeral one) has a lowercase mapping to `\u2160` (Roman numeral one). The `toUpperCase()` method maps such characters to their uppercase equivalents even though the method `isLowerCase()` does not return `true` for such characters.

Instance Methods

charValue

```
public char charValue()
```

Returns

The `char` value contained by the object.

equals

```
public boolean equals(Object obj)
```

Parameters

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `obj` is an instance of `Character`, and it contains the same value as the object this method is associated with.

hashCode

```
public int hashCode()
```

Returns

> A hashcode based on the `char` value of the object.

Overrides

> `Object.hashCode()`

toString

```
public String toString()
```

Returns

> A `String` of length one that contains the character value of the object.

Overrides

> `Object.toString()`

Description

> This method returns a string representation of the `Character` object.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

Character literals; Class; Integer types; Object

PREVIOUS HOME NEXT
Byte BOOK INDEX Class

4.12 Conditional Operator

The conditional operator (`? :`) is a ternary operator. The operator selects one of two expressions for evaluation, based on the value of its first operand. In this way, the conditional operator is similar to an `if` statement. A conditional operator may appear in a conditional expression:

[Graphic: Figure from the text]

The conditional operator produces a pure value. Conditional expressions group from right to left. Consider the following expression:

```
g?f:e?d:c?b:a
```

It is equivalent to

```
g?f:(e?d:(c?b:a))
```

The first operand of the conditional operator must be of type `boolean`, or a compile-time error occurs. If the first operand evaluates to `true`, the operator evaluates the second operand (i.e., the one following the `?`) and produces the pure value of that expression. Otherwise, if the first operand evaluates to `false`, the operator evaluates the third operand (i.e., the one following the `:`) and produces the pure value of that expression. Note that the conditional operator evaluates either its second operand or its third operand, but not both.

The second and third operands of the conditional operator may be of any type, but they must both be of the same kind of type or a compile-time error occurs. If one operand is of an arithmetic type, the other must also be of an arithmetic type. If one operand is of type `boolean`, the other must also be of type `boolean`. If one operand is a reference type, the other must also be a reference type. Note that neither the second nor the third operand can be an expression that invokes a `void` method.

The types of the second and third operands determine the type of pure value that the conditional operator produces. If the second and third operands are of different types, the operator may perform a type conversion on the operand that it evaluates. The operator does this to ensure that it always produces the same type of result for a given expression, regardless of the value of its first operand.

If the second and third operands are both of arithmetic types, the conditional operator determines the type of value it produces as follows:[6]

- If both operands are of the same type, the conditional operator produces a pure value of that type.

- If one operand is of type `short` and the other operand is of type `byte`, the conditional operator produces a `short` value.
- If one operand is of type `short`, `char`, or `byte` and the other operand is a constant expression that can be represented as a value of that type, the conditional operator produces a pure value of that type.
- Otherwise, if either operand is of type `double`, the operator produces a `double` value.
- Otherwise, if either operand is of type `float`, the operator produces a `float` value.
- Otherwise, if either operand is of type `long`, the operator produces a `long` value.
- Otherwise, if either operand is of type `int`, the operator produces an `int` value.

If the second and third operands are both of type `boolean`, the conditional operator produces a pure `boolean` value.

If the second and third operands are both reference types, the conditional operator determines the type of value it produces as follows:

- If both operands are `null`, the conditional operator produces the pure value `null`.
- Otherwise, if exactly one of the operands is `null`, the conditional operator produces a value of the type of the other operand.

[6] Some of these rules are different from the way it is done in C/C++. In those languages, integer data of types smaller than `int` are always converted to `int` when they appear in any expression.

- Otherwise, it must be possible to cast the value of one of the operands to the type of the other operand, or a compile-time error occurs. The conditional operator produces a value of the type that would be the target of the cast.

References Arithmetic Types; Boolean Type; Boolean OR Operator ||; *Expression* 4; Order of Operations; Reference Types

4.9 Equality Comparison Operators

The equality comparison operators in Java are used for equal-to (==) and not-equal-to (!=) comparison operations. The equality comparison operators may appear in an equality expression:

[Graphic: Figure from the text]

The equality comparison operators are equal in precedence and are evaluated from left to right. The == and != comparison operators can perform numerical comparisons, `boolean` comparisons, and reference type comparisons. Both of these operators produce `boolean` values.

References Relational Comparison Operators; Order of Operations

Equal-To Operator ==

The equal-to operator == performs a comparison between its operands and returns a `boolean` value. It returns the pure value `true` if the operands are equal to each other; otherwise it returns the pure value `false`. The == operator may appear as part of an equality expression. The equal-to operator never throws an exception.

The operands of == may be of any type, but they must both be of the same kind of type or a compile-time error occurs. If one operand is of an arithmetic type, the other must also be of an arithmetic type. If one operand is of type `boolean`, the other must also be of type `boolean`. If one operand is a reference type, the other must also be a reference type. Note that neither operand can be an expression that invokes a `void` method.

If both operands are of arithmetic types, then the operator performs an arithmetic equality comparison.

The operator may perform type conversions on the operands:

- If either operand is of type `double`, then the other operand is converted to `double`.
- Otherwise, if either operand is of type `float`, the other operand is converted to `float`.
- Otherwise, if either operand is of type `long`, the other operand is converted to `long`.
- Otherwise, both operands are converted to `int`.

The equality comparison of any two arithmetic values produces `true` if and only if both operands are the same value; otherwise the comparison produces `false`. The comparison of floating-point data is governed by the following additional rules:

- If either operand is not-a-number (NaN), the comparison produces `false`.
- Positive infinity is a distinct value that is equal to itself, and not equal to any other value.
- Negative infinity is a distinct value that is equal to itself, and not equal to any other value.
- Positive and negative zero are treated as equal, so `-0.0==0.0` produces `true`.

If both operands are `boolean` values, the operator performs a Boolean equality comparison. The comparison produces `true` if both operands are `true` or both operands are `false`. Otherwise, the comparison produces `false`.

If both operands are reference types, the operator performs an object equality comparison. In order to perform this type of comparison, it must be possible to cast the value of one of the operands to the type of the other operand, or a compile-time error occurs. The comparison produces `true` if both of its operands refer to the same object or if both of its operands are `null`; otherwise the comparison produces `false`.

Because the == operator determines if two objects are the same object, it is not appropriate for comparisons that need to determine if two objects have the same contents. For example, if you need to know whether two `String` objects contain the same sequences of characters, the == operator is inappropriate. You should use the `equals()` method instead:[4]

> [4] This is similar to the difference in C between writing `string1==string2` and `strcmp(string1, string2)==0`.

```
string1.equals (string2)  // Compares contents of strings
```

```
string1 == string2        // Compares actual string objects
```

References Arithmetic Types; Boolean Type; Reference Types

Not-Equal-To-Operator !=

The not-equal-to operator `!=` performs a comparison between its operands and returns a `boolean` value. It returns the pure value `true` if the operands are not equal to each other; otherwise it returns the pure value `false`. The `!=` operator may appear as part of an equality expression. The not-equal-to operator never throws an exception.

The operands of `!=` may be of any type, but they must both be of the same kind of type or a compile-time error occurs. If one operand is of an arithmetic type, the other must also be of an arithmetic type. If one operand is of type `boolean`, the other must also be of type `boolean`. If one operand is a reference type, the other must also be a reference type. Note that neither operand can be an expression that invokes a `void` method.

If both operands are of arithmetic types, the operator performs an arithmetic inequality comparison. The operator may perform type conversions on the operands:

- If either operand is of type `double`, then the other operand is converted to `double`.
- Otherwise, if either operand is of type `float`, the other operand is converted to `float`.
- Otherwise, if either operand is of type `long`, the other operand is converted to `long`.
- Otherwise, both operands are converted to `int`.

The inequality comparison of any two arithmetic values produces `true` if and only if both operands are not the same value; otherwise the comparison produces `false`. The comparison of floating-point data is governed by the following additional rules:

- If either operand is not-a-number (NaN), the comparison produces `true`. NaN is the only value that compares as not equal to itself.
- Positive infinity is a distinct value that is equal to itself, and not equal to any other value.
- Negative infinity is a distinct value that is equal to itself, and not equal to any other value.
- Positive and negative zero are treated as equal, so `-0.0!=0.0` produces `false`.

If both operands are `boolean` values, the operator performs a Boolean inequality comparison. The comparison produces `false` if both operands are `true` or both operands are `false`. Otherwise, the comparison produces `true`.

If both operands are reference types, the operator performs an object equality comparison. In order to perform this type of comparison, it must be possible to cast the value of one of the operands to the type of the other operand, or a compile-time error occurs. The comparison produces `true` if both of its operands refer to different objects and if both of its operands are not `null`; otherwise the comparison produces `false`.

Because the `!=` operator determines if two objects are different objects, it is not appropriate for comparisons that need to determine if two objects have different contents. For example, if you need to know whether two `String` objects contain different sequences of characters, the `!=` operator is inappropriate. You should use the `equals()` method instead:[5]

> [5] This is similar to the difference in C between writing `string1!=string2` and `strcmp(string1, string2)!=0`.

```
!string1.equals (string2) // Compares contents of strings
string1 != string2        // Compares actual string objects
```

References Arithmetic Types; Boolean Type; Reference Types

4.3 Increment/Decrement Operators

The ++ operator is used to increment the contents of a variable or an array element by one, while the -- operator is used to decrement such a value by one. The operand of ++ or -- must evaluate to a variable or an array element; it cannot be an expression that produces a pure value. For example, the following operations succeed because the operand of the ++ operator produces a variable:

```
int g = 0;
g++;
```

However, the following uses of ++ generate error messages:

```
final int h = 23;
h++;
5++;
```

The expression h++ produces an error because h is declared final, which means that its value cannot be changed. The expression 5++ generates an error message because 5 is a literal value, not a variable.

The increment and decrement operators can be used in both postfix expressions (e.g., i++ or i--) and in prefix expressions (e.g., ++i or --i). Although both types of expression have the same side effect of incrementing or decrementing a variable, they differ in the values that they produce. A postfix expression produces a pure value that is the value of the variable before it is incremented or decremented, while a prefix expression produces a pure value that is the value of the variable after it has been incremented or decremented. For example, consider the following code fragment:

```
int i = 3, j = 3;
System.out.println( "i++ produces " + i++);
System.out.println( "++j produces " + ++j);
```

The above code fragment produces the following output:

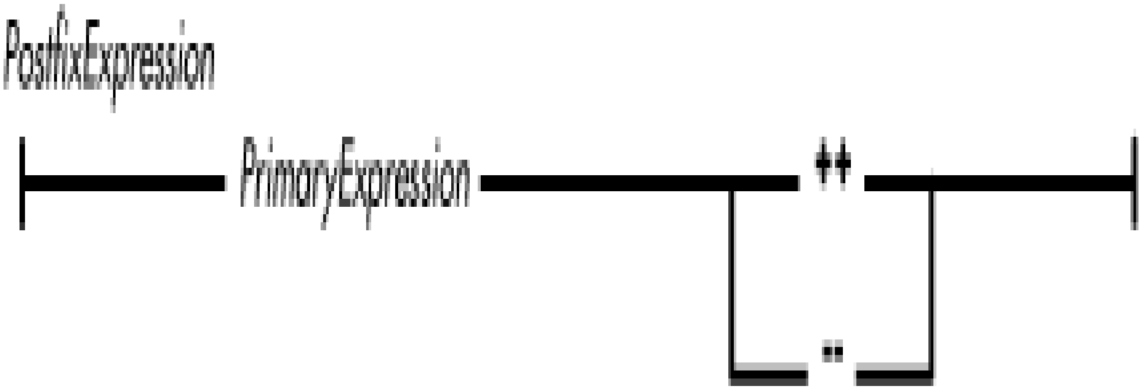
```
i++ produces 3
++j produces 4
```

After the code fragment has been evaluated, both `i` and `j` have the value 4.

In essence, what you need to remember is that a prefix expression performs its increment or decrement before producing a value, while a postfix expression performs its increment or decrement after producing a value.

Postfix Increment/Decrement Operators

A postfix increment/decrement expression is a primary expression that may be followed by either a `++` or a `- -`:



The postfix increment and decrement operators are equal in precedence and are effectively non-associative.

If a postfix expression includes a `++` or `- -`, the primary expression must produce a variable or an array element of an arithmetic type. The postfix increment operator (`++`) has the side effect of incrementing the contents of the variable or array element by one. The postfix decrement operator (`- -`) has the side effect of decrementing the contents of the variable or array element by one.

The data type of the value produced by a postfix increment/decrement operator is the same as the data type of the variable or array element produced by the primary expression. A postfix increment/decrement operator produces the original pure value stored in the variable or array element before it is incremented or decremented.

The following is an example of using a postfix decrement operator:

```
char j = '\u0100';
while (j-- > 0)        // call doit for char values
```

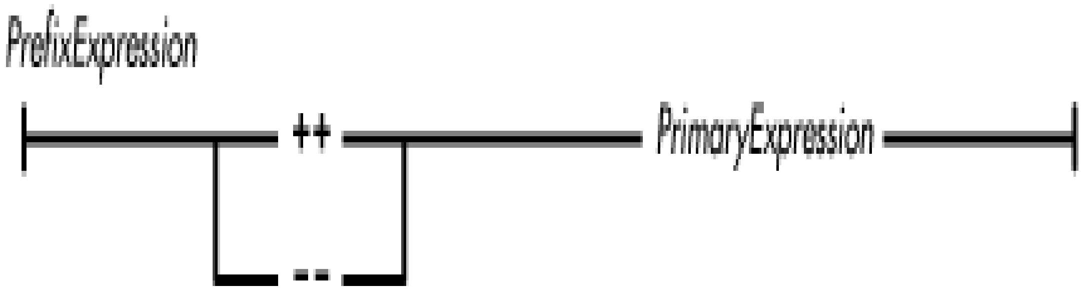
```
    doit(j);           // '\u00ff' through '\u0000'
```

This example works because Java treats `char` as an arithmetic data type.

References Arithmetic Types; Order of Operations; Primary Expressions

Prefix Increment/Decrement Operators

A prefix increment/decrement expression is a primary expression that may be preceded by either a `++` or a `- -`:



The prefix increment and decrement operators are equal in precedence and are effectively non-associative.

If a prefix expression includes a `++` or `- -`, the primary expression must produce a variable or an array element of an arithmetic type. The prefix increment operator (`++`) has the side effect of incrementing the contents of the variable or array element by one. The prefix decrement operator (`- -`) has the side effect of decrementing the contents of the variable or array element by one.

The data type of the value produced by a prefix increment/decrement operator is the same as the data type of the variable or array element produced by the primary expression. A prefix increment/decrement operator produces the pure value stored in the variable or array element after it has been incremented or decremented.

Here's an example of using a prefix increment operator:

```
void foo(int a[]) {
    int j = -1;
    while (++j < a.length)    // call doit for each element
        doit(a[j]);              // of a
    }
```

References Arithmetic Types; Order of Operations; Primary Expressions

Integer

Name

Integer

Synopsis

Class Name:

`java.lang.Integer`

Superclass:

`java.lang.Number`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The `Integer` class provides an object wrapper for an `int` value. This is useful when you need to treat an `int` value as an object. For example, there are a number of utility methods that take a reference to an `Object` as one of their arguments. You cannot specify an `int` value for one of these arguments, but you can provide a reference to an `Integer` object that encapsulates the `int` value. Also, as of JDK 1.1, the `Integer` class is necessary to support the

Reflection API and class literals.

The `Integer` class also provides a number of utility methods for converting `int` values to other primitive types and for converting `int` values to strings and vice versa.

Class Summary

```
public final class java.lang.Integer extends java.lang.Number {
    // Constants
    public static final int MAX_VALUE;
    public static final int MIN_VALUE;
    public final static Class TYPE;                    // New in 1.1
    // Constructors
    public Integer(int value);
    public Integer(String s);
    // Class Methods
    public static Integer decode(String nm)            // New in 1.1
    public static Integer getInteger(String nm);
    public static Integer getInteger(String nm, int val);
    public static Integer getInteger(String nm, Integer val);
    public static int parseInt(String s);
    public static int parseInt(String s, int radix;
    public static String toBinaryString(long i);
    public static String toHexString(long i);
    public static String toOctalString(long i);
    public static String toString(int i);
    public static String toString(int i, int radix);
    public static Integer valueOf(String s);
    public static Integer valueOf(String s, int radix);
    // Instance Methods
    public byte byteValue();                           // New in 1.1
    public double doubleValue();
    public boolean equals(Object obj);
    public float floatValue();
    public int hashCode();
    public int intValue();
    public long longValue();
    public short shortValue();                         // New in 1.1
    public String toString();
}
```

Constants

MAX_VALUE

`public static final int MAX_VALUE = 0x7fffffff // 2147483647`

Description

The largest value that can be represented by an `int`.

MIN_VALUE

`public static final int MIN_VALUE = 0x80000000 // -2147483648`

Description

The smallest value that can be represented by an `int`.

TYPE

`public static final Class TYPE`

Availability

New as of JDK 1.1

Description

The `Class` object that represents the type `int`. It is always true that `Integer.TYPE` == `int.class`.

Constructors

Integer

`public Integer(int value)`

Parameters

`value`

The `int` value to be encapsulated by this object.

Description

Creates an `Integer` object with the specified `int` value.

`public Integer(String s) throws NumberFormatException`

Parameters

`s`

The string to be made into an `Integer` object.

Throws

`NumberFormatException`

If the sequence of characters in the given `String` does not form a valid `int` literal.

Description

Constructs an `Integer` object with the value specified by the given string. The string should consist of one or more digit characters. The digit characters can be preceded by a single `` `-' `` character. If the string contains any other characters, the constructor throws a `NumberFormatException`.

Class Methods

decode

`public static Integer decode(String nm)`

Availability

New as of JDK 1.1

Parameters

`nm`

A `String` representation of the value to be encapsulated by an `Integer` object. If the string begins with # or `0x`, it is a radix 16 representation of the value. If the string begins with `0`, it is a radix 8 representation of the value. Otherwise, it is assumed to be a radix 10 representation of the value.

Returns

An `Integer` object that encapsulates the given value.

Throws

`NumberFormatException`

If the `String` contains any nondigit characters other than a leading minus sign or the value represented by the `String` is less than `Integer.MIN_VALUE` or greater than `Integer.MAX_VALUE`.

Description

This method returns an `Integer` object that encapsulates the given value.

getInteger

`public static Integer getInteger(String nm)`

Parameters

`nm`

The name of a system property.

Returns

The value of the system property as an `Integer` object, or an `Integer` object with the value 0 if the named property does not exist or cannot be parsed.

Description

This method retrieves the value of the named system property and returns it as an `Integer` object. The method obtains the value of the system property as a `String` using `System.getProperty()`.

If the value of the property begins with `0x` or `#` and is not followed by a minus sign, the rest of the value is parsed as a hexadecimal integer. If the value begins with `0`, it's parsed as an octal integer; otherwise it's parsed as a decimal integer.

`public static Integer getInteger(String nm, int val)`

Parameters

`nm`

The name of a system property.

`val`

A default `int` value for the property.

Returns

The value of the system property as an `Integer` object, or an `Integer` object with the value `val` if the named property does not exist or cannot be parsed.

Description

This method retrieves the value of the named system property and returns it as an `Integer` object. The method obtains the value of the system property as a `String` using `System.getProperty()`.

If the value of the property begins with `0x` or `#` and is not followed by a minus sign, the rest of the value is parsed as a hexadecimal integer. If the value begins with `0`, it's parsed as an octal integer; otherwise it's parsed as a decimal integer.

`public static Integer getInteger(String nm, Integer val)`

Parameters

`nm`

The name of a system property.

`val`

A default `Integer` value for the property.

Returns

The value of the system property as an `Integer` object, or the `Integer` object `val` if the named property does not exist or cannot be parsed.

Description

This method retrieves the value of the named system property and returns it as an `Integer` object. The method obtains the value of the system property as a `String` using `System.getProperty()`.

If the value of the property begins with `0x` or `#` and is not followed by a minus sign, the rest of the value is parsed as a hexadecimal integer. If the value begins with `0`, it's parsed as an octal integer; otherwise it's as a decimal integer.

parseInt

`public static int parseInt(String s) throws NumberFormatException`

Parameters

`s`

The `String` to be converted to an `int` value.

Returns

The numeric value of the integer represented by the `String` object.

Throws

`NumberFormatException`

If the `String` does not contain a valid representation of an integer.

Description

This method returns the numeric value of the integer represented by the contents of the given `String` object. The `String` must contain only decimal digits, except that the first character may be a minus sign.

`public static int parseInt(String s, int radix) throws NumberFormatException`

Parameters

`s`

The `String` to be converted to an `int` value.

`radix`

The radix used in interpreting the characters in the `String` as digits. This value must be in the range `Character.MIN_RADIX` to `Character.MAX_RADIX`. If `radix` is in the range 2 through 10, only characters for which the `Character.isDigit()` method returns `true` are considered to be valid digits. If `radix` is in the range 11 through 36, characters in the ranges `A' through `Z' and `a' through `z' may be considered valid digits.

Returns

The numeric value of the integer represented by the `String` object in the specified radix.

Throws

`NumberFormatException`

If the `String` does not contain a valid representation of an integer, or `radix` is not in the appropriate range.

Description

This method returns the numeric value of the integer represented by the contents of the given `String` object in the specified radix. The `String` must contain only valid digits of the specified radix, except that the first character may be a minus sign. The digits are parsed in the specified radix to produce the numeric value.

toBinaryString

`public static String toBinaryString(int value)`

Parameters

`value`

The `int` value to be converted to a string.

Returns

A string that contains the binary representation of the given value.

Description

This method returns a `String` object that contains the representation of the given value as an unsigned binary number. To convert the given value to an unsigned quantity, the method simply uses the value as if it were not negative. In other words, if the given value is negative, the method adds 2^32 to it. Otherwise the value is used as it is.

The string returned by this method contains a sequence of one or more `0' and `1' characters. The method returns "0" if its argument is `0`. Otherwise, the string returned by this method begins with `1'.

toHexString

`public static String toHexString(int value)`

Parameters

`value`

The `int` value to be converted to a string.

Returns

A string that contains the hexadecimal representation of the given value.

Description

This method returns a `String` object that contains the representation of the given value as an unsigned hexadecimal number. To convert the given value to an unsigned quantity, the method simply uses the value as if it were not negative. In other words, if the given value is negative, the method adds 2^32 to it. Otherwise the value is used as it is.

The string returned by this method contains a sequence of one or more of the characters `0', `1', `2', `3', `4', `5', `6', `7', `8', `9', `a', `b', `c', `d', `e', and `f'. The method returns "0" if its argument is `0`. Otherwise, the string returned by this method does not begin with `0'.

To produce a string that contains upper- instead of lowercase letters, use the `String.toUpperCase()` method.

toOctalString

`public static String toOctalString(int value)`

Parameters

`value`

The `int` value to be converted to a string.

Returns

A string that contains the octal representation of the given value.

Description

This method returns a `String` object that contains a representation of the given value as an unsigned octal number. To convert the given value to an unsigned quantity, the method simply uses the value as if it were not negative. In other words, if the given value is negative, the method adds 2^32 to it. Otherwise the value is used as it is.

The string returned by this method contains a sequence of one or more of the characters `0', `1', `2', `3', `4', `5', `6', and `7'. The method returns "0" if its argument is `0`. Otherwise, the string returned by this method does not begin with `0'.

toString

`public static String toString(int i)`

Parameters

`i`

The `int` value to be converted to a string.

Returns

The string representation of the given value.

Description

This method returns a `String` object that contains the decimal representation of the given value.

This method returns a string that begins with `-' if the given value is negative. The rest of the string is a sequence of one or more of the characters `0', `1', `2', `3', `4', `5', `6', `7', `8', and `9'. This method returns "0" if its argument is 0. Otherwise, the string returned by this method does not begin with "0" or "-0".

public static String toString(int i, int radix)

Parameters

i

The int value to be converted to a string.

radix

The radix used in converting the value to a string. This value must be in the range Character.MIN_RADIX to Character.MAX_RADIX.

Returns

The string representation of the given value in the specified radix.

Description

This method returns a String object that contains the representation of the given value in the specified radix.

This method returns a string that begins with `-' if the given value is negative. The rest of the string is a sequence of one or more characters that represent the magnitude of the given value. The characters that can appear in the sequence are determined by the value of radix. If *N* is the value of radix, the first *N* characters on the following line can appear in the sequence:

```
0123456789abcdefghijklmnopqrstuvwxyz
```

The method does not verify that radix is in the proper range. If radix is less than Character.MIN_RADIX or greater than Character.MAX_RADIX, the value 10 is used instead of the given value.

This method returns "0" if its argument is 0. Otherwise, the string returned by this method does not begin with "0" or "-0".

valueOf

public static Integer valueOf(String s) throws NumberFormatException

Parameters

s

The string to be made into an `Integer` object.

Returns

The `Integer` object constructed from the string.

Throws

`NumberFormatException`

If the `String` does not contain a valid representation of an integer.

Description

Constructs an `Integer` object with the value specified by the given string. The string should consist of one or more digit characters. The digit characters can be preceded by a single `-' character. If the string contains any other characters, the method throws a `NumberFormatException`.

`public static Integer valueOf(String s, int radix) throws NumberFormatException`

Parameters

`s`

The string to be made into an `Integer` object.

`radix`

The radix used in converting the string to a value. This value must be in the range `Character.MIN_RADIX` to `Character.MAX_RADIX`.

Returns

The `Integer` object constructed from the string.

Throws

`NumberFormatException`

If the `String` does not contain a valid representation of an integer or `radix` is not in the appropriate range.

Description

Constructs an `Integer` object with the value specified by the given string in the specified radix. The string should consist of one or more digit characters or characters in the range `A' to `Z' or `a' to `z' that are considered digits in the given radix. The digit characters can be preceded by a single `-' character. If the string contains any other characters, the method throws a `NumberFormatException`.

Instance Methods

byteValue

`public byte byteValue()`

Availability

New as of JDK 1.1

Returns

The value of this object as a `byte`.

Overrides

`Number.byteValue()`

Description

This method returns the value of this object as a `byte`. The high order bits of the value are discarded.

doubleValue

`public double doubleValue()`

Returns

The value of this object as a `double`.

Overrides

`Number.doubleValue()`

Description

This method returns the value of this object as a `double`.

equals

`public boolean equals(Object obj)`

Parameters

`obj`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `obj` is an instance of `Integer` and it contains the same value as the object this method is associated with.

floatValue

`public float floatValue()`

Returns

The value of this object as a `float`.

Overrides

`Number.floatValue()`

Description

This method returns the value of this object as a `float`. Rounding may occur.

hashCode

`public int hashCode()`

Returns

A hashcode based on the `int` value of the object.

Overrides

`Object.hashCode()`

Description

This method returns a hashcode computed from the value of this object.

intValue

`public int intValue()`

Returns

The value of this object as an `int`.

Overrides

`Number.intValue()`

Description

This method returns the value of this object as an `int`.

longValue

`public long longValue()`

Returns

The value of this object as a `long`.

Overrides

`Number.longValue()`

Description

This method returns the value of this object as a `long`.

shortValue

`public short shortValue()`

Availability

New as of JDK 1.1

Returns

The value of this object as a short.

Overrides

Number.shortValue()

Description

This method returns the value of this object as a short. The high order bits of the value are discarded.

toString

public String toString()

Returns

The string representation of the value of this object.

Overrides

Object.toString()

Description

This method returns a String object that contains the decimal representation of the value of this object.

This method returns a string that begins with `-' if the value is negative. The rest of the string is a sequence of one or more of the characters `0', `1', `2', `3', `4', `5', `6', `7', `8', and `9'. This method returns "0" if the value of the object is 0. Otherwise, the string returned by this method does not begin with "0" or "-0".

Inherited Methods

Method	Inherited From	Method	Inherited From
clone()	Object	finalize()	Object
getClass()	Object	notify()	Object
notifyAll()	Object	wait()	Object
wait(long)	Object	wait(long, int)	Object

See Also

Character; Class; Exceptions; Integer literals; Integer types; Long; Number; String; System

PREVIOUS HOME NEXT
Float BOOK INDEX Long

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

PREVIOUS

NEXT

Long

Name

Long

Synopsis

Class Name:

`java.lang.Long`

Superclass:

`java.lang.Number`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The `Long` class provides an object wrapper for a `long` value. This is useful when you need to treat a `long` value as an object. For example, there are a number of utility methods that take a reference to an `Object` as one of their arguments. You cannot specify a `long` value for one of these arguments, but you can provide a reference to a `Long` object that encapsulates the `long` value. Furthermore, as of JDK 1.1, the `Long` class is necessary to support the Reflection API and class literals.

The `Long` class also provides a number of utility methods for converting `long` values to other primitive types and for converting `long` values to strings and vice versa.

Class Summary

```
public final class java.lang.Long extends java.lang.Number {
    // Constants
    public static final long MIN_VALUE;
    public static final long MAX_VALUE;
    public final static Class TYPE;                        // New in 1.1
    // Constructors
    public Long(long value);
    public Long(String s);
    // Class Methods
    public static Long getLong(String nm);
    public static Long getLong(String nm, long val);
    public static Long getLong(String nm, Long val);
    public static long parseLong(String s);
    public static long parseLong(String s, int radix);
    public static String toBinaryString(long i);
    public static String toHexString(long i);
    public static String toOctalString(long i);
    public static String toString(long i);
    public static String toString(long i, int radix);
    public static Long valueOf(String s);
    public static Long valueOf(String s, int radix);
    // Instance Methods
    public byte byteValue();                               // New in 1.1
    public double doubleValue();
    public boolean equals(Object obj);
    public float floatValue();
    public int hashCode();
    public int intValue();
    public long longValue();
    public short shortValue();                             // New in 1.1
    public String toString();
}
```

Constants

MAX_VALUE

`public static final long MAX_VALUE = 0x7fffffffffffffffL`

Description

The largest value that can be represented by a `long`.

MIN_VALUE

`public static final long MIN_VALUE = 0x8000000000000000L`

Description

The smallest value that can be represented by a `long`.

TYPE

`public static final Class TYPE`

Availability

New as of JDK 1.1

Description

The `Class` object that represents the type `long`. It is always true that `Long.TYPE` == `long.class`.

Constructors

Long

`public Long(long value)`

Parameters

`value`

The `long` value to be encapsulated by this object.

Description

Creates a `Long` object with the specified `long` value.

`public Long(String s) throws NumberFormatException`

Parameters

`s`

The string to be made into a `Long` object.

Throws

`NumberFormatException`

If the sequence of characters in the given `String` does not form a valid `long` literal.

Description

Constructs a `Long` object with the value specified by the given string. The string should consist of one or more digit characters. The digit characters can be preceded by a single `-' character. If the string contains any other characters, the constructor throws a `NumberFormatException`.

Class Methods

getLong

`public static Integer getLong(String nm)`

Parameters

`nm`

The name of a system property.

Returns

The value of the system property as a `Long` object or a `Long` object with the value 0 if the named property does not exist or cannot be parsed.

Description

This method retrieves the value of the named system property and returns it as a `Long` object. The method obtains the value of the system property as a `String` using `System.getProperty()`.

If the value of the property begins with `0x` or `#` and is not followed by a minus sign, the rest of the value is parsed as a hexadecimal integer. If the value begins with `0`, it's parsed as an octal integer; otherwise it's parsed as a decimal integer.

`public static Long getLong(String nm, long val)`

Parameters

`nm`

The name of a system property.

`val`

A default value for the property.

Returns

The value of the system property as a `Long` object or a `Long` object with the value `val` if the named property does not exist or cannot be parsed.

Description

This method retrieves the value of the named system property and returns it as a `Long` object. The method obtains the value of the system property as a `String` using `System.getProperty()`.

If the value of the property begins with `0x` or # and is not followed by a minus sign, the rest of the value is parsed as a hexadecimal integer. If the value begins with `0`, it's parsed as an octal integer; otherwise it's parsed as a decimal integer.

`public static Long getLong(String nm, Long val)`

Parameters

`nm`

The name of a system property.

`val`

A default value for the property.

Returns

The value of the system property as a `Long` object, or the `Long` object `val` if the named property does not exist or cannot be parsed.

Description

This method retrieves the value of the named system property and returns it as a `Long` object. The method obtains the value of the system property as a `String` using `System.getProperty()`.

If the value of the property begins with `0x` or # and is not followed by a minus sign, the rest of the value is parsed as a hexadecimal integer. If the value begins with `0`, it's parsed as an octal integer; otherwise it's parsed as a decimal integer.

parseLong

`public static long parseLong(String s) throws NumberFormatException`

Parameters

`s`

The `String` to be converted to a `long` value.

Returns

The numeric value of the `long` represented by the `String` object.

Throws

`NumberFormatException`

If the `String` does not contain a valid representation of a `long` value.

Description

This method returns the numeric value of the `long` represented by the contents of the given `String` object. The `String` must contain only decimal digits, except that the first character may be a minus sign.

`public static long parseLong(String s, int radix) throws NumberFormatException`

Parameters

`s`

The `String` to be converted to a `long` value.

`radix`

The radix used in interpreting the characters in the `String` as digits. It must be in the range `Character.MIN_RADIX` to `Character.MAX_RADIX`. If `radix` is in the range 2 through 10, only characters for which the `Character.isDigit()` method returns `true` are considered valid digits. If `radix` is in the range 11 through 36, characters in the ranges `A' through `Z' and `a' through `z' may be considered valid digits.

Returns

The numeric value of the `long` represented by the `String` object in the specified radix.

Throws

`NumberFormatException`

If the `String` does not contain a valid representation of a `long` or `radix` is not in the appropriate range.

Description

This method returns the numeric value of the `long` represented by the contents of the given `String` object in the specified radix. The `String` must contain only valid digits of the specified radix, except that the first character may be a minus sign. The digits are parsed in the specified radix to produce the numeric value.

toBinaryString

`public static String toBinaryString(long value)`

Parameters

`value`

The `long` value to be converted to a string.

Returns

A string that contains the binary representation of the given value.

Description

This method returns a `String` object that contains the representation of the given value as an unsigned binary number. To convert the given value to an unsigned quantity, the method simply uses the value as if it were not negative. In other words, if the given value is negative, the method adds 2^64 to it. Otherwise the value is used as it is.

The string returned by this method contains a sequence of one or more `0' and `1' characters. The method returns "0" if its argument is `0`. Otherwise, the string returned by this method begins with `1'.

toHexString

`public static String toHexString(long value)`

Parameters

`value`

The `long` value to be converted to a string.

Returns

A string that contains the hexadecimal representation of the given value.

Description

This method returns a `String` object that contains the representation of the given value as an unsigned hexadecimal number. To convert the given value to an unsigned quantity, the method simply uses the value as if it were not negative. In other words, if the given value is negative, the method adds 2^64 to it. Otherwise the value is used as it is.

The string returned by this method contains a sequence of one or more of the characters \`0', \`1', \`2', \`3', \`4', \`5', \`6', \`7', \`8', \`9', \`a', \`b', \`c', \`d', \`e', and \`f'. The method returns "0" if its argument is `0`. Otherwise, the string returned by this method does not begin with \`0'.

To produce a string that contains upper- instead of lowercase letters, use the `String.toUpperCase()` method.

toOctalString

`public static String toOctalString(long value)`

Parameters

`value`

The `long` value to be converted to a string.

Returns

A string that contains the octal representation of the given value.

Description

This method returns a `String` object that contains a representation of the given value as an unsigned octal number. To convert the given value to an unsigned quantity, the method simply uses the value as if it were not negative. In other words, if the given value is negative, the method adds 2^64 to it. Otherwise the value is used as it is.

The string returned by this method contains a sequence of one or more of the characters \`0', \`1', \`2', \`3', \`4', \`5', \`6', and \`7'. The method returns "0" if its argument is `0`. Otherwise, the string returned by this method does not begin with \`0'.

toString

`public static String toString(long i)`

Parameters

`i`

The `long` value to be converted to a string.

Returns

The string representation of the given value.

Description

This method returns a `String` object that contains the decimal representation of the given value.

This method returns a string that begins with `-' if the given value is negative. The rest of the string is a sequence of one or more of the characters `0', `1', `2', `3', `4', `5', `6', `7', `8', and `9'. This method returns "0" if its argument is `0`. Otherwise, the string returned by this method does not begin with "0" or "`-0`".

`public static String toString(long i, int radix)`

Parameters

`i`

The `long` value to be converted to a string.

`radix`

The radix used in converting the value to a string. This value must be in the range `Character.MIN_RADIX` to `Character.MAX_RADIX`.

Returns

The string representation of the given value in the specified radix.

Description

This method returns a `String` object that contains the representation of the given value in the specified radix.

This method returns a string that begins with`-' if the given value is negative. The rest of the string is a sequence of one or more characters that represent the magnitude of the given value. The characters that can appear in the sequence are determined by the value of `radix`. If *N* is the value of `radix`, the first *N* characters on the following line can appear in the sequence:

`0123456789abcdefghijklmnopqrstuvwxyz`

The method does not verify that `radix` is in the proper range. If `radix` is less than `Character.MIN_RADIX` or greater than `Character.MAX_RADIX`, the value 10 is used instead of the given value.

This method returns "0" if its argument is `0`. Otherwise, the string returned by this method does not begin with "0" or "`-`0".

valueOf

`public static Long valueOf(String s) throws NumberFormatException`

Parameters

`s`

The string to be made into a `Long` object.

Returns

The `Long` object constructed from the string.

Throws

`NumberFormatException`

If the `String` does not contain a valid representation of a `long`.

Description

Constructs a `Long` object with the value specified by the given string. The string should consist of one or more digit characters. The digit characters can be preceded by a single `-` character. If the string contains any other characters, the method throws a `NumberFormatException`.

`public static Long valueOf(String s, int radix) throws NumberFormatException`

Parameters

`s`

The string to be made into a `Long` object.

`radix`

The radix used in converting the string to a value. This value must be in the range `Character.MIN_RADIX` to `Character.MAX_RADIX`.

Returns

The `Long` object constructed from the string.

Throws

`NumberFormatException`

If the `String` does not contain a valid representation of a `long`.

Description

Constructs a `Long` object with the value specified by the given string in the specified radix. The string should consist of one or more digit characters or characters in the range `A' to `Z' or `a' to `z' that are considered digits in the given radix. The digit characters can be preceded by a single `-' character. If the string contains any other characters, the method throws a `NumberFormatException`.

The method does not verify that `radix` is in the proper range. If `radix` is less than `Character.MIN_RADIX` or greater than `Character.MAX_RADIX`, the value 10 is used instead of the given value.

Instance Methods

byteValue

`public byte byteValue()`

Availability

New as of JDK 1.1

Returns

The value of this object as a `byte`.

Overrides

`Number.byteValue()`

Description

This method returns the value of this object as a `byte`. The high order bits of the value are discarded.

doubleValue

`public double doubleValue()`

Returns

The value of this object as a `double`.

Overrides

`Number.doubleValue()`

Description

This method returns the value of this object as a `double`. Rounding may occur.

equals

`public boolean equals(Object obj)`

Parameters

`obj`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `obj` is an instance of `Long` and it contains the same value as the object this method is associated with.

floatValue

`public float floatValue()`

Returns

The value of this object as a `float`.

Overrides

`Number.floatValue()`

Description

This method returns the value of this object as a `float`. Rounding may occur.

hashCode

`public int hashCode()`

Returns

A hashcode based on the `long` value of the object.

Overrides

`Object.hashCode()`

Description

This method returns a hashcode computed from the value of this object. More specifically, the result is the exclusive OR of the two halves of the `long` value represented by the object. If `value` is the value of the object, the method returns a result equivalent to the following expression:

```
(int)(value^(value>>>32))
```

intValue

`public int intValue()`

Returns

The value of this object as an `int`.

Overrides

`Number.intValue()`

Description

This method returns the value of this object as an `int`. The high-order bits of the value are discarded.

longValue

`public long longValue()`

Returns

The value of this object as a `long`.

Overrides

`Number.longValue()`

Description

This method returns the value of this object as a `long`.

shortValue

`public short shortValue()`

Availability

New as of JDK 1.1

Returns

The value of this object as a `short`.

Overrides

`Number.shortValue()`

Description

This method returns the value of this object as a `short`. The high-order bits of the value are discarded.

toString

`public String toString()`

Returns

The string representation of the value of this object.

Overrides

```
Object.toString()
```

Description

This method returns a `String` object that contains the decimal representation of the value of this object.

This method returns a string that begins with `-' if the value is negative. The rest of the string is a sequence of one or more of the characters `0', `1', `2', `3', `4', `5', `6', `7', `8', and `9'. This method returns "0" if the value of the object is 0. Otherwise, the string returned by this method does not begin with "0" or "-0".

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

Character; Class; Exceptions; Integer; Integer literals; Integer types; Number; String; System

PREVIOUS
Integer

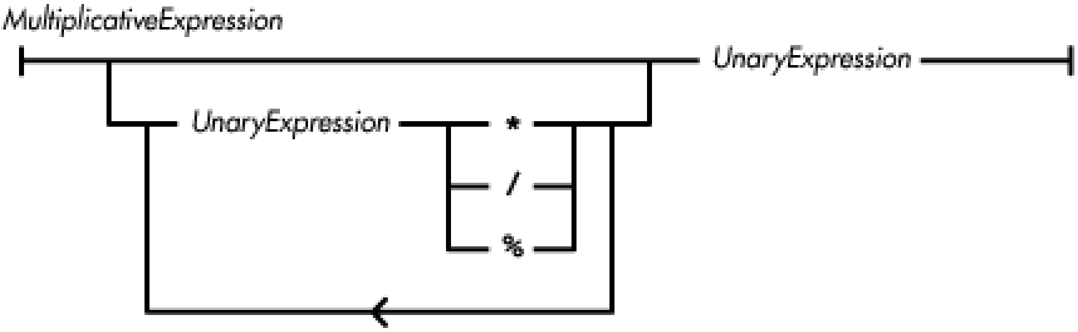
HOME
BOOK INDEX

NEXT
Math

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

4.5 Multiplicative Operators

The multiplicative operators in Java are binary operators that are used for multiplication (*), division (/), and the remainder operation (%). The multiplicative operators appear in multiplicative expressions:



The multiplicative operators are equal in precedence and are evaluated from left to right.

References Unary Operators; Order of Operations

Multiplication Operator *

The binary multiplication operator * produces a pure value that is the product of its operands. The * operator may appear in a multiplicative expression. The multiplication operator never throws an exception.

Here is an example that uses the multiplication operator:

```
int doubleIt(int x) {
    return x*2;
}
```

The types of both operands of the multiplication operator must be arithmetic types, or a compile-time error occurs. The * operator may perform type conversions on its operands:

- If either operand is of type `double`, the other operand is converted to `double` and the operation produces a `double` value.
- Otherwise, if either operand is of type `float`, the other operand is converted to `float` and the operation produces a `float` value.
- Otherwise, if either operand is of type `long`, the other operand is converted to `long` and the operation produces a `long` value.
- Otherwise, both operands are converted to `int` and the operation produces an `int` value.

If the multiplication of integer data overflows, the low order bits of the product are returned; no exception is thrown. The most significant bit of the low order bits is treated as a sign bit. When overflow occurs, the sign of the number produced may not be the same as the sign of the mathematically correct product, due to the limitations of the two's complement representation used for integer data.

The multiplication of floating-point data is governed by the following rules:

- If either operand is not-a-number (NaN), the product is NaN.
- If neither operand is NaN and if both operands have the same sign, the product is positive.
- If neither operand is NaN and if the operands have different signs, the product is negative.
- If one of the operands is positive or negative infinity and the other operand is positive or negative zero, the product is NaN.
- If one of the operands is an infinity value and the other operand is neither zero nor NaN, the product is either positive or negative infinity, as determined by the rules governing the sign of products.
- If neither operand is a zero value, an infinity value, or NaN, the product is rounded to the nearest representable value. If the magnitude of the product is too large to be represented, the operation overflows and an infinity value of the appropriate sign is produced. If the magnitude of the product is too small to be represented, the operation underflows and a zero value of the appropriate sign is produced.

References Arithmetic Types

Division Operator /

The binary division operator `/` produces a pure value that is the quotient of its operands. The left operand is the dividend and the right operand is the divisor. The `/` operator may appear in a multiplicative expression.

Here is an example that uses the division operator:

```
int halfIt(int x) {
    return x/2;
}
```

The types of both operands of the division operator must be arithmetic types, or a compile-time error occurs. The `/` operator may perform type conversions on its operands:

- If either operand is of type `double`, the other operand is converted to `double` and the operation produces a `double` value.

- Otherwise, if either operand is of type `float`, the other operand is converted to `float` and the operation produces a `float` value.

- Otherwise, if either operand is of type `long`, the other operand is converted to `long` and the operation produces a `long` value.

- Otherwise, both operands are converted to `int` and the operation produces an `int` value.

The division of integer data rounds toward zero. If the divisor of an integer division operator is zero, an `ArithmeticException` is thrown. If the dividend is `Integer.MIN_VALUE` or `Long.MIN_VALUE` and the divisor is -1, the quotient produced is `Integer.MIN_VALUE` or `Long.MIN_VALUE`, due to the limitations of the two's complement representation used for integer data.

The division of floating-point data is governed by the following rules:

- If either operand is not-a-number (NaN), the quotient is NaN.

- If neither operand is NaN and if both operands have the same sign, the quotient is positive.

- If neither operand is NaN and if the operands have different signs, the quotient is negative.

- If both of the operands are positive or negative infinity, the quotient is NaN.

- If the dividend is an infinity value and the divisor is a finite number, the quotient is either positive or negative infinity, as determined by the rules governing the sign of quotients.
- If the dividend is a finite number and the divisor is an infinity value, the quotient is either positive or negative zero, as determined by the rules governing the sign of quotients.
- If the divisor is positive or negative zero and the dividend is not zero or NaN, the quotient is either positive or negative infinity, as determined by the rules governing the sign of quotients.
- If both operands are zero values, the quotient is NaN.
- If the dividend is a zero value and the divisor is a non-zero finite number, the quotient is either positive or negative zero, as determined by the rules governing the sign of quotients.
- If the dividend is a non-zero finite number and the divisor is a zero value, the quotient is either positive or negative infinity, as determined by the rules governing the sign of quotients.
- If neither operand is a zero value, an infinity value, or NaN, the quotient is rounded to the nearest representable value. If the magnitude of the quotient is too large to be represented, the operation overflows and an infinity value of the appropriate sign is produced. If the magnitude of the quotient is too small to be represented, the operation underflows and a zero value of the appropriate sign is produced.

References Arithmetic Types; Integer; Long; Runtime exceptions

Remainder Operator %

The binary remainder operator % produces a pure value that is the remainder from an implied division of its operands. The left operand is the dividend and the right operand is the divisor. The % operator may appear in a multiplicative expression.

Here is an example that uses the remainder operator:

```
// format seconds into hours, minutes and seconds
String formatTime(int t) {
    int minutes, seconds;
    seconds = t%60;
    t /= 60;
    minutes = t%60;
    return t/60 + ":" + minutes + ":" + seconds;
}
```

The types of both operands of the remainder operator must be arithmetic types, or a compile-time error occurs. The % operator may perform type conversions on its operands:

- If either operand is of type `double`, the other operand is converted to `double` and the operation produces a `double` value.
- Otherwise, if either operand is of type `float`, the other operand is converted to `float` and the operation produces a `float` value.
- Otherwise, if either operand is of type `long`, the other operand is converted to `long` and the operation produces a `long` value.
- Otherwise, both operands are converted to `int` and the operation produces an `int` value.

When the remainder operation is performed on integer data, the following expression is guaranteed to produce the same value as `a%b`:

```
a-((a/b)*b)
```

The sign of the value produced by the remainder operator is always the sign of the dividend. The magnitude of the value produced by the remainder operator is always less than the absolute value of the divisor. If the divisor is zero, an `ArithmeticException` is thrown.

Unlike C/C++, Java provides a remainder operation for floating-point data. The remainder of floating-point data is computed in a manner similar to the remainder of integer data. The remainder operation uses a truncating division to compute its value. This is unlike the IEEE 754 remainder operation, which uses a rounding division. The IEEE remainder operation is provided by the `Math.IEEEremainder()` method.

The computation of the remainder of `double` and `float` data is governed by the following rules:

- If either operand is not-a-number (NaN), the remainder is NaN.
- If neither operand is NaN, the sign of the remainder is the same as the sign of the dividend.
- If the dividend is positive or negative infinity or the divisor is positive or negative zero, the remainder is NaN.
- If the dividend is a finite number and the divisor is an infinity value, the remainder is equal to the dividend.

- If the dividend is a zero value and the divisor is a finite number, the remainder is equal to the dividend.

- If neither operand is a zero value, an infinity value, or NaN, the remainder is computed according to the following mathematical formula:

  $$p - \left\lfloor \frac{p}{d} \right\rfloor \times d$$

  p is the dividend and d is the divisor. The notation $\lfloor x \rfloor$ means the greatest integer less than or equal to *x* ; this is called the floor operation.

References Arithmetic Types; Math; Runtime exceptions

PREVIOUS | HOME | NEXT
Unary Operators | BOOK INDEX | Additive Operators

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

4.8 Relational Comparison Operators

The relational comparison operators in Java are used for less than (<), less than or equal to (<=), greater than or equal to (>=), greater than (>), and `instanceof` comparison operations. They may appear in a relational expression:

[Graphic: Figure from the text]

The relational comparison operators are equal in precedence and are evaluated from left to right. The <, <=, >=, and > operators are numerical comparison operators, while `instanceof` is a type comparison operator. All of these operators produce `boolean` values.

References Shift Operators; Order of Operations; *Type* 3

Less-Than Operator <

The less-than operator < performs a comparison between its operands and returns a `boolean` value. It returns the pure value `true` if its left operand is less than its right operand; otherwise the operator returns the pure value `false`. The < operator may appear as part of a relational expression. The less-than operator never throws an exception.

The types of both operands of the less-than operator must be arithmetic types, or a compile-time error occurs. The < operator may perform type conversions on its operands:

- If either operand is of type `double`, then the other operand is converted to `double`.
- Otherwise, if either operand is of type `float`, the other operand is converted to `float`.

- Otherwise, if either operand is of type `long`, the other operand is converted to `long`.
- Otherwise, both operands are converted to `int`.

The comparison of any two arithmetic values produces `true` if the value of the left operand is less than the value of the right operand; otherwise the comparison produces `false`. The comparison of floating-point data is governed by the following additional rules:

- If either operand is not-a-number (NaN), the comparison produces `false`.
- Negative infinity is the most negative value. If the left operand is negative infinity, the comparison produces `true`, unless the right operand is also negative infinity, in which case the comparison produces `false`.
- Positive infinity is the most positive value. If the right operand is positive infinity, the comparison produces `true`, unless the left operand is also positive infinity, in which case the comparison produces `false`.
- Positive and negative zero are treated as equal, so `-0.0 < 0.0` produces `false`.

References Arithmetic Types

Less-Than-Or-Equal-To Operator <=

The less-than-or-equal-to operator <= performs a comparison between its operands and returns a `boolean` value. It returns the pure value `true` if its left operand is less than or equal to its right operand; otherwise the operator returns the pure value `false`. The <= operator may appear as part of a relational expression. The less-than-or-equal-to operator never throws an exception.

The types of both operands of the less-than-or-equal-to operator must be arithmetic types, or a compile-time error occurs. The <= operator may perform type conversions on its operands:

- If either operand is of type `double`, then the other operand is converted to `double`.
- Otherwise, if either operand is of type `float`, the other operand is converted to `float`.
- Otherwise, if either operand is of type `long`, the other operand is converted to `long`.
- Otherwise, both operands are converted to `int`.

The comparison of any two arithmetic values produces `true` if the value of the left operand is less than or equal to the value of the right operand; otherwise the comparison produces `false`. The comparison of floating-point data is governed by the following additional rules:

- If either operand is not-a-number (NaN), the comparison produces `false`.
- Negative infinity is the most negative value. If the left operand is negative infinity, the comparison always produces `true`.
- Positive infinity is the most positive value. If the right operand is positive infinity, the comparison always produces `true`.
- Positive and negative zero are treated as equal, so `0.0 <= -0.0` produces `true`.

References Arithmetic Types

Greater-Than-Or-Equal-To Operator >=

The greater-than-or-equal-to operator `>=` performs a comparison between its operands and returns a `boolean` value. It returns the pure value `true` if its left operand is greater than or equal to its right operand; otherwise the operator returns the pure value `false`. The `>=` operator may appear as part of a relational expression. The greater-than-or-equal-to operator never throws an exception.

The types of both operands of the greater-than-or-equal-to operator must be arithmetic types, or a compile-time error occurs. The `>=` operator may perform type conversions on its operands:

- If either operand is of type `double`, then the other operand is converted to `double`.
- Otherwise, if either operand is of type `float`, the other operand is converted to `float`.
- Otherwise, if either operand is of type `long`, the other operand is converted to `long`.
- Otherwise, both operands are converted to `int`.

The comparison of any two arithmetic values produces `true` if the value of the left operand is greater than or equal to the value of the right operand; otherwise the comparison produces `false`. The comparison of floating-point data is governed by the following additional rules:

- If either operand is not-a-number (NaN), the comparison produces `false`.
- Negative infinity is the most negative value. If the right operand is negative infinity, the

comparison always produces `true`.

- Positive infinity is the most positive value. If the left operand is positive infinity, the comparison always produces `true`.

- Positive and negative zero are treated as equal, so `-0.0 >= 0.0` produces `true`.

References Arithmetic Types

Greater-Than Operator >

The greater-than operator `>` performs a comparison between its operands and returns a `boolean` value. It returns the pure value `true` if its left operand is greater than its right operand; otherwise the operator returns the pure value `false`. The `>` operator may appear as part of a relational expression. The greater-than operator never throws an exception.

The types of both operands of the greater-than operator must be arithmetic types, or a compile-time error occurs. The `>` operator may perform type conversions on its operands:

- If either operand is of type `double`, then the other operand is converted to `double`.

- Otherwise, if either operand is of type `float`, the other operand is converted to `float`.

- Otherwise, if either operand is of type `long`, the other operand is converted to `long`.

- Otherwise, both operands are converted to `int`.

The comparison of any two arithmetic values produces `true` if the value of the left operand is greater than the value of the right operand; otherwise the comparison produces `false`. The comparison of floating-point data is governed by the following additional rules:

- If either operand is not-a-number (NaN), the comparison produces `false`.

- Negative infinity is the most negative value. If the right operand is negative infinity, the comparison produces `true`, unless the left operand is also negative infinity, in which case the comparison produces `false`.

- Positive infinity is the most positive value. If the left operand is positive infinity, the comparison produces `true`, unless the right operand is also positive infinity, in which case the comparison produces `false`.

- Positive and negative zero are treated as equal, so `0.0 > -0.0` produces `false`.

References Arithmetic Types

The instanceof Operator

The `instanceof` operator performs a type comparison between its operands and returns a `boolean` value. It returns the pure value `true` if the object referred to by the left operand can be cast to the type specified as the right operand; otherwise the operator returns the pure value `false`. If the value of the left operand is `null`, the `instanceof` operator returns the pure value `false`. The `instanceof` operator may appear as part of a relational expression. The `instanceof` operator never throws an exception.

The type of the left operand of the `instanceof` operator must be a reference type, or a compile-time error occurs.

All objects inherit a method called `equals()` from the `Object` class. The `equals()` method defined in the `Object` class returns `true` if the two objects being compared are the same object. For some classes, it is more appropriate to override the `equals()` method so that it compares the contents of two objects. Before such a method can do the comparison, it should verify that the objects are instances of the same class by using `instanceof`. For example, let's suppose that you are defining a class to represent complex numbers. Since you want the `equals()` method to compare the contents of complex number objects, you define an `equals` method for the complex number class that looks like this:

```
boolean equals (Object o) {
   if (o instanceof complexNumber)
       return o.real == this.real
           && o.imaginary == this.imaginary;
}
```

The `instanceof` operator can also be used to find out if an object is an instance of a class that implements an interface. For example:

```
if (o instanceof Runnable)
    (new Thread((Runnable)o).start;
```

References Casts; Class Types; Interface Types

PREVIOUS HOME NEXT

9.4 The Exception Hierarchy

The possible exceptions in a Java program are organized in a hierarchy of exception classes. The `Throwable` class, which is an immediate subclass of `Object`, is at the root of the exception hierarchy. `Throwable` has two immediate subclasses: `Exception` and `Error`. Figure 9.1 shows the standard exception classes defined in the `java.lang` package, while Figure 9.2 shows the standard error classes defined in `java.lang`.

Figure 9.1: Standard Java exception classes

[Graphic: Figure 9-1]

Figure 9.2: Standard Java error classes

[Graphic: Figure 9-2]

Exceptions

All of the subclasses of `Exception` represent exceptional conditions that a normal Java program may want to handle. Many of the standard exceptions are also subclasses of `RuntimeException`. Runtime exceptions represent runtime conditions that can generally occur in any Java method, so a method is not required to declare that it throws any of the runtime exceptions. However, if a method can throw any of the other standard exceptions, it must declare them in its `throws` clause.

A Java program should try to handle all of the standard exception classes, since they represent routine abnormal conditions that should be anticipated and caught to prevent program termination.

Runtime exceptions

The `java.lang` package defines the following standard runtime exception classes:

`ArithmeticException`

This exception is thrown to indicate an exceptional arithmetic condition, such as integer division by zero.

`ArrayIndexOutOfBoundsException`

This exception is thrown when an out-of-range index is detected by an array object. An out-of-range index occurs when the index is less than zero or greater than or equal to the size of the array.

`ArrayStoreException`

This exception is thrown when there is an attempt to store a value in an array element that is incompatible with the type of the array.

`ClassCastException`

This exception is thrown when there is an attempt to cast a reference to an object to an inappropriate type.

`IllegalArgumentException`

This exception is thrown to indicate that an illegal argument has been passed to a method.

`IllegalMonitorStateException`

This exception is thrown when an object's `wait()`, `notify()`, or `notifyAll()` method is called from a thread that does not own the object's monitor.

`IllegalStateException`

This exception is thrown to indicate that a method has been invoked when the run-time environment is in an inappropriate state for the requested operation. This exception is new in Java 1.1.

`IllegalThreadStateException`

This exception is thrown to indicate an attempt to perform an operation on a thread that is not legal for the thread's current state, such as attempting to resume a dead thread.

`IndexOutOfBoundsException`

The appropriate subclass of this exception (i.e., `ArrayIndexOutOfBoundsException` or

`StringIndexOutOfBoundsException`) is thrown when an array or string index is out of bounds.

`NegativeArraySizeException`

This exception is thrown in response to an attempt to create an array with a negative size.

`NullPointerException`

This exception is thrown when there is an attempt to access an object through a `null` object reference. This can occur when there is an attempt to access an instance variable or call a method through a `null` object or when there is an attempt to subscript an array with a `null` object.

`NumberFormatException`

This exception is thrown to indicate that an attempt to parse numeric information in a string has failed.

`RuntimeException`

The appropriate subclass of this exception is thrown in response to a runtime error detected at the virtual machine level. Because these exceptions are so common, methods that can throw objects that are instances of `RuntimeException` or one of its subclasses are not required to declare that fact in their `throws` clauses.

`SecurityException`

This exception is thrown in response to an attempt to perform an operation that violates the security policy implemented by the installed `SecurityManager` object.

`StringIndexOutOfBoundsException`

This exception is thrown when a `String` or `StringBuffer` object detects an out-of-range index. An out-of-range index occurs when the index is less than zero or greater than or equal to the length of the string.

Other exceptions

The `java.lang` package defines the following standard exception classes that are not runtime exceptions:

`ClassNotFoundException`

This exception is thrown to indicate that a class that is to be loaded cannot be found.

`CloneNotSupportedException`

This exception is thrown when the `clone()` method has been called for an object that does not implement the `Cloneable` interface and thus cannot be cloned.

`Exception`

The appropriate subclass of this exception is thrown in response to an error detected at the virtual machine level. If a program defines its own exception classes, they should be subclasses of the `Exception` class.

`IllegalAccessException`

This exception is thrown when a program tries to dynamically load a class (i.e., uses the `forName()` method of the `Class` class, or the `findSystemClass()` or the `loadClass()` method of the `ClassLoader` class) and the currently executing method does not have access to the specified class because it is in another package and not `public`. This exception is also thrown when a program tries to create an instance of a class (i.e., uses the `newInstance()` method of the `Class` class) that does not have a zero-argument constructor accessible to the caller.

`InstantiationException`

This exception is thrown in response to an attempt to instantiate an `abstract` class or an interface using the `newInstance()` method of the `Class` class.

`InterruptedException`

This exception is thrown to signal that a thread that is sleeping, waiting, or otherwise paused has been interrupted by another thread.

`NoSuchFieldException`

This exception is thrown when a specified variable cannot be found. This exception is new in Java 1.1.

`NoSuchMethodException`

This exception is thrown when a specified method cannot be found.

Errors

The subclasses of `Error` represent errors that are normally thrown by the class loader, the virtual machine, or other support code. Application-specific code should not normally throw any of these standard error classes. If a method does throw an `Error` class or any of its subclasses, the method is not required to declare that fact in its `throws` clause.

A Java program should not try to handle the standard error classes. Most of these error classes represent non-recoverable errors and as such, they cause the Java runtime system to print an error message and terminate program execution.

The `java.lang` package defines the following standard error classes:

`AbstractMethodError`

> This error is thrown in response to an attempt to invoke an `abstract` method.

`ClassCircularityError`

> This error is thrown when a circular reference among classes is detected during class initialization.

`ClassFormatError`

> This error is thrown when an error is detected in the format of a file that contains a class definition.

`Error`

> The appropriate subclass of this error is thrown when an unpredictable error, such as running out of memory, occurs. Because of the unpredictable nature of these errors, methods that can throw objects that are instances of `Error` or one of its subclasses are not required to declare that fact in their `throws` clauses.

`ExceptionInInitializerError`

> This error is thrown when an unexpected exception is thrown in a static initializer. This error is new in Java 1.1.

`IllegalAccessError`

This error is thrown when a class attempts to access a field or call a method it does not have access to. Usually this error is caught by the compiler; this error can occur at run-time if the definition of a class changes after the class that references it was last compiled.

`IncompatibleClassChangeError`

This error or one of its subclasses is thrown when a class refers to another class in an incompatible way. This situation occurs when the current definition of the referenced class is incompatible with the definition of the class that was found when the referring class was compiled. For example, say class `A` refers to a method in class `B`. Then, after class `A` is compiled, the method is removed from class `B`. When class `A` is loaded, the run-time system discovers that the method in class `B` no longer exists and throws an error.

`InstantiationError`

This error is thrown in response to an attempt to instantiate an `abstract` class or an interface. Usually this error is caught by the compiler; this error can occur at run-time if the definition of a class is changed after the class that references it was last compiled.

`InternalError`

This error is thrown to signal an internal error within the virtual machine.

`LinkageError`

The appropriate subclass of this error is thrown when there is a problem resolving a reference to a class. Reasons for this may include a difficulty in finding the definition of the class or an incompatibility between the current definition and the expected definition of the class.

`NoClassDefFoundError`

This error is thrown when the definition of a class cannot be found.

`NoSuchFieldError`

This error is thrown in response to an attempt to reference an instance or class variable that is not defined in the current definition of a class. Usually this error is caught by the compiler; this error can occur at run-time if the definition of a class is changed after the class that references it was last compiled.

`NoSuchMethodError`

> This error is thrown in response to an attempt to reference a method that is not defined in the current definition of a class. Usually this error is caught by the compiler; this error can occur at run-time if the definition of a class is changed after the class that references it was last compiled.

`OutOfMemoryError`

> This error is thrown when an attempt to allocate memory fails.

`StackOverflowError`

> This error is thrown when a stack overflow error occurs within the virtual machine.

`ThreadDeath`

> This error is thrown by the `stop()` method of a `Thread` object to kill the thread. Catching `ThreadDeath` objects is not recommended. If it is necessary to catch a `ThreadDeath` object, it is important to re-throw the object so that it is possible to cleanly stop the catching thread.

`UnknownError`

> This error is thrown when an error of unknown origins is detected in the run-time system.

`UnsatisfiedLinkError`

> This error is thrown when the implementation of a native method cannot be found.

`VerifyError`

> This error is thrown when the byte-code verifier detects that a class file, though well-formed, contains some sort of internal inconsistency or security problem.

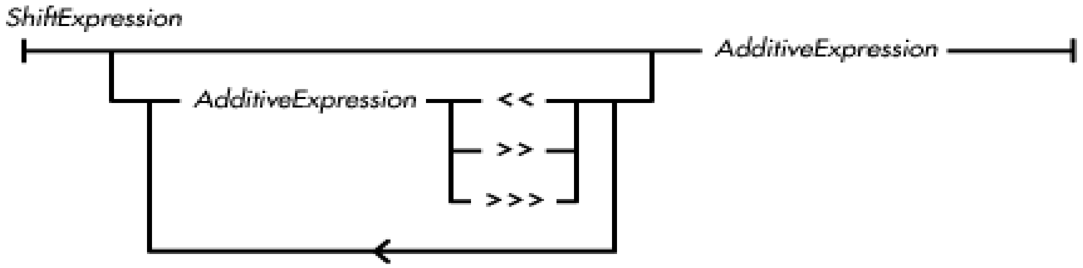
`VirtualMachineError`

> The appropriate subclass of this error is thrown to indicate that the Java virtual machine has encountered an error.

PREVIOUS HOME NEXT

4.7 Shift Operators

The shift operators in Java are used for left shift (<<), right shift (>>), and unsigned right shift (>>>) operations. The shift operators may appear in a shift expression:



The shift operators are equal in precedence and are evaluated from left to right.

References Additive Operators; Order of Operations

Left Shift Operator <<

The left shift operator << produces a pure value that is its left operand left-shifted by the number of bits specified by its right operand. The << operator may appear in a shift expression. The left shift operator never throws an exception.

Here are some examples of the left shift operator:

```
(3<<2) == 12
(-3<<2) == -12
(0x01234567<<4) == 0x12345670
(0xF1234567<<4) == 0x12345670
```

The type of each operand of the left shift operator must be an integer data type, or a compile-time error occurs. The << operator may perform type conversions on its operands; unlike arithmetic binary operators, each operand is converted independently. If the type of an operand is `byte`, `short`, or `char`, that operand is converted to an `int` before the value of the operator is computed. The type of the value produced by the left shift operator is the type of its left operand.

If the converted type of the left operand is `int`, only the five least significant bits of the value of the right operand are used as the shift distance. Therefore, the shift distance is in the range 0 through 31. In this case, the value produced by `r << s` is mathematically equivalent to:

$$r \times 2^{s \bmod 32}$$

If the type of the left operand is `long`, only the six least significant bits of the value of the right operand are used as the shift distance. Therefore, the shift distance is in the range 0 through 63. In this case, the value produced by `r << s` is mathematically equivalent to:

$$r \times 2^{s \bmod 64}$$

References Integer types

Right Shift Operator >>

The right shift operator `>>` produces a pure value that is its left operand right-shifted with sign extension by the number of bits specified by its right operand. Right-shifting with sign extension means that shifting a value `n` places to the right causes the `n` high order bits to contain the same value as the sign bit of the unshifted value. The >> operator may appear as part of a shift expression. The right shift operator never throws an exception. Here are some examples of the right shift operator:

```
(0x01234567>>4) == 0x00123456
(0xF1234567>>4) == 0xFF123456
```

The type of each operand of the right shift operator must be an integer data type, or a compile-time error occurs. The `>>` operator may perform type conversions on its operands; unlike arithmetic binary operators, each operand is converted independently. If the type of an operand is `byte`, `short`, or `char`, that operand is converted to an `int` before the value of the operator is computed. The type of the value produced by the right shift operator is the type of its left operand.

If the converted type of the left operand is `int`, only the five least significant bits of the value of the right operand are used as the shift distance. Therefore, the shift distance is in the range 0 through 31.

In this case, the value produced by `r >> s` is mathematically equivalent to:

$$\left\lfloor \frac{r}{2^{s \bmod 32}} \right\rfloor$$

The notation $\lfloor x \rfloor$ means the greatest integer less than or equal to *x* ; this is called the floor operation.

If the type of the left operand is `long`, only the six least significant bits of the value of the right operand are used as the shift distance. Therefore, the shift distance is in the range 0 through 63. In this case, the value produced by `r >> s` is mathematically equivalent to:

$$\left\lfloor \frac{r}{2^{s \bmod 64}} \right\rfloor$$

References Integer types

Unsigned Right Shift Operator >>>

The unsigned right shift operator `>>>` produces a pure value that is its left operand right-shifted with zero extension by the number of bits specified by its right operand. Right-shifting with zero extension means that shifting a value `n` places to the right causes the `n` high order bits to contain zero. The `>>>` operator may appear as part of a shift expression. The unsigned right shift operator never throws an exception.

Here are some examples of the unsigned right shift operator:

```
(0x01234567>>>4) == 0x00123456
(0xF1234567>>>4) == 0x0F123456
```

The type of each operand of the unsigned right shift operator must be an integer data type, or a compile-time error occurs. The `>>>` operator may perform type conversions on its operands; unlike arithmetic binary operators, each operand is converted independently. If the type of an operand is `byte`, `short`, or `char`, that operand is converted to an `int` before the value of the operator is computed. The type of the value produced by the unsigned right shift operator is the type of its left operand. If the converted type of the left operand is `int`, only the five least significant bits of the value of the right operand are used as the shift distance. So, the shift distance is in the range 0 through 31. Here, the value produced by `r >>> s` is the same as:

```
s==0 ? r : (r >> s) & ~(-1<<(32-s))
```

If the type of the left operand is `long`, then only the six least significant bits of the value of the right operand are used as the shift distance. So, the shift distance is in the range 0 through 63. Here, the value produced by `r >>> s` is the same as the following:

```
s==0 ? r : (r >> s) & ~(-1<<(64-s))
```

References Integer types

Short

Name

Short

Synopsis

Class Name:

`java.lang.Short`

Superclass:

`java.lang.Number`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

New as of JDK 1.1

Description

The `Short` class provides an object wrapper for a `short` value. This is useful when you need to treat a `short` value as an object. For example, there are a number of utility methods that take a reference to an `Object` as one of their arguments. You cannot specify a `short` value for one of these arguments, but you can provide a reference to a `Short` object that encapsulates the `byte` value. Furthermore, the `Short` class is necessary as of JDK 1.1 to support the

Reflection API and class literals.

The `Short` class also provides a number of utility methods for converting `short` values to other primitive types and for converting `short` values to strings and vice-versa.

Class Summary

```
public final class java.lang.Short extends java.lang.Number {
  // Constants
  public static final short MAX_VALUE;
  public static final short MIN_VALUE;
  public static final Class TYPE;
  // Constructors
  public Short(short value);
  public Short(String s);
  // Class Methods
  public static Short decode(String nm);
  public static short parseShort(String s);
  public static short parseShort(String s, int radix);
  public static String toString(short s);
  public static Short valueOf(String s, int radix);
  public static Short valueOf(String s);
  // Instance Methods
  public byte byteValue();
  public double doubleValue();
  public boolean equals(Object obj);
  public float floatValue();
  public int hashCode();
  public int intValue();
  public long longValue();
  public short shortValue();
  public String toString();
}
```

Constants

MAX_VALUE

public static final short MAX_VALUE= 32767

The largest value that can be represented by a `short`.

MIN_VALUE

public static final byte MIN_VALUE= -32768

The smallest value that can be represented by a `short`.

TYPE

public static final Class TYPE

The `Class` object that represents the primitive type `short`. It is always true that `Short.TYPE` == `short.class`.

Constructors

Short

public Short(short value)

Parameters

`value`

The `short` value to be encapsulated by this object.

Description

Creates a `Short` object with the specified `short` value.

public Short(String s) throws NumberFormatException

Parameters

`s`

The string to be made into a `Short` object.

Throws

`NumberFormatException`

If the sequence of characters in the given `String` does not form a valid `short` literal.

Description

Constructs a `Short` object with the value specified by the given string. The string should consist of one or more digit characters. The digit characters can be preceded by a single `` `-' `` character. If the string contains any other characters, the constructor throws a `NumberFormatException`.

Class Methods

decode

public static Short decode(String nm) throws NumberFormatException

Parameters

`nm`

A `String` representation of the value to be encapsulated by a `Short` object. If the string begins with # or `0x`, it is a radix 16 representation of the value. If the string begins with `0`, it is a radix 8 representation of the value. Otherwise, it is assumed to be a radix 10 representation of the value.

Returns

A `Short` object that encapsulates the given value.

Throws

`NumberFormatException`

If the `String` contains any non-digit characters other than a leading minus sign or the value represented by the `String` is less than `Short.MIN_VALUE` or greater than `Short.MAX_VALUE`.

Description

This method returns a `Short` object that encapsulates the given value.

parseByte

```
public static short parseShort(String s) throws NumberFormatException
```

Parameters

`s`

The `String` to be converted to a `short` value.

Returns

The numeric value of the `short` represented by the `String` object.

Throws

`NumberFormatException`

If the `String` does not contain a valid representation of a `short` or the value represented by the `String` is less than `Short.MIN_VALUE` or greater than `Short.MAX_VALUE`.

Description

This method returns the numeric value of the `short` represented by the contents of the given `String` object. The `String` must contain only decimal digits, except that the first character may be a minus sign.

`public static short parseShort(String s, int radix) throws NumberFormatException`

Parameters

`s`

The `String` to be converted to a short value.

`radix`

The radix used in interpreting the characters in the `String` as digits. This value must be in the range `Character.MIN_RADIX` to `Character.MAX_RADIX`. If radix is in the range 2 through 10, only characters for which the `Character.isDigit()` method returns `true` are considered to be valid digits. If `radix` is in the range 11 through 36, characters in the ranges `A' through `Z' and `a' through `z' are considered valid digits.

Returns

The numeric value of the `short` represented by the `String` object in the specified radix.

Throws

`NumberFormatException`

If the `String` does not contain a valid representation of a `short`, radix is not in the appropriate range, or the value represented by the `String` is less than `Short.MIN_VALUE` or greater than `Short.MAX_VALUE`.

Description

This method returns the numeric value of the `short` represented by the contents of the given `String` object in the specified radix. The `String` must contain only valid digits of the specified radix, except that the first character may be a minus sign. The digits are parsed in the specified radix to produce the numeric value.

toString

public String toString(short s)

Parameters

`s`

The `short` value to be converted to a string.

Returns

The `string` representation of the given value.

Description

This method returns a `String` object that contains the decimal representation of the given value.

This method returns a string that begins with `-' if the given value is negative. The rest of the string is a sequence of one or more of the characters `0', `1', `2', `3', `4', `5', `6', `7', `8', and `9'. This method returns "0" if its argument is `0`. Otherwise, the string returned by this method does not begin with "0" or "-0".

valueOf

public static Short valueOf(String s) throws NumberFormatException

Parameters

`s`

The string to be made into a `Short` object.

Returns

The `Short` object constructed from the string.

Throws

`NumberFormatException`

If the `String` does not contain a valid representation of a `short` or the value represented by the `String` is less than `Short.MIN_VALUE` or greater than `Short.MAX_VALUE`.

Description

Constructs a `Short` object with the value specified by the given string. The string should consist of one or more digit characters. The digit characters can be preceded by a single `-'. If the string contains any other characters, the method throws a `NumberFormatException`.

`public static Short valueOf(String s, int radix) throws NumberFormatException`

Parameters

`s`

The string to be made into a `Short` object.

`radix`

The radix used in converting the string to a value. This value must be in the range `Character.MIN_RADIX` to `Character.MAX_RADIX`.

Returns

The `Short` object constructed from the string.

Throws

`NumberFormatException`

If the `String` does not contain a valid representation of a `short`, `radix` is not in the appropriate range, or the value represented by the `String` is less than `Short.MIN_VALUE` or greater than `Short.MAX_VALUE`.

Description

Constructs a `Short` object with the value specified by the given string in the specified radix. The string should consist of one or more digit characters or characters in the range `A' to `Z' or `a' to `z' that are considered digits in the given radix. The digit characters can be preceded by a single `-' character. If the string contains any other characters, the method throws a `NumberFormatException`.

Instance Methods

byteValue

public byte byteValue()

Returns

The value of this object as a `byte`. The high order bits of the value are discarded.

Overrides

`Number.byteValue()`

Description

This method returns the value of this object as a `byte`.

doubleValue

public double doubleValue()

Returns

The value of this object as a `double`.

Overrides

`Number.doubleValue()`

Description

This method returns the value of this object as a `double`.

equals

public boolean equals(Object obj)

Parameters

`obj`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `obj` is an instance of `Short` and it contains the same value as the object this method is associated with.

floatValue

public float floatValue()

Returns

The value of this object as a `float`.

Overrides

`Number.floatValue()`

Description

This method returns the value of this object as a `float`.

hashCode

public int hashCode()

Returns

A hashcode based on the `short` value of the object.

Overrides

`Object.hashCode()`

Description

This method returns a hash code computed from the value of this object.

intValue

public int intValue()

Returns

The value of this object as an `int`.

Overrides

`Number.intValue()`

Description

This method returns the value of this object as an `int`.

longValue

public long longValue()

Returns

The value of this object as a `long`.

Overrides

`Number.longValue()`

Description

This method returns the value of this object as a `long`.

shortValue

public short shortValue()

Returns

The value of this object as a `short`.

Overrides

`Number.shortValue()`

Description

This method returns the value of this object as a `short`.

toString

public String toString()

Returns

The string representation of the value of this object.

Overrides

`Object.toString()`

Description

This method returns a `String` object that contains the decimal representation of the value of this object.

This method returns a string that begins with `-' if the value is negative. The rest of the string is a sequence of one or more of the characters `0', `1', `2', `3', `4', `5', `6', `7', `8', and `9'. This method returns "0" if the value of the object is `0`. Otherwise, the string returned by this method does not begin with "0" or "-0".

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

Byte; Character; Class; Double; Exceptions; Float; Integer literals; Integer types; Integer; Long; Number; String

PREVIOUS
SecurityManager

HOME
BOOK INDEX

NEXT
String

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

4.4 Unary Operators

Unary operators are operators that take exactly one argument. Unary operators may appear in a unary expression:

[Graphic: Figure from the text]

The unary plus and minus operators, a Boolean negation operator (`!`), a bitwise negation operator (~), and the cast construct comprise the unary operators in Java. The unary operators are equal in precedence and are evaluated from right to left.

References Order of Operations; Postfix Increment/Decrement Operators; Prefix Increment/Decrement Operators; Primary Expressions; *Type* 3

Unary Plus Operator +

The unary plus operator (`+`) can appear as part of a unary expression. The operator does no explicit computation; it produces the same pure value that is produced by its operand. However, the unary + operator may perform a type conversion on its operand. The type of the operand must be an arithmetic data type, or a compile-time error occurs. If the type of the operand is `byte`, `short`, or `char,` the unary `+` operator produces an `int` value; otherwise the operator produces a value of the same type as its operand.

References Arithmetic Types

Unary Minus Operator -

The unary minus operator (-) can appear as part of a unary expression. The type of the operand of the unary - operator must be an arithmetic data type, or a compile-time error occurs. The operator produces a pure value that is the arithmetic negation (i.e., additive inverse) of the value of its operand.

The unary - operator may perform a type conversion. If the type of the operand is `byte`, `short`, or `char`, the operation converts the operand to an `int` before computing the value's arithmetic negation and producing an `int` value. Otherwise, unary - produces a value of the same type as its operand.

For integer data types, the unary - operator produces a value equivalent to subtracting its operand from zero. There are, however, negative values for which the unary - operator cannot produce a positive value; in these cases it produces the same negative value as its operand. This behavior results from the two's complement representation Java uses for integer values. The magnitude of the most negative number that can be represented using two's complement notation cannot be represented as a positive number. No exception is thrown when the unary - operator is given a value that cannot be negated. However, you can detect this situation by explicitly testing for these special values. The most negative `int` value is available as the predefined constant `Integer.MIN_VALUE` and the most negative `long` value is available as the predefined constant `Long.MIN_VALUE`.

For floating-point data types, the unary - operator changes the sign of its operand from + to - or from - to +, for both regular values, positive and negative zero, and positive and negative infinity. The only case where this is not true occurs when the operand is not-a-number (NaN). Given the value NaN, the unary - operator produces NaN.

References Arithmetic Types; Integer; Long

Boolean Negation Operator !

The Boolean negation operator (!) may appear as part of a unary expression. The type of the operand of the ! operator must be `boolean`, or a compile-time error occurs. If the value of its operand is `false`, the ! operator produces the pure `boolean` value `true`. If the value of its operand is `true`, the operator produces the pure `boolean` value `false`.

Here is an example that uses the Boolean negation operator:

```
public void paint(Graphics g) {
    if (!loaded) {
        //The next 2 lines are executed if loaded is false
        g.drawString("Loading data", 25, 25);
        return;
    }
    g.drawImage(img, 25, 25, this);
}
```

References Boolean Type

Bitwise Negation Operator ~

The bitwise negation operator (~) may appear as part of a unary expression. The type of the operand of the ~ operator must be an integer data type, or a compile-time error occurs. The ~ operator may perform a type conversion before it performs its computation. If the type of the operand is `byte`, `short`, or `char`, the operator converts its operand to `int` before producing a value. Otherwise the ~ operator produces a value of the same type as its operand.

After type conversion, the bitwise negation operator produces a pure value that is the bitwise complement of its operand. In other words, if a bit in the converted operand contains a one, the corresponding bit in the result contains a zero. If a bit in the converted operand contains a zero, the corresponding bit in the result contains a one.

Here's an example that shows the use of the bitwise negation operator:

```
// zero low order four bits
int getNibble(int x) {
    return x & ~0xf;
}
```

References Integer types

Casts

A *Type* enclosed in parentheses specifies a type cast operation. A cast may appear as part of a unary expression. A cast operation always produces a pure value of the specified type, by converting its operand to that type if necessary. This is different from a type cast in C/C++, which can produce garbage if it is given a pointer to a data type different than that implied by the pointer's declaration. If the actual data type of the operand of a cast cannot be guaranteed at compile-time, the Java compiler must produce code to check the type of the operand at runtime. In Java, any value that gets past all of the type-checking done on a cast is guaranteed to be compatible with the type specified by the cast.

A cast can convert between certain primitive types. A cast between object reference types never alters the type or content of the object, but may alter the type of the reference to the object.

Because it is not possible to convert between all types, some cast operations are permitted and others are not. Here are the rules governing casts:

- A value of any data type can be cast to its own type.
- A value of any arithmetic data type can be cast to any other arithmetic data type. Casting a floating-point value to an integer data type rounds toward zero.
- A value of the `boolean` data type cannot be cast to any other data type, nor can a value of any other data type be cast to `boolean`.
- A value of any primitive data type cannot be cast to a reference data type, nor can a reference be cast to any primitive data type.
- A reference to a class type can be cast to the type of the superclass of that class.
- A reference to a class type can be cast to the type of a subclass of that class if the reference actually refers to an object of the specified class or any of its subclasses. Unless the Java compiler can prove that the object actually referenced is of the specified class or any of its subclasses, the compiler must generate a runtime test to verify that the object is of an appropriate type. At runtime, if the object actually referenced is not of an appropriate type, a `ClassCastException` is thrown. Consider the following example:

  ```
  Object o = "ABC";
  String s = (String)o;   // This is okay
  Double d = (Double)o;   // Throws an exception
  ```

  The cast of `o` to `String` is fine because `o` is really a reference to a `String` object. The cast of `o` to `Double` throws an exception at runtime because the object that `o` references is not an instance of `Double`.

- A reference to a class type can be cast to an interface type if the reference actually refers to an object of a class that implements the specified interface. If the class of the reference being cast is a `final` class, the compiler can determine if the reference actually refers to an object of a class that implements the specified interface, because a `final` class cannot have any subclasses. Otherwise, the compiler must generate a runtime test to determine if the reference actually refers to an object of a class that implements the specified interface. At runtime, if the object actually referenced is not of a class that implements the interface, a `ClassCastException` is thrown. Here is an example that illustrates the rules governing casts to interface types:

  ```
  interface Weber { double flux(double x); }
  class B {}
  ```

```
final class C {}
class D implements Weber {
    public double flux(double x) {
        return Math.PI*x*x;
    }
}
class Intercast {
    public void main(String[] argv) {
        B b = new B();
        C c = new C();
        D d = new D();
        Weber w;
        w = (Weber)b;   // Throws an exception
        w = (Weber)c;   // Compiler complains
        w = (Weber)d;   // Okay, D implements Weber
    }
}
```

The cast of `b` to `Weber` is fine with the compiler because the class `B` might have a subclass that implements `Weber`. At runtime, however, this cast throws an exception because `B` does not implement `Weber`. The cast of `c` to `Weber` produces an error message from the compiler, as the `C` class does not implement `Weber`. Because `C` is `final`, it will not have any subclasses and therefore there is no possibility of `c` containing a reference to an object that implements the `Weber` interface. The cast of `d` to `Weber` is fine because the `D` class implements the `Weber` interface.

- A reference to the class `Object` can be cast to an array type if the reference actually refers to an array object of the specified type. The compiler generates a runtime test to determine if the reference actually refers to the specified type of array object. At runtime, if the object actually referenced is not the specified type of array, a `ClassCastException` is thrown.

- A reference to an interface type can be cast to a class type if the reference actually refers to an instance of the specified class or any of its subclasses. If the specified class is a `final` class that does not implement the referenced interface, the compiler can reject the cast because a `final` class cannot have any subclasses. Otherwise, the compiler generates a runtime test to determine if the reference actually refers to an object of the appropriate type. At runtime, if the object actually referenced is not of the appropriate type, a `ClassCastException` is thrown.

  Here is an example to illustrate these points:

```
interface Weber { double flux(double x); }
class B {}
```

```
final class C {}
class D implements Weber {
    public double flux(double x) {
        return Math.PI*x*x;
    }
}
class Intercast {
    public void doit(Weber w) {
        B b = (B)w;   // May throw an exception
        C c = (C)w;   // Compiler complains
        D d = (D)w;   // Okay
    }
}
```

The cast of `w` to class `B` is fine with the compiler even though `B` does not implement `Weber`. The compiler lets it pass because `B` might have a subclass that implements `Weber` and `w` could contain a reference to that class. However, at runtime, the cast will throw an exception if the object actually referenced is not an instance of `B` or a subclass of `B`. The cast of `w` to class `C` produces an error message from the compiler. `C` does not implement `Weber` and `C` cannot have any subclasses because it is `final`; any object that implements `Weber` cannot be an instance of `C`. The cast of `w` to class `D` is fine at compile-time because `D` implements `Weber`. At runtime, if `w` references an object that is not an instance of `D`, a `ClassCastException` is thrown.

- A reference to an interface type can be cast to another interface type if the reference actually refers to an object of a class that implements the specified interface. If the referenced interface extends the specified interface, the compiler knows that the cast is legal. Otherwise, the compiler generates a runtime test to determine if the reference actually refers to an object that implements the specified interface. At runtime, if the object actually referenced does not implement the specified interface, a `ClassCastException` is thrown.

  Here is an example to illustrate these points:

```
interface Weber { double flux(double x); }
interface Dyn { double squeeze(); }
interface Press extends Dyn {
    double squeeze(double theta);
}
class D implements Press {
    public double squeeze() { return Math.PI; }
    public double squeeze(double theta) {
        return Math.PI*Math.sin(theta);
    }
}
```

```
class Interinter {
    public static void doit(D d) {
        Dyn dyn = d;          // Okay
        Weber w = (Weber)d;   // May throw exception
    }
}
```

The assignment of `d` to `dyn` works because `d` is of class `D`, `D` implements `Press`, and `Press` extends `Dyn`. Therefore, `d` refers to an object that implements `Dyn` and we have assignment compatibility. The compiler lets the cast of `d` to `Weber` pass because there may be a subclass of `D` that implements `Weber`. At runtime, the cast will throw an exception if `D` does not implement `Weber`.

- A reference to an array object can be cast to the class type `Object`.
- A reference to an array object can be cast to another array type if either of the following is true:
    - The elements of the referenced array and the elements of the specified array type are of the same primitive type.
    - The elements of the referenced array are of a type that can be cast to the type of the elements of the specified array type.

Any cast operation not covered by the preceding rules is not allowed and the Java compiler issues an error message.

References Arithmetic Types; Array Types; Boolean Type; Class Types; Interface Types; Runtime exceptions

Double

Name

Double

Synopsis

Class Name:

`java.lang.Double`

Superclass:

`java.lang.Number`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The `Double` class provides an object wrapper for a `double` value. This is useful when you need to treat a `double` value as an object. For example, there are a number of utility methods that take a reference to an `Object` as one of their arguments. You cannot specify a `double` value for one of these arguments, but you can provide a reference to a `Double` object that encapsulates the `double` value. Furthermore, as of JDK 1.1, the `Double` class is necessary to support the Reflection API and class literals.

In Java, `double` values are represented using the IEEE 754 format. The `Double` class provides constants for the three special values that are mandated by this format: `POSITIVE_INFINITY`, `NEGATIVE_INFINITY`, and `NaN` (not-a-number).

The `Double` class also provides some utility methods, such as methods for determining whether a `double` value is an infinity value or NaN, for converting `double` values to other primitive types, and for converting a `double` to a `String` and vice versa.

Class Summary

```
public final class java.lang.Double extends java.lang.Number {
    // Constants
    public final static double MAX_VALUE;
    public final static double MIN_VALUE;
    public final static double NaN;
    public final static double NEGATIVE_INFINITY;
    public final static double POSITIVE_INFINITY;
    public final static Class TYPE;                    // New in 1.1
    // Constructors
    public Double(double value);
    public Double(String s);
    // Class Methods
    public static native long doubleToLongBits(double value);
    public static boolean isInfinite(double v);
    public static boolean isNaN(double v);
    public static native double longBitsToDouble(long bits);
    public static String toString(double d);
    public static Double valueOf(String s);
    // Instance Methods
    public byte byteValue();                           // New in 1.1
    public double doubleValue();
    public boolean equals(Object obj);
    public float floatValue();
    public int hashCode();
    public int intValue();
    public boolean isInfinite();
    public boolean isNaN();
    public long longValue();
    public short shortValue();                          // New in 1.1
    public String toString();
}
```

Constants

MAX_VALUE

`public static final double MAX_VALUE = 1.79769313486231570e+308`

Description

The largest value that can be represented by a `double`.

MIN_VALUE

`public static final double MIN_VALUE = 4.94065645841246544e-324`

Description

The smallest value that can be represented by a `double`.

NaN

`public static final double NaN = 0.0 / 0.0`

Description

This variable represents the value not-a-number (NaN), which is a special value produced by `double` operations such as division of zero by zero. When NaN is one of the operands, most arithmetic operations return NaN as the result.

Most comparison operators (<, <=, ==, >=, >) return `false` when one of their arguments is NaN. The exception is `!=`, which returns `true` when one of its arguments is NaN.

NEGATIVE_INFINITY

`public static final double NEGATIVE_INFINITY = -1.0 / 0.0`

Description

> This variable represents the value negative infinity, which is produced when a `double` operation underflows or a negative `double` value is divided by zero. Negative infinity is by definition less than any other `double` value.

POSITIVE_INFINITY

`public static final double POSITIVE_INFINITY = 1.0 / 0.0`

Description

> This variable represents the value positive infinity, which is produced when a `double` operation overflows or a positive `double` value is divided by zero. Positive infinity is by definition greater than any other `double` value.

TYPE

`public static final Class TYPE`

Availability

> New as of JDK 1.1

Description

> The `Class` object that represents the type `double`. It is always true that `Double.TYPE` == `double.class`.

Constructors

Double

`public Double(double value)`

Parameters

`value`

The `double` value to be encapsulated by this object.

Description

Creates a `Double` object with the specified `double` value.

public Double(String s) throws NumberFormatException

Parameters

`s`

The string to be made into a `Double` object.

Throws

`NumberFormatException`

If the sequence of characters in the given `String` does not form a valid `double` literal.

Description

Constructs a `Double` object with the value specified by the given string. The string must contain a sequence of characters that forms a legal `double` literal.

Class Methods

doubleToLongBits

`public static native long doubleToLongBits(double value)`

Parameters

`value`

The `double` value to be converted.

Returns

The `long` value that contains the same sequence of bits as the representation of the given `double` value.

Description

This method returns the `long` value that contains the same sequence of bits as the representation of the given `double` value. The meaning of the bits in the result is defined by the IEEE 754 floating-point format: bit 63 is the sign bit, bits 62-52 are the exponent, and bits 51-0 are the mantissa.

An argument of `POSITIVE_INFINITY` produces the result `0x7ff0000000000000L`, an argument of `NEGATIVE_INFINITY` produces the result `0xfff0000000000000L`, and an argument of `NaN` produces the result `0x7ff8000000000000L`.

The value returned by this method can be converted back to the original `double` value by the `longBitsToDouble()` method.

isInfinite

`static public boolean isInfinite(double v)`

Parameters

`v`

The `double` value to be tested.

Returns

`true` if the specified value is equal to `POSITIVE_INFINITY` or `NEGATIVE_INFINITY`; otherwise `false`.

Description

This method determines whether or not the specified value is an infinity value.

isNaN

`public static boolean isNaN(double v)`

Parameters

`v`

The `double` value to be tested.

Returns

`true` if the specified value is equal to `NaN`; otherwise `false`.

Description

This method determines whether or not the specified value is NaN.

longBitsToDouble

`public static native double longBitsToDouble(long bits)`

Parameters

`bits`

The `long` value to be converted.

Returns

The `double` value whose representation is the same as the bits in the given `long` value.

Description

This method returns the `double` value whose representation is the same as the bits in the given `double` value. The meaning of the bits in the `long` value is defined by the IEEE 754 floating-point format: bit 63 is the sign bit, bits 62-52 are the exponent, and bits 51-0 are the mantissa. The argument `0x7f80000000000000L` produces the result `POSITIVE_INFINITY` and the argument `0xff80000000000000L` produces the result `NEGATIVE_INFINITY`. Arguments in the ranges `0x7ff0000000000001L` through `0x7fffffffffffffffL` and

`0xfff0000000000001L` through `0xffffffffffffffffL` all produce the result `NaN`.

Except for NaN values not normally used by Java, this method is the inverse of the `doubleToLongBits()` method.

toString

`public static String toString(double d)`

Parameters

`d`

The `double` value to be converted.

Returns

A string representation of the given value.

Description

This method returns a `String` object that contains a representation of the given `double` value.

The values `NaN`, `NEGATIVE_INFINITY`, `POSITIVE_INFINITY`, `-0.0`, and `+0.0` are represented by the strings `"NaN"`, `"--Infinity"`, `"Infinity"`, `"--0.0"`, and `"0.0"`, respectively.

For other values, the exact string representation depends on the value being converted. If the absolute value of `d` is greater than or equal to 10^-3 or less than or equal to 10^7, it is converted to a string with an optional minus sign (if the value is negative) followed by up to eight digits before the decimal point, a decimal point, and the necessary number of digits after the decimal point (but no trailing zero if there is more than one significant digit). There is always a minimum of one digit after the decimal point.

Otherwise, the value is converted to a string with an optional minus sign (if the value is negative), followed by a single digit, a decimal point, the necessary number of digits after the decimal point (but no trailing zero if there is more than one significant digit), and the letter `E` followed by a plus or a minus sign and a base 10 exponent of at least one digit. Again, there is always a minimum of one digit after the decimal point.

Note that the definition of this method has changed as of JDK 1.1. Prior to that release, the method provided a string representation that was equivalent to the `%g` format of the `printf` function in C.

valueOf

`public static Double valueOf(String s) throws NumberFormatException`

Parameters

`s`

The string to be made into a `Double` object.

Returns

The `Double` object constructed from the string.

Throws

`NumberFormatException`

If the sequence of characters in the given `String` does not form a valid `double` literal.

Description

Constructs a `Double` object with the value specified by the given string. The string must contain a sequence of characters that forms a legal `double` literal. This method ignores leading and trailing white space in the string.

Instance Methods

byteValue

`public byte byteValue()`

Availability

New as of JDK 1.1

Returns

The value of this object as a `byte`.

Overrides

`Number.byteValue()`

Description

This method returns the truncated value of this object as a `byte`. More specifically, if the value of the object is `NaN`, the method returns 0. If the value is `POSITIVE_INFINITY`, or any other value that is too large to be represented by an `byte`, the method returns `Byte.MAX_VALUE`. If the value is `NEGATIVE_INFINITY`, or any other value that is too small to be represented by an `byte`, the method returns `Byte.MIN_VALUE`. Otherwise, the value is rounded toward zero and returned.

doubleValue

`public double doubleValue()`

Returns

The value of this object as a `double`.

Overrides

`Number.doubleValue()`

Description

This method returns the value of this object as a `double`.

equals

`public boolean equals(Object obj)`

Parameters

`obj`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `obj` is an instance of `Double` and it contains the same value as the object this method is associated with. More specifically, the method returns `true` if the `doubleToLongBits()` method returns the same result for the values of both objects.

This method produces a different result than the `==` operator when both values are `NaN`. In this case, the `==` operator produces `false`, while this method returns `true`. By the same token, the method also produces a different result when the two values are `+0.0` and `-0.0`. In this case, the `==` operator produces `true`, while this method returns `false`.

floatValue

`public float floatValue()`

Returns

The value of this object as a `float`.

Overrides

`Number.floatValue()`

Description

This method returns this object value as a `float`. Rounding may occur.

hashCode

`public int hashCode()`

Returns

A hashcode based on the `double` value of the object.

Overrides

`Object.hashCode()`

Description

This method returns a hashcode computed from the value of this object. More specifically, if `d` is the value of the object, and `bitValue` is defined as:

```
long bitValue = Double.doubleToLongBits(d)
```

then the hashcode returned by this method is computed as follows:

```
(int)(bitValue ^ (bitValue>>>32))
```

intValue

`public int intValue()`

Returns

The value of this object as an `int`.

Overrides

`Number.intValue()`

Description

This method returns the truncated value of this object as an `int`. More specifically, if the value of the object is `NaN`, the method returns 0. If the value is `POSITIVE_INFINITY`, or any other value that is too large to be represented by an `int`, the method returns `Integer.MAX_VALUE`. If the value is `NEGATIVE_INFINITY`, or any other value that is too small to be represented by an `int`, the method returns `Integer.MIN_VALUE`. Otherwise, the value is rounded toward zero and returned.

isInfinite

`public boolean isInfinite()`

Returns

> `true` if the value of this object is equal to `POSITIVE_INFINITY` or `NEGATIVE_INFINITY`; otherwise `false`.

Description

> This method determines whether or not the value of this object is an infinity value.

isNaN

`public boolean isNaN()`

Returns

> `true` if the value of this object is equal to `NaN`; otherwise `false`.

Description

> This method determines whether or not the value of this object is NaN.

longValue

`public long longValue()`

Returns

> The value of this object as a `long`.

Overrides

> `Number.longValue()`

Description

> This method returns the truncated value of this object as a `long`. More specifically, if the value of

the object is NaN, the method returns 0. If the value is POSITIVE_INFINITY, or any other value too large to be represented by a long, the method returns Long.MAX_VALUE. If the value is NEGATIVE_INFINITY, or any other value too small to be represented by a long, the method returns Long.MIN_VALUE. Otherwise, the value is rounded toward zero and returned.

shortValue

public short shortValue()

Availability

New as of JDK 1.1

Returns

The value of this object as a short.

Overrides

Number.shortValue()

Description

This method returns the truncated value of this object as a short. More specifically, if the value of the object is NaN, the method returns 0. If the value is POSITIVE_INFINITY, or any other value that is too large to be represented by an short, the method returns Short.MAX_VALUE. If the value is NEGATIVE_INFINITY, or any other value that is too small to be represented by an short, the method returns Short.MIN_VALUE. Otherwise, the value is rounded toward zero and returned.

toString

public String toString()

Returns

A string representation of the value of this object.

Overrides

```
Object.toString()
```

Description

This method returns a `String` object that contains a representation of the value of this object.

The values `NaN`, `NEGATIVE_INFINITY`, `POSITIVE_INFINITY`, `-0.0`, and `+0.0` are represented by the strings `"NaN"`, `"--Infinity"`, `"Infinity"`, `"--0.0"`, and `"0.0"`, respectively.

For other values, the exact string representation depends on the value being converted. If the absolute value of this object is greater than or equal to 10^-3 or less than or equal to 10^7, it is converted to a string with an optional minus sign (if the value is negative) followed by up to eight digits before the decimal point, a decimal point, and the necessary number of digits after the decimal point (but no trailing zero if there is more than one significant digit). There is always a minimum of one digit after the decimal point.

Otherwise, the value is converted to a string with an optional minus sign (if the value is negative), followed by a single digit, a decimal point, the necessary number of digits after the decimal point (but no trailing zero if there is more than one significant digit), and the letter `E` followed by a plus or a minus sign and a base 10 exponent of at least one digit. Again, there is always a minimum of one digit after the decimal point.

Note that the definition of this method has changed as of JDK 1.1. Prior to that release, the method provided a string representation that was equivalent to the `%g` format of the `printf` function in C.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

Class; Exceptions; Float; Floating-point literals; Floating-point types; Number; String

PREVIOUS HOME NEXT
Compiler BOOK INDEX Float

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Float

Name

Float

Synopsis

Class Name:

`java.lang.Float`

Superclass:

`java.lang.Number`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The `Float` class provides an object wrapper for a `float` value. This is useful when you need to treat a `float` value as an object. For example, there are a number of utility methods that take a reference to an `Object` as one of their arguments. You cannot specify a `float` value for one of these arguments, but you can provide a reference to a `Float` object that encapsulates the `float` value. Furthermore, as of JDK 1.1, the `Float` class is necessary to support the Reflection API and class literals.

In Java, `float` values are represented using the IEEE 754 format. The `Float` class provides constants for the three special values that are mandated by this format: `POSITIVE_INFINITY`, `NEGATIVE_INFINITY`, and `NaN` (not-a-number).

The `Float` class also provides some utility methods, such as methods for determining whether a `floatx` value is an infinity value or NaN, for converting `float` values to other primitive types, and for converting a `float` to a `String` and vice versa.

Class Summary

```
public final class java.lang.Float extends java.lang.Number {
    // Constants
    public static final float MIN_VALUE;
    public static final float MAX_VALUE;
    public static final float NaN;
    public static final float NEGATIVE_INFINITY;
    public static final float POSITIVE_INFINITY;
    public final static Class TYPE;                    // New in 1.1
    // Constructors
    public Float(double value);
    public Float(float value);
    public Float(String s);
    // Class Methods
    public static native int floatToIntBits(float value);
    public static native float intBitsToFloat(int bits);
    public static boolean isInfinite(float v);
    public static boolean isNaN(float v);
    public static String toString(float f);
    public static Float valueOf(String s);
    // Instance Methods
    public byte byteValue();                           // New in 1.1
    public double doubleValue();
    public boolean equals(Object obj);
    public float floatValue();
    public int hashCode();
    public int intValue();
    public boolean isInfinite();
    public boolean isNaN();
    public long longValue();
    public short shortValue();                         // New in 1.1
    public String toString();
}
```

Constants

MAX_VALUE

`public static final float MAX_VALUE = 3.40282346638528860e+38f`

Description

> The largest value that can be represented by a `float`.

MIN_VALUE

`public static final float MIN_VALUE = 1.40129846432481707e-45f`

Description

> The smallest value that can be represented by a `float`.

NaN

`public static final float NaN = 0.0f / 0.0f`

Description

> This variable represents the value NaN, a special value produced by `float` operations such as division of zero by zero. When NaN is one of the operands, most arithmetic operations return NaN as the result. Most comparison operators (`<`, `<=`, `==`, `>=`, `>`) return `false` when one of their arguments is NaN. The exception is `!=`, which returns `true` when one of its arguments is NaN.

NEGATIVE_INFINITY

`public static final float NEGATIVE_INFINITY = -1.0f / 0.0f`

Description

> This variable represents the value negative infinity, which is produced when a `float` operation underflows or a negative `float` value is divided by zero. Negative infinity is by definition less than any other `float` value.

POSITIVE_INFINITY

`public static final float POSITIVE_INFINITY = 1.0f / 0.0f`

Description

> This variable represents the value positive infinity, which is produced when a `float` operation overflows or a positive `float` value is divided by zero. Positive infinity is by definition greater than any other `float` value.

TYPE

`public static final Class TYPE`

Availability

> New as of JDK 1.1

Description

> The `Class` object that represents the type `float`. It is always true that `Float.TYPE` == `float.class`.

Constructors

Float

`public Float(double value)`

Parameters

`value`

The `double` value to be encapsulated by this object.

Description

Creates a `Float` object with the specified `double` value. The value is rounded to `float` precision.

public Float(float value)

Parameters

`value`

The `float` value to be encapsulated by this object.

Description

Creates a `Float` object with the specified `float` value.

public Float(String s) throws NumberFormatException

Parameters

`s`

The string to be made into a `Float` object.

Throws

`NumberFormatException`

If the sequence of characters in the given `String` does not form a valid `float` literal.

Description

Constructs a `Float` object with the value specified by the given string. The string must contain a sequence of characters that forms a legal `float` literal.

Class Methods

floatToIntBits

`public static native int floatToIntBits(float value)`

Parameters

> `value`
>
> > The `float` value to be converted.

Returns

> The `int` value that contains the same sequence of bits as the representation of the given `float` value.

Description

> This method returns the `int` value that contains the same sequence of bits as the representation of the given `float` value. The meaning of the bits in the result is defined by the IEEE 754 floating-point format: bit 31 is the sign bit, bits 30-23 are the exponent, and bits 22-0 are the mantissa. An argument of `POSITIVE_INFINITY` produces the result `0x7f800000`, an argument of `NEGATIVE_INFINITY` produces the result `0xff800000`, and an argument of `NaN` produces the result `0x7fc00000`.
>
> The value returned by this method can be converted back to the original `float` value by the `intBitsToFloat()` method.

intBitsToFloat

`public static native float intBitsToFloat(int bits)`

Parameters

> `bits`
>
> > The `int` value to be converted.

Returns

The `float` value whose representation is the same as the bits in the given `int` value.

Description

This method returns the `float` value whose representation is the same as the bits in the given `int` value. The meaning of the bits in the `int` value is defined by the IEEE 754 floating-point format: bit 31 is the sign bit, bits 30-23 are the exponent, and bits 22-0 are the mantissa. The argument `0x7f800000` produces the result `POSITIVE_INFINITY`, and the argument `0xff800000` produces the result `NEGATIVE_INFINITY`. Arguments in the ranges `0x7f800001` through `0x7f8fffff` and `0xff800001` through `0xff8fffffL` all produce the result `NaN`.

Except for NaN values not normally used by Java, this method is the inverse of the `floatToIntBits()` method.

isInfinite

`public static boolean isInfinite(float v)`

Parameters

`v`

The `float` value to be tested.

Returns

`true` if the specified value is equal to `POSITIVE_INFINITY` or `NEGATIVE_INFINITY`; otherwise `false`.

Description

This method determines whether or not the specified value is an infinity value.

isNaN

`public static boolean isNaN(float v)`

Parameters

`v`

The `float` value to be tested.

Returns

`true` if the specified value is equal to `NaN`; otherwise `false`.

Description

This method determines whether or not the specified value is NaN.

toString

`public static String toString(float f)`

Parameters

`f`

The `float` value to be converted.

Returns

A string representation of the given value.

Description

This method returns a `String` object that contains a representation of the given `float` value.

The values `NaN`, `NEGATIVE_INFINITY`, `POSITIVE_INFINITY`, `-0.0`, and `+0.0` are represented by the strings `"NaN"`, `"--Infinity"`, `"Infinity"`, `"--0.0"`, and `"0.0"`, respectively.

For other values, the exact string representation depends on the value being converted. If the absolute value of `f` is greater than or equal to 10^-3 or less than or equal to 10^7, it is converted to a string with an optional minus sign (if the value is negative) followed by up to eight digits before the decimal point, a decimal point, and the necessary number of digits after the decimal point (but no trailing zero if there is more than one significant digit). There is always a minimum of one digit after the decimal point.

Otherwise, the value is converted to a string with an optional minus sign (if the value is negative), followed by a single digit, a decimal point, the necessary number of digits after the decimal point (but no trailing zero if there is more than one significant digit), and the letter `E` followed by a plus or a minus sign and a base 10 exponent of at least one digit. Again, there is always a minimum of one digit after the decimal point.

Note that the definition of this method has changed as of JDK 1.1. Prior to that release, the method provided a string representation that was equivalent to the `%g` format of the `printf` function in C.

valueOf

```
public static Float valueOf(String s) throws NumberFormatException
```

Parameters

`s`

The string to be made into a `Float` object.

Returns

The `Float` object constructed from the string.

Throws

`NumberFormatException`

If the sequence of characters in the given `String` does not form a valid `float` literal.

Description

Constructs a `Float` object with the value specified by the given string. The string must contain a sequence of characters that forms a legal `float` literal. This method ignores leading and trailing whitespace in the string.

Instance Methods

byteValue

`public byte byteValue()`

Availability

New as of JDK 1.1

Returns

The value of this object as a `byte`.

Overrides

`Number.byteValue()`

Description

This method returns the truncated value of this object as a `byte`. More specifically, if the value of the object is `NaN`, the method returns 0. If the value is `POSITIVE_INFINITY`, or any other value that is too large to be represented by an `byte`, the method returns `Byte.MAX_VALUE`. If the value is `NEGATIVE_INFINITY`, or any other value that is too small to be represented by an `byte`, the method returns `Byte.MIN_VALUE`. Otherwise, the value is rounded toward zero and returned.

doubleValue

`public double doubleValue()`

Returns

The value of this object as a `double`.

Overrides

`Number.doubleValue()`

Description

This method returns the value of this object as a `double`.

equals

public boolean equals(Object obj)

Parameters

`obj`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `obj` is an instance of `Float` and it contains the same value as the object this method is associated with. More specifically, the method returns `true` if the `floatToIntBits()` method returns the same result for the values of both objects.

This method produces a different result than the `==` operator when both values are `NaN`. In this case, the `==` operator produces `false`, while this method returns `true`. By the same token, the method also produces a different result when the two values are `+0.0` and `-0.0`. In this case, the `==` operator produces `true`, while this method returns `false`.

floatValue

public float floatValue()

Returns

The value of this object as a `float`.

Overrides

`Number.floatValue()`

Description

This method returns the value of this object as a `float`.

hashCode

`public int hashCode()`

Returns

A hashcode based on the `float` value of the object.

Overrides

`Object.hashCode()`

Description

This method returns a hashcode computed from the value of this object. More specifically, if `f` is the value of the object, this method returns `Float.floatToIntBits(f)`.

intValue

`public int intValue()`

Returns

The value of this object as an `int`.

Overrides

`Number.intValue()`

Description

This method returns the truncated value of this object as an `int`. More specifically, if the value of the object is `NaN`, the method returns 0. If the value is `POSITIVE_INFINITY`, or any other value that is too large to be represented by an `int`, the method returns `Integer.MAX_VALUE`. If the value is `NEGATIVE_INFINITY`, or any other value that is too small to be represented by an `int`, the method returns `Integer.MIN_VALUE`. Otherwise, the value is rounded toward zero and returned.

isInfinite

`public boolean isInfinite(float v)`

Returns

`true` if the value of this object is equal to `POSITIVE_INFINITY` or `NEGATIVE_INFINITY`; otherwise `false`.

Description

This method determines whether or not the value of this object is an infinity value.

isNaN

`public boolean isNaN()`

Returns

`true` if the value of this object is equal to `NaN`; otherwise `false`.

Description

This method determines whether or not the value of this object is NaN.

longValue

`public long longValue()`

Returns

The value of this object as a `long`.

Overrides

`Number.longValue()`

Description

This method returns the truncated value of this object as a `long`. More specifically, if the value of the object is `NaN`, the method returns 0. If the value is `POSITIVE_INFINITY`, or any other value that is too large to be represented by a `long`, the method returns `Long.MAX_VALUE`. If the value is `NEGATIVE_INFINITY`, or any other value that is too small to be represented by a `long`, the method returns `Long.MIN_VALUE`. Otherwise, the value is rounded toward zero and returned.

shortValue

`public short shortValue()`

Availability

New as of JDK 1.1

Returns

The value of this object as a `short`.

Overrides

`Number.shortValue()`

Description

This method returns the truncated value of this object as a `short`. More specifically, if the value of the object is `NaN`, the method returns 0. If the value is `POSITIVE_INFINITY`, or any other value that is too large to be represented by an `short`, the method returns `Short.MAX_VALUE`. If the value is `NEGATIVE_INFINITY`, or any other value that is too small to be represented by an `short`, the method returns `Short.MIN_VALUE`. Otherwise, the value is rounded toward zero and returned.

toString

`public String toString()`

Returns

A string representation of the value of this object.

Overrides

`Object.toString()`

Description

This method returns a `String` object that contains a representation of the value of this object.

The values `NaN`, `NEGATIVE_INFINITY`, `POSITIVE_INFINITY`, `-0.0`, and `+0.0` are represented by the strings `"NaN"`, `"--Infinity"`, `"Infinity"`, `"--0.0"`, and `"0.0"`, respectively.

For other values, the exact string representation depends on the value being converted. If the absolute value of this object is greater than or equal to 10^-3 or less than or equal to 10^7, it is converted to a string with an optional minus sign (if the value is negative) followed by up to eight digits before the decimal point, a decimal point, and the necessary number of digits after the decimal point (but no trailing zero if there is more than one significant digit). There is always a minimum of one digit after the decimal point.

Otherwise, the value is converted to a string with an optional minus sign (if the value is negative), followed by a single digit, a decimal point, the necessary number of digits after the decimal point (but no trailing zero if there is more than one significant digit), and the letter `E` followed by a plus or a minus sign and a base 10 exponent of at least one digit. Again, there is always a minimum of one digit after the decimal point.

Note that the definition of this method has changed as of JDK 1.1. Prior to that release, the method provided a string representation that was equivalent to the `%g` format of the `printf` function in C.

Inherited Methods

Method	**Inherited From**	**Method**	**Inherited From**
`clone()`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

Class; Double; Exceptions; Floating-point literals; Floating-point types; Number; String

PREVIOUS HOME NEXT
Double BOOK INDEX Integer

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

4.11 Boolean Operators

The Boolean operators in Java are used for conditional AND (&&) and conditional OR (||) operations. These operators have different precedence; the && operator has the higher precedence and || the lower precedence. Both of the operators are evaluated from left to right.

The unary operator ! provides a Boolean negation operation.

References Boolean Negation Operator !; Order of Operations

Boolean AND Operator &&

The conditional AND operator && produces a pure `boolean` value that is the conditional AND of its operands. The && operator may appear in a conditional AND expression:

[Graphic: Figure from the text]

The conditional AND operator is evaluated from left to right. The operator never throws an exception.

Here is a code example that shows the use of the conditional AND operator:

```
public final short readShort() throws IOException {
    int ch1, ch2;
    if ((ch1 = in.read()) >= 0 && (ch2 = in.read()) >= 0)
        return (short)((ch1 << 8) + ch2);
    throw new EOFException();
}
```

The operands of the conditional AND operator must both be of type `boolean`, or a compile-time error occurs.

The operands of the conditional AND operator are evaluated in a different way from the operands for most other operators in Java. Most other operators evaluate all of their operands before performing their operation; the conditional AND operator does not necessarily evaluate both of its operands.

As with all binary operators, the left operand of && is evaluated first. If the left operand evaluates to `true`, the conditional AND operator evaluates its right operand and produces a pure value that has the same value as its right operand. However, if the left operand evaluates to `false`, the right operand is not evaluated and the operator produces the pure value `false`.

In the above example, the expression `(ch2 = in.read())` is evaluated only if the expression `(ch1 = in.read())` produces a value that is greater than or equal to zero.

References Bitwise/Logical AND Operator &; Boolean Type; Bitwise/Logical Inclusive OR Operator |; Order of Operations

Boolean OR Operator ||

The conditional OR operator `||` produces a pure `boolean` value that is the conditional OR of its operands. The `||` operator may appear in a conditional OR expression:

[Graphic: Figure from the text]

The conditional OR operator is evaluated from left to right. The operator never throws an exception.

Here is a code example that shows the use of the conditional OR operator:

```
public final short readShort() throws IOException {
    int ch1, ch2;
    if ((ch1 = in.read()) < 0 || (ch2 = in.read()) < 0)
        throw new EOFException();
    return (short)((ch1 << 8) + ch2);
}
```

The operands of the conditional OR operator must both be of type `boolean`, or a compile-time error occurs.

The operands of the conditional OR operator are evaluated in a different way from the operands for most other operators in Java. Most other operators evaluate all of their operands before performing their operation; the conditional OR operator does not necessarily evaluate both of its operands.

As with all binary operators, the left operand of `||` is evaluated first. If the left operand evaluates to `false`, the conditional OR operator evaluates its right operand and produces a pure value that has the same value as its right operand. However, if the left operand evaluates to `true`, the right operand is not evaluated and the operator produces the pure value `true`.

References Bitwise/Logical Inclusive OR Operator |; Boolean Type; Boolean AND Operator &&; Order of Operations

6.7 Iteration Statements

Iteration statements are used to specify the logic of a loop. Java has three varieties of iteration statement: `while`, `do`, and `for`.

[Graphic: Figure from the text]

References The do Statement; The for Statement; The while Statement

The while Statement

A `while` statement evaluates a Boolean expression. If the expression is `true`, a given statement is repeatedly executed for as long as the expression continues to evaluate to `true`.

[Graphic: Figure from the text]

In Java, the expression in parentheses must produce a `boolean` value. This is different from C/C++, which allows any type of expression.

If the expression in parentheses evaluates to `true`, the statement contained in the `while` statement is executed and the expression in parentheses is evaluated again. This process continues until the expression evaluates to `false`.

If the expression in parentheses evaluates to `false`, the statement following the `while` statement is the next statement to be executed. The expression in parentheses is evaluated before the contained statement is executed, so it is possible for the contained statement not to be executed even once.

Here is an example of a `while` statement:

```
while ( (c = in.read()) >= 0) {
    out.write(c);
}
```

References Boolean Type; *Expression* 4; *Statement* 6

The do Statement

A `do` statement executes a given statement and then evaluates a Boolean expression. If the expression evaluates to `true`, the statement is executed repeatedly as long as the expression continues to evaluate to `true`:

[Graphic: Figure from the text]

In Java, the expression in parentheses must produce a `boolean` value. This is unlike C/C++, which allows any type of expression.

The statement contained in the `do` statement is executed and then the expression in parentheses is evaluated. If the expression evaluates to `true`, the process is repeated.

If the expression evaluates to false, the statement following the `do` statement is the next statement to be executed. Because the expression is evaluated after the contained statement is executed, the statement is always executed at least once.

Here's an example of a `do` statement:

```
do {
    c = in.read();
    out.write(c);
} while (c != ';');
```

References Boolean Type; *Expression* 4; *Statement* 6

The for Statement

A `for` statement is a more structured form of a `while` statement. A `for` statement performs an initialization step and then evaluates a Boolean expression. If the expression evaluates to `true`, a given statement is executed and an increment expression is evaluated repeatedly as long as the expression continues to evaluate to `true`:

[Graphic: Figure from the text]

[Graphic: Figure from the text]

[Graphic: Figure from the text]

Here is an example of a `for` statement:

```
for (i = 0; i < a.length; i++) {
    a[i] = i;
}
```

The initialization part of the `for` statement is executed first. If the initialization part contains nothing, no initialization is performed. The expression that follows must produce a `boolean` value. Before the body of the `for` statement is executed, the expression is evaluated. If the expression portion of the `for` statement is omitted, the default expression `true` is used. If the expression evaluates to `true`, the body of the `for` statement is executed and then the increment portion of the `for` statement is evaluated. Finally, the expression is evaluated again to determine if there should be another iteration. This process continues until the expression evaluates to `false`, at which point the statement following the `for` statement is the next statement to be executed. The `for` statement in the above example can be rewritten as a `while` statement as follows:

```
i = 0;
while (i < a.length) {
    a[i] = i;
    i++;
}
```

One difference between comparable `for` and `while` loops is that a `continue` statement in the body of a `for` statement causes the increment portion of the statement to be evaluated. However, this may not be the case in a comparable `while` statement.

Here's a new version of our `for` example:

```
for (i = 0; i < a.length; i++) {
    a[i] = i;
    continue;
}
```

The added `continue` statement at the end of the `for` loop does not change the behavior of the loop. In particular, `i++` is still evaluated after each iteration through the body of the loop. Now let's add a `continue` statement at the equivalent place in our `while` example:

```
i = 0;
while (i < a.length) {
    a[i] = i;
    continue;
    i++;
}
```

The `continue` statement in this `while` loop prevents the statement `i++` from being executed. The `continue` statement would have to be moved after the increment operation to match the logic of the `for` statement.

If the expression portion of a `for` statement is omitted, the default expression `true` is supplied. Take, for example, the following `for` statement:

```
for ( FileInputStream in = new FileInputStream(fname);;) {
    c = in.read();
    if (c < 0)
        return;
    System.out.print((char)c);
}
```

This example uses a local variable declaration in the initialization portion of the `for` statement. Local variable declarations in a `for` statement are subject to the same restrictions as local variable declarations in a block. In particular, a `for` statement cannot declare a local variable with the same name as a local variable or formal parameter that is defined in an enclosing block.

The above `for` statement is equivalent to the following `while` statement:

```
{
    FileInputStream in = new FileInputStream(fname);
    while (true) {
        c = in.read();
        if (c < 0)
            return;
        System.out.print((char)c);
    }
}
```

The enclosing block in the above example is provided to limit the scope of the local variable `in` to just the `while` statement.

The initialization portion of a `for` statement can also be empty. The following statement is a legal way of specifying an infinite loop:

```
for (;;) {...}
```

This is equivalent to the following `while` statement:

```
while (true) {...}
```

Unlike C/C++, there is no comma operator in Java. However, commas are explicitly allowed in the initialization portion of a `for` statement. For example, a `for` initialization can consist of multiple expressions separated by commas:

```
i=2, j=5, k=44
```

When the initialization portion of a `for` statement contains local variable declarations, commas are also allowed because the syntax for declarations allows multiple variables, separated by commas, to be declared in one declaration. For example:

```
int i=2, j=5, k=44
```

References Boolean Type; *Expression* 4; *Statement* 6; Local Variables; *TopLevelExpression* 6.4; The continue Statement; The while Statement

PREVIOUS
The switch Statement

HOME
BOOK INDEX

NEXT
The break Statement

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

6.5 The if Statement

An if statement determines which of two statements is executed, based on the value of a Boolean expression:

[Graphic: Figure from the text]

In Java, the expression in parentheses must produce a boolean value. This is different from C/C++, which allows any type of expression.

If the expression in parentheses evaluates to true, the statement after the parentheses is executed. After that statement has been executed, the statement following the entire if statement is executed. If the expression between the parentheses evaluates to false, the next statement to be executed depends on whether or not the if statement has an else clause. If there is an else clause, the statement after the else is executed. Otherwise, the statement after the entire if statement is executed.

When if statements are nested, each else clause is matched with the last preceding if statement in the same block that has not yet been matched with an if statement.

Here is an example of an if statement:

```
if (j == 4) {
    if (x > 0 ) {
        x *= 7;
    } else {
        x *= -7;
    }
}
return;
```

The outer `if` statement has no `else` clause. If `j` is not 4, the `return` statement is executed. Otherwise, the inner `if` statement is executed. This `if` statement does have an `else` clause. If `x` is greater than zero, the value of `x` is multiplied by 7. Otherwise, the value of `x` is multiplied by -7. Regardless of the value of `x`, the `return` statement is executed.

References Boolean Type; *Expression* 4; *Statement* 6

PREVIOUS | HOME | NEXT
Expression Statements | BOOK INDEX | The switch Statement

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

4.2 Allocation Expressions

An *allocation expression* is a primary expression that creates an object or an array. An allocation expression also produces a reference to the newly created object or array:

[Graphic: Figure from the text]

[Graphic: Figure from the text]

When *AllocationExpression* contains parentheses, the allocation expression creates a non-array object. When *AllocationExpression* contains square brackets, the allocation expression creates an array.

An object created by an allocation expression continues to exist until the program terminates or it is freed by the garbage collector (see Object Destruction). Consider the following code:

```
class X {
    Double perm;
    void foo() {
        Double d = new Double(8.9473);
        int a[] = new int [2];
        d = new Double(3.1415926);
        a[0] = d.intValue();
        perm = d;
    }
}
```

The first line of `foo()` creates a `Double` object and uses it as the initial value of the variable `d`. The second line creates an array of integers and uses it as the initial value of the variable `a`. At this point, neither of the two objects that has been created is a candidate for garbage collection because there is a variable referencing each of them.

The third line of `foo` creates another `Double` object and assigns it to the variable `d`. Now there is nothing that refers to the first `Double` object that we created, so that object can now be garbage collected at any time.

When the block in this example finishes executing, the variables declared inside of the block, `a` and `d`, pass out of scope. Now there is nothing referring to the array object that we created; now that object can be garbage-collected at any time. However, because the variable `perm` now refers to the second `Double` object we created, that object is not a candidate for garbage collection.

References *ArgumentList* 4.1.8; *ClassBody* 5.4; Variable initializers; *Expression* 4; Identifiers; Object Creation; Object Destruction; Primary Expressions; *Type* 3

Object Allocation Expressions

An allocation expression that contains parentheses creates a non-array object; that is, an instance of a class. For example:

```
new Double(93.1872)
```

The *Type* in an object allocation expression must be a class or interface type. The argument list supplied between the parentheses provides the actual arguments to be passed to the object's constructor. However, if a *ClassBody* follows the parentheses, no arguments may appear between the parentheses, and different rules apply. (These rules are discussed in Allocating instances of anonymous classes.)

If `new` is preceded by a *PrimaryExpression* and a dot, the *PrimaryExpression* must produce a reference to an object. Furthermore, the object's class must be an inner or nested top-level class that is named by the identifier that follows `new`. If the specified class is a non-`static` inner class, the object created by the allocation expression has the object referenced by the *PrimaryExpression* as its enclosing instance. For example:

```
class Z {
    class Y {
    ...
    }
    public static void main(String[] argv) {
        Z myZ = new Z();
        Z.Y myY = myZ.new Y();
    }
}
```

In the preceding example, we must supply an explicit enclosing instance of `Z` to create a `Z.Y` object because `main()` is a `static` method. A non-`static` method of `Z` could create an instance of `Z.Y` without supplying an explicit enclosing instance of `Z` because the method itself is associated with an instance of `Z`. However, because a `static` method is not associated with an instance of its class, it must supply an explicit enclosing instance when creating an instance of an inner class.

The syntax that allows `new` to be preceded by a *PrimaryExpression* and a dot is not supported prior to Java 1.1.

The remainder of this section applies only to allocation expressions that contain parentheses but no *ClassBody*. Allocation expressions that contain a *ClassBody* are described in Allocating instances of anonymous classes.

An object allocation expression performs the following steps:

1. Creates a new object with all of its instance variables set to their default values. The default values for these variables are determined by their types.

2. Calls the constructor that matches the given argument list.

3. Produces a reference to the initialized object.

The process of selecting the appropriate constructor to call is similar to the process used to select the method invoked by a method call expression. The compiler determines which constructors have formal parameters compatible with the given arguments. If there is more than one suitable constructor, the compiler must select the constructor that most closely matches the given arguments. If the compiler cannot select one constructor as a better match than the others, the constructor selection process fails and an error message is issued.

Here are the details of the constructor selection process:

Step One

The constructor definitions are searched for constructors that, taken in isolation, could be called by the allocation expression. A constructor is a candidate if it meets the following criteria:

- The constructor is accessible to the allocation expression, based on any access modifiers specified in the constructor's declaration.

- The number of formal parameters declared for the constructor is the same as the number of actual arguments provided in the allocation expression.
- The data type of each actual parameter is assignment-compatible with the corresponding formal parameter.

Step Two

> If more than one constructor meets the criteria in Step One, the Java compiler tries to determine if one constructor is a more specific match than the others. If there is no constructor that matches more specifically, the constructor selection process fails and an error message is issued.
>
> Given two constructors that are both candidates to be invoked by the same object allocation expression, one constructor is more specific than another constructor if each parameter of the first constructor is assignment-compatible with the corresponding parameter of the second constructor.

When an object allocation expression is evaluated, the constructor selected in Step Two is invoked. This constructor returns a reference to the newly created object.

Here's an example that shows how the constructor selection process works:

```
class Consel {
    Consel() { }
    Consel(Object o, double d) {}
    Consel(String s, int i) {}
    Consel(int i, int j) {}
    public void main(String[] argv) {
        Consel c = new Consel("abc",4);
    }
}
```

The `main()` method in the `Consel` class creates a new `Consel` object. The process of selecting which constructor to call proceeds as follows:

1. The compiler finds all of the constructors that are accessible to the `new` operator. Since all of the constructors are accessible, the compiler then narrows its choices to those constructors that have the same number of formal parameters as the number of actual arguments in the allocation expression. This step eliminates the constructor with no formal parameters, so now there are three choices. The compiler again narrows its choices to those constructors with formal parameters that are assignment-compatible with the actual values. Because a `String` is not assignment-compatible with an `int` variable, the compiler eliminates the constructor that takes two `int`

parameters.

2. Now the compiler must choose which of the two remaining constructors is more specific than the other. Because a `String` object reference can be assigned to an `Object` variable and an `int` value can be assigned to a `double` variable, the constructor `Consel(String s, int i)` is the more specific of the two. This constructor is the one that is invoked to create the `Consel` object.

References Allocating instances of anonymous classes; Assignment Compatibility; *ClassBody* 5.4; Class Types; Constructors; Interface Types; Primary Expressions

Allocating instances of anonymous classes

An allocation expression that contains a *ClassBody* creates an instance of an *anonymous class*. It is called an anonymous class because it has no name of its own. The variables and methods of an anonymous class are defined in the *ClassBody*. If the type specified after `new` is a class, the anonymous class is a subclass of that class. If the type specified after `new` is an interface, the anonymous class implements that interface and is a subclass of `Object`. For example:

```
public class MainFrame extends Frame {
...
    public MainFrame(String title) {
        super(title);
        WindowAdapter listener;
        listener = new WindowAdapter() {
            public void windowClosing(WindowEvent evt) {
                exit();
            }
        };
        addWindowListener(listener);
    }
...
}
```

The example creates an instance of an anonymous subclass of the `WindowAdapter` class. If an allocation expression contains a *ClassBody*, it cannot contain any arguments between the parentheses because an anonymous class cannot declare any constructors. Instead, an anonymous class must use instance initializers to handle any complex initialization.

The body of an anonymous class cannot declare any `static` variables, `static` methods, `static` classes, or static initializers. Anonymous classes are not supported prior to Java 1.1.

References Anonymous classes; *ClassBody* 5.4; Constructors; Instance Initializers; Methods; Nested Top-Level and Member Classes; Static Initializers; Variables

Array Allocation Expressions

An allocation expression that contains square brackets creates an array, such as:

```
new int[10]
```

An array allocation expression performs the following steps:

1. Allocates storage for the array

2. Sets the `length` variable of the array and initializes the array elements to their default values

3. Produces a reference to the initialized array

Although Java does not support multi-dimensional arrays, it does support arrays of arrays. The most important distinction between a multi-dimensional array and an array of arrays is that in an array of arrays, each array need not be of the same length. Because arrays of arrays are most often used to represent multi-dimensional arrays, this book refers to them as multi-dimensional arrays, even though that is not precisely correct.

The type of the array created by an array allocation expression can be expressed by removing both the word `new` and the expressions from within the square brackets. For example, here is an allocation expression:

```
new int[3][4][5]
```

The type of the array produced by that expression is:

```
int[][][]
```

This means that the number of dimensions in the array produced by an allocation expression is the same as the number of pairs of square brackets in the allocation expression.

The expressions that appear in the square brackets must be of type `int`, `short`, `char`, or `byte`. Each of the expressions specifies the length of a single dimension of the array that is being created. For example, the allocation expression above creates an array of 3 arrays of 4 arrays of 5 `int` values. The length supplied for an array must not be negative. At runtime, if an expression in square brackets produces a negative array length, a `NegativeArraySizeException` is thrown.

The syntax of an array allocation expression specifies that the first pair of square brackets must contain an expression, while the trailing square brackets do not need to. This means that an array allocation expression can be written to build fewer dimensions of an array than there are dimensions in the array's type. For example, consider this allocation expression:

```
new char [10][]
```

The array produced by this allocation expression is an array of arrays of `char`. The allocation expression creates a single array of 10 elements, where each of those elements is a `char` array of unspecified length.

Array allocation expressions are often used to initialize array variables. Here are some examples:

```
int j[] = new int[10];           // array of 10 ints
ing k[][] = new float[3][4];     // array of 3 arrays
                                 // of 4 floats
```

Here's an example that builds an array of different length arrays, or in other words a non-rectangular array of arrays:

```
int a[][] = new int [3][];
a[0] = new int [5];
a[1] = new int [6];
a[2] = new int [7];
```

None of the array allocation expressions presented so far have used array initializers. When an array allocation expression does not include an array initializer, the array is created with all of its elements set to a default value. The default value is based on the type of the array. Table 4-1 shows the default values used for the various types in Java.

Table 4.1: Default Values for Array Elements

Array Type	**Default Value**
`byte`	`0`
`char`	`'\u0000'`
`short`	`0`
`int`	`0`

`long`	`0L`
`float`	`0.0F`
`double`	`0.0`
`Boolean`	`false`
Object reference	`null`

If you want to create an array that contains elements with different initial values, you can include an *ArrayInitializer* at the end of the allocation expression. For example:

```
new int [] { 4,7,9 }
```

Notice that there is no expression between the square brackets. If an allocation expression contains square brackets and no *ArrayInitializer*, at least the first pair of square brackets must contain an expression. However, if an allocation expression does contain an *ArrayInitializer*, there cannot be any expressions between any of the square brackets. An allocation expression that contains an *ArrayInitializer* can be used to create an anonymous array: one that is created and initialized without using a variable initializer.

The syntax that allows an *ArrayInitializer* in an allocation expression is not supported prior to Java 1.1.

References Array Types; Variable initializers; Index Expressions

PREVIOUS
Primary Expressions

HOME
BOOK INDEX

NEXT
Increment/Decrement Operators

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

4.14 Order of Operations

In an expression that contains multiple operators, Java uses a number of rules to decide the order in which the operators are evaluated. The first and most important rule is called *operator precedence*. Operators in an expression that have higher precedence are executed before operators with lower precedence. For example, multiplication has a higher precedence than addition. In the expression `2+3*4`, the multiplication is done before the addition, producing a result of 14.

If consecutive operators in an expression have the same precedence, a rule called *associativity* is used to decide the order in which those operators are evaluated. An operator can be left-associative, right-associative, or non-associative:

- Left-associative operators of the same precedence are evaluated in order from left to right. For example, addition and subtraction have the same precedence and they are left-associative. In the expression `10-4+2`, the subtraction is done first because it is to the left of the addition, producing a value of 8.
- Right-associative operators of the same precedence are evaluated in order from right to left. For example, assignment is right-associative. Consider the following code fragment:

  ```
  int a = 3;
  int b = 4;
  a = b = 5;
  ```

  After the code has been evaluated, both `a` and `b` contain 5 because the assignments are evaluated from right to left.
- A non-associative operator cannot be combined with other operators of the same precedence.

Table 4-2 shows the precedence and associativity of all the operators in Java.[7]

[7] Although the precedence of operators in Java is similar to that in C++, there are some differences. For example, new has a higher precedence in Java than it does in C++. Another difference is that the ++ and - - operators are effectively non-associative in Java.

Table 4.2: Precedence and Associativity of Operators in Java

Precedence	Operator	Associativity
1	(), []	non-associative
2	new	non-associative
3	.	left-associative
4	++, - -	non-associative
5	- (unary), + (unary), !, ~, ++, - -, (*type*)	right-associative
6	*, /, %	left-associative
7	+, -	left-associative
8	<<, >>, >>>	left-associative
9	<, >, <=, >=, instanceof	non-associative
10	==, !=	left-associative
11	&	left-associative
12	^	left-associative
13	|	left-associative
14	&&	left-associative
15	||	left-associative
16	?:	right-associative
17	=, *=, /=, %=, -=, <<=, >>=, >>>=, &=, ^=, |=	right-associative

As in C/C++, the order in which operators are evaluated can be modified by the use of parentheses.

The rest of the rules that concern order of operations have to do with the evaluation of operands or arguments in a single expression.

- The left operand of a binary operator is evaluated before its right operand.

- The operands of an operator are evaluated before the operator is evaluated. Consider the following expression:

```
((x=4) * x)
```

First, the left operand of * is evaluated; it produces the value 4. Then the right operand of * is evaluated. Since evaluation of the left operand set x to 4, evaluation of the right operand produces 4. Finally, the * operator itself is evaluated, producing the value 16.

- In an index expression, the expression to the left of the square brackets is evaluated before the expression inside the square brackets.
- In an expression that calls a method through an object reference, the object reference is evaluated before the argument expressions.
- In any expression that calls a method or constructor, the expressions supplied as the actual arguments are evaluated from left to right.
- In an array allocation expression, the expressions that appear in square brackets and provide the dimensions of the array are evaluated from left to right.

The intent of all of these rules is to guarantee that every implementation of Java evaluates any given expression in the same way.[8] In order to produce optimized code, a Java compiler is allowed to deviate from the rules governing the order in which operations are performed, provided that the result is the same as if it had followed the rules.

[8] This is different than C/C++, which leaves a number of details of expression evaluation up to an implementation, such as the order in which the actual parameters of a function call are evaluated.

References Array Allocation Expressions; Index Expressions; Method Call Expression; Object Allocation Expressions

PREVIOUS	HOME	NEXT
Assignment Operators	BOOK INDEX	Data Type of an Expression

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

4.16 Constant Expressions

A constant expression is an expression that always produces the same result. More precisely, a constant expression is an expression that produces a pure value of a primitive data type and is only composed of the following:

- Literals of primitive data types
- String literals
- Variables that are declared `final` and are initialized by constant expressions
- Type casts to primitive data types or the type `String`
- The unary operators `+ -`, `~`, and `!`
- The binary operators `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `>>>`, `<`, `<=`, `>=`, `>`, `==`, `!=`, `&`, `^`, `|`, `&&`, and `||`
- The ternary operator `?:`

Note that expressions that use `++`, `- -`, and `instanceof` are not constant expressions. Also note that expressions that produce or contain references to objects that are not `String` objects are never constant expressions.

The compiler generally evaluates a constant expression and substitutes the result for the expression during the compilation process.

References Additive Operators; Bitwise/Logical Operators; Boolean Operators; Casts; Conditional Operator; Equality Comparison Operators; Interface Variables; Local Variables; Literals; Multiplicative Operators; Relational Comparison Operators; Shift Operators; Unary Operators; Variables

5.3 Object-Orientation Java Style

Before considering class and interface declarations in Java, it is essential that you understand the object-oriented model used by the language. No useful programs can be written in Java without using objects. Java deliberately omits certain C++ features that promote a less object-oriented style of programming. Thus, all executable code in a Java program must be part of an object (or a class to be more precise).

The two main characteristics of objects in Java are:

- Objects are always dynamically allocated. The lifetime of the storage occupied by an object is determined by the program's logic, not by the lifetime of a procedure call or the boundaries of a block. The lifetime of the storage occupied by an object refers to the span of time that begins when the object is created and ends at the earliest time it can be freed by the garbage collector.

- Objects are not contained by variables. Instead, a variable contains a reference to an object. A reference is similar to what is called a pointer in some other languages. If there are two variables of the same reference type and one of the variables is assigned to the other, both variables refer to the same object. If the information in that object is changed, the change is visible through both variables.

Classes

An object is a collection of variables, associated methods, and other associated classes. Objects in Java are described by *classes*; a particular object is an *instance* of a particular class. A class describes the data an object can contain by defining variables to contain the data in each instance of the class. A class describes the behavior of an object by defining methods for the class and possibly other auxiliary classes. Methods are named pieces of executable code; they are similar to what other programming languages call functions or procedures. Collectively, the variables, methods, and auxiliary classes of a class are called its members.

A class can define multiple methods with the same name if the number or type of parameters for each method is different. Multiple methods with the same name are called *overloaded methods*. Like C++, Java supports overloaded methods, but unlike C++, Java does not support overloaded operators. Overloaded methods are useful when you want to describe similar operations on different types of data. For example, Java provides a class called `java.io.OutputStream` that is used to write data. The `OutputStream` class defines three different `write()` methods: one to write a single byte of data, another to write some of the bytes in an array, and another to write all of the bytes in an array.

References Class Declarations

Encapsulation

Encapsulation is the technique of hiding the details of the implementation of an object, while making its functionality available to other objects. When encapsulation is used properly, you can change an object's implementation without worrying that any other object can see, and therefore depend on, the implementation details.

The portion of an object that is accessible to other types of objects is called the object's *interface*.[1] For example, consider a class called `Square`. The interface for this class might consist of:

> [1] The notion of an object's interface is a commonly accepted concept in the object-oriented community. Later in this chapter, a Java construct called an interface is described. A Java interface is not the same thing as the interface of an object, so there is some potential for confusion. Outside of this section, the term "interface" is only used to mean the Java interface construct.

- Methods to get and set the size of a square.

- A method to tell a square to draw itself at a particular location on the screen.

The implementation of this `Square` class would include executable code that implements the various methods, as well as an internal variable that an object would use to remember its size. Variables that an object uses to remember things about itself are called *state variables*.

The point of the distinction between the interface and the implementation of a class is that it makes programs easier to maintain. The implementation of a class may change, but as long as the interface remains the same, these changes do not require changes to any other classes that may use the class.

In Java, encapsulation is implemented using the `public`, `protected`, and `private` access modifiers. If a field of a class is part of the interface for the class, the field should be declared with the `public` modifier or with no access modifier. The `private` and `protected` modifiers limit the accessibility of a field, so these modifiers should be used for state variables and other implementation-specific functionality.

Here's a partial definition of a `Square` class that has the interface just described:

```
class Square {
    private int sideLength;
    public void setSideLength(int len) {
        sideLength = len;
    }
    public int getSideLength() {
        return sideLength;
    }
    public void draw(int x, int y) {
        // code to draw the square
        ...
    }
}
```

References Method modifiers; Inner class modifiers; Variable modifiers

Object Creation

An object is typically created using an allocation expression. The `newInstance()` methods of the `Class` or `java.lang.reflect.Contructor` class can also be used to create an instance of a class. In either case, the storage needed for the object is allocated by the system.

When a class is instantiated, a special kind of method called a *constructor* is invoked. A constructor for a class does not have its own name; instead it has the same name as the class of which it is a part. Constructors can have parameters, just like regular methods, and they can be overloaded, so a class can have multiple constructors. A constructor does not have a return type. The main purpose of a constructor is to do any initialization that is necessary for an object.

If a class declaration does not define any constructors, Java supplies a default `public` constructor that takes no parameters. You can prevent a class from being instantiated by methods in other classes by defining at least one `private` constructor for the class without defining any `public` constructors.

References Class; Constructors; Object Allocation Expressions

Object Destruction

Java does not provide any way to explicitly destroy an object. Instead, an object is automatically destroyed when the garbage collector detects that it is safe to do so. The idea behind *garbage collection* is that if it is possible to prove that a piece of storage will never be accessed again, that piece of storage can be freed for reuse. This is a more reliable way of managing storage than having a program explicitly deallocate its own storage. Explicit memory allocation and deallocation is the single largest source of programming errors in C/C++. Java eliminates this source of errors by handling the deallocation of memory for you.

Java's garbage collector runs continuously in a low priority thread. You can cause the garbage collector to take a single pass through allocated storage by calling `System.gc()`.

Garbage collection will never free storage before it is safe to do so. However, garbage collection usually does not free storage as soon as it would be freed using explicit deallocation. The logic of a program can sometimes help the garbage collector recognize that it is safe to free some storage sooner rather than later. Consider the following code:

```
class G {
    byte[] buf;
    String readIt(FileInputStream f) throws IOException {
        buf = new byte[20000];
        int length = f.read(buf);
        return new String(buf, 0, 0, length);
    }
}
```

The first time `readIt()` is called, it allocates an array that is referenced by the instance variable `buf`. The variable `buf` continues to refer to the array until the next time that `readIt()` is called, when `buf` is set to a new array. Since there is no longer any reference to the old array, the garbage collector will free the storage on its next pass. This situation is less than optimal. It would be better if the garbage collector could recognize that the array is no longer needed once a call to `readIt()` returns. Defining the variable `buf` as a local variable in `readIt()` solves this problem:

```
class G {
    String readIt(FileInputStream f) throws IOException {
        byte[] buf;
        buf = new byte[20000];
        int length = f.read(buf);
        return new String(buf, 0, 0, length);
    }
}
```

Now the reference to the array is in a local variable that disappears when `readIt()` returns. After `readIt()` returns, there is no longer any reference to the array, so the garbage collector will free the storage on its next pass.

Just as a constructor is called when an object is created, there is a special method that is called before an object is destroyed by the garbage collector. This method is called a *finalizer* ; it has the name `finalize()`. A `finalize()` method is similar to a destructor in C++. The `finalize()` method for a class must be declared with no parameters, the `void` return type, and no modifiers. A finalizer can be used to clean up after a class, by doing such things as closing files and terminating network connections.

If an object has a `finalize()` method, it is normally called by the garbage collector before the object is destroyed. A program can also explicitly call an object's `finalize()` method, but in this case, the garbage collector does not call the method during the object destruction process. If the garbage collector does call an object's `finalize()` method, the garbage collector does not immediately destroy the object because the `finalize()` method might do something that causes a variable to refer to the object again.[2] Thus the garbage collector waits to destroy the object until it can again prove it is safe to do so. The next time the garbage collector decides it is safe to destroy the object, it does so without calling the finalizer again. In any case, a `finalize()` method is never called more than once for a particular object.

> [2] A `finalize()` method should not normally do something that results in a reference to the object being destroyed, but Java does not do anything to prevent this situation from happening.

The garbage collector guarantees that the thread it uses to call a `finalize()` method will not be holding any programmer-visible synchronization locks when the method is called. This means that a `finalize()` method never has to wait for the garbage collector to release a lock. If the garbage collector calls a `finalize()` method and the `finalize()` method throws any kind of exception, the garbage collector catches and ignores the exception.

References System; The finalize method

Inheritance

One of the most important benefits of object-oriented programming is that it promotes the reuse of code, particularly by means of inheritance. *Inheritance* is a way of organizing related classes so that they can share common code and state information. Given an existing class declaration, you can create a similar class by having it inherit all of the fields in the existing definition. Then you can add any fields that are needed in the new class. In addition, you can replace any methods that need to behave differently in the new class.

To illustrate the way that inheritance works, let's start with the following class definition:

```
class RegularPolygon {
    private int numberOfSides;
    private int sideLength;
    RegularPolygon(int n, int len) {
        numberOfSides = n;
        sideLength = len;
    }
    public void setSideLength(int len) {
        sideLength = len;
    }
    public int getSideLength() {
        return sideLength;
    }
    public void draw(int x, int y) {
        // code to draw the regular polygon
        ...
    }
}
```

The `RegularPolygon` class defines a constructor, methods to set and get the side length of the regular polygon, and a method to draw the regular polygon. Suppose that after writing this class you realize that you have been using it to draw a lot of squares. You can use inheritance to build a more specific `Square` class from the existing `RegularPolygon` class as follows:

```
class Square extends   RegularPolygon {
    Square(int len) {
        super(4,len);
    }
}
```

The `extends` clause indicates that the `Square` class is a *subclass* of the `RegularPolygon` class, or looked at another way, `RegularPolygon` is a *superclass* of `Square`. When one class is a subclass of another class, the subclass inherits all of the fields of its superclass that are not `private`. Thus `Square` inherits `setSideLength()`, `getSideLength()`, and `draw()` methods from `RegularPolygon`. These methods work fine without any modification, which is why the definition of `Square` is so short. All the `Square` class needs to do is define a constructor, since constructors are not inherited.

There is no limit to the depth to which you can carry subclassing. For example, you could choose to write a class called `ColoredSquare` that is a subclass of the `Square` class. The `ColoredSquare` class would inherit the public methods from both `Square` and `RegularPolygon`. However, `ColoredSquare` would need to override the `draw()` method with an implementation that handles drawing in color.

Having defined the three classes `RegularPolygon`, `Square`, and `ColoredSquare`, it is correct to say that `RegularPolygon` and `Square` are superclasses of `ColoredSquare` and `ColoredSquare` and `Square` are subclasses of `RegularPolygon`. To describe a relationship between classes that extends through exactly one level of inheritance, you can use the terms *immediate superclass* and *immediate subclass*. For example, `Square` is an immediate subclass of `RegularPolygon`, while `ColoredSquare` is an immediate subclass of `Square`. By the same token, `RegularPolygon` is the immediate superclass of `Square`, while `Square` is the immediate superclass of `ColoredSquare`.

A class can have any number of subclasses or superclasses. However, a class can only have one immediate superclass. This constraint is enforced by the syntax of the `extends` clause; it can only specify the name of one superclass. This style of inheritance is called *single inheritance* ; it is different from the multiple inheritance scheme that is used in C++.

Every class in Java (except `Object`) has the class `Object` as its ultimate superclass. The class `Object` has no superclass. The subclass relationships between all of the Java classes can be drawn as a tree that has the `Object` class as its root. Another important difference between Java and C++ is that C++ does not have a class that is the ultimate superclass of all of its classes.

References Class Inheritance; Interfaces; Object

Abstract classes

If a class is declared with the `abstract` modifier, the class cannot be instantiated. This is different than C++, which has no way of explicitly specifying that a class cannot be instantiated. An `abstract` class is typically used to declare a common set of methods for a group of classes when there are no reasonable or useful implementations of the methods at that level of abstraction.

For example, the `java.lang` package includes classes called `Byte`, `Short`, `Integer`, `Long`, `Float`, and `Double`. These classes are subclasses of the `abstract` class `Number`, which declares the following methods: `byteValue()`, `shortValue()`, `intValue()`, `longValue()`, `floatValue()`, and `doubleValue()`. The purpose of these methods is to return the value of an object converted to the type implied by the method's name. Every subclass of `Number` implements all of these methods. The advantage of the abstraction is that it allows you to write code to extract whatever type of value you need from a `Number` object, without knowing the actual type of the underlying object.

Methods defined in an `abstract` class can be declared `abstract`. An `abstract` method is declared without any implementation; it must be overridden in a subclass to provide an implementation.

References Class Modifiers; Inner class modifiers; Local class modifiers; Method modifiers; Number

Final classes

If a class is declared with the `final` modifier, the class cannot be subclassed. Declaring a class `final` is useful if you need to ensure the exact properties and behavior of that class. Many of the classes in the `java.lang` package are declared `final` for that reason.

Methods defined in a non-`abstract` class can be declared `final`. A `final` method cannot be overridden by any subclasses of the class in which it appears.

References Class Modifiers; Inner class modifiers; Local class modifiers; Method modifiers

Interfaces

Java provides a construct called an interface to support certain multiple inheritance features that are desirable in an object-oriented language. An interface is similar to a class, in that an interface declaration can define both variables and methods. But unlike a class, an interface cannot provide implementations for its methods.

A class declaration can include an `implements` clause that specifies the name of an interface. When a class declaration specifies that it implements an interface, the class inherits all of the variables and methods declared in that interface. The class declaration must then provide implementations for all of the methods declared in the interface, unless the class is declared as an `abstract` class. Unlike the `extends` clause, which can only specify one class, the `implements` clause can specify any number of interfaces. Thus a class can implement an unlimited number of interfaces.

Interfaces are most useful for declaring that an otherwise unrelated set of classes have a common set of methods, without needing to provide a common implementation. For example, if you want to store a variety of objects in a database, you might want all of the those objects to have a common set of methods for storing and fetching. Since the fetch and store methods for each object need to be different, it is appropriate to declare these methods in an interface. Then any class that needs fetch and store methods can implement the interface.

Here is a simplistic example that illustrates such an interface:

```
public interface Db {
    void dbStore(Database d, Object key);
    Object dbFetch(Database d, Object key);
}
```

The `Db` interface declaration contains two methods, `dbStore()` and `dbFetch()`. Here is a partial class definition for a class that implements the `Db` interface:

```
class DbSquare extends Square implements Db {
    public void dbStore(Database d, Object key) {
        // Perform database operation to store Square
        ...
    }
    public Square dbFetch(Database d, Object key) {
        // Perform database operation to fetch Square
        ...
    }
    ...
}
```

The `DbSquare` class defines implementations for both of the methods declared in the `Db` interface. The point of this interface is that it provides a uniform way for unrelated objects to arrange to be stored in a database. The following code shows part of a class that encapsulates database operations:

```
class Database {
    ...
    public void store(Object o, Object key) {
        if (o instanceof Db)
          ((Db)o).dbStore(this, key);
    }
    ...
}
```

When the database is asked to store an object, it does so only if the object implements the `Db` interface, in which case it can call the `dbStore()` of the object.

References Interface Declarations

Inner Classes

Java 1.1 provides a new feature that allows programmers to encapsulate even more functionality within objects. With the addition of inner classes to the Java language, classes can be defined as members of other classes, just like variables and methods. Classes can also be defined within blocks of Java code, just like local variables. The ability to declare a class inside of another class allows you to encapsulate auxiliary classes inside of a class, thereby limiting access to the auxiliary classes. A class that is declared inside of another class may have access to the instance variables of the enclosing class; a class declared within a block may have access to the local variable and/or formal parameters of that block.

Nested top-level classes and interfaces

A *nested top-level class* or *interface* is declared as a `static` member of an enclosing top-level class or interface. The declaration of a nested top-level class uses the `static` modifier, so you may also see these classes called *static classes*. A nested interface is implicitly `static`, but you can declare it to be `static` to make it explicit. Nested top-level classes and interfaces are typically used to group related classes in a convenient way.

A nested top-level class or interface functions like a normal top-level class or interface, except that the name of the nested entity includes the name of the class in which it is defined. For example, consider the following declaration:

```
public class Queue {
    ...
    public static class EmptyQueueException extends Exception {
    }
    ...
}
```

Code that calls a method in `Queue` that throws an `EmptyQueueException` can catch that exception with a `try` statement like this:

```
try {
    ...
} catch (Queue.EmptyQueueException e) {
    ...
}
```

A nested top-level class cannot access the instance variables of its enclosing class. It also cannot call any non-`static` methods of the enclosing class without an explicit reference to an instance of that class. However, a nested top-level class can use any of the `static` variables and methods of its enclosing class without qualification.

Only top-level classes in Java can contain nested top-level classes. In other words, a `static` class can only be declared as a direct member of a class that is declared at the top level, directly as a member of a package. In addition, a nested top-level class cannot declare any `static` variables, `static` methods, or static initializers.

References Class Declarations; Methods; Nested Top-Level and Member Classes; Variables

Member classes

A *member class* is an inner class that is declared within an enclosing class without the `static` modifier.

Member classes are analogous to the other members of a class, namely the instance variables and methods. The code within a member class can refer to any of the variables and methods of its enclosing class, including `private` variables and methods.

Here is a partial definition of a `Queue` class that uses a member class:

```
public class Queue {
    private QueueNode queue;
    ...
    public Enumeration elements() {
        return new QueueEnumerator();
    }
    ...
    private class QueueEnumerator implements Enumeration {
        private QueueNode start, end;
        QueueEnumerator() {
            synchronized (Queue.this) {
                if (queue != null) {
                    start = queue.next;
                    end = queue;
                }
            }
        }
        public boolean hasMoreElements() {
            return start != null;
        }
        public synchronized Object nextElement() {
            ...
        }
    }
    private static class QueueNode {
        private Object obj;
        QueueNode next;
        QueueNode(Object obj) {
            this.obj = obj;
        }
        Object getObject() {
            return obj;
        }
    }
}
```

The `QueueEnumerator` class is a `private` member class that implements the `java.util.Enumeration` interface. The advantage of this approach is that the `QueueEnumerator` class can access the `private` instance variable `queue` of the enclosing `Queue` class. If `QueueEnumerator` were declared outside of the `Queue` class, this `queue` variable would need to be `public`, which would compromise the encapsulation of the `Queue` class. Using a member class that implements the `Enumeration` interface provides a means to offer controlled access to the data in a `Queue` without exposing the internal data structure of the class.

An instance of a member class has access to the instance variables of exactly one instance of its enclosing class. That instance of the enclosing class is called the *enclosing instance*. Thus, every `QueueEnumerator` object has exactly one `Queue` object that is its enclosing instance. To access an enclosing instance, you use the construct *ClassName*`.this`. The `QueueEnumerator` class uses this construct in the `synchronized` statement in its constructor to synchronize on its enclosing instance. This synchronization is necessary to ensure that the newly created `QueueEnumerator` object has exclusive access to the internal data of the `Queue` object.

The `Queue` class also contains a nested top-level, or `static`, class, `QueueNode`. However, this class is also declared `private`, so it is not accessible outside of `Queue`. The main difference between `QueueEnumerator` and `QueueNode` is that `QueueNode` does not need access to any instance data of `Queue`.

A member class cannot declare any `static` variables, `static` methods, `static` classes, or static initializers.

Although member classes are often declared `private`, they can also be `public` or `protected` or have the default accessibility. To refer to a class declared inside of another class from outside of that class, you prefix the class name with the names of the enclosing classes, separated by dots. For example, consider the following declaration:

```
public class A {
    public class B {
        public class C {
            ...
        }
        ...
    }
    ...
}
```

Outside of the class named `A`, you can refer to the class named `C` as `A.B.C`.

References Class Declarations; Field Expressions; Methods; Nested Top-Level and Member Classes; Variables

Local classes

A *local class* is an inner class that is declared inside of a block of Java code. A local class is only visible within the block in which it is declared, so it is analogous to a local variable. However, a local class can access the variables and methods of any enclosing classes. In addition, a local class can access any `final` local variables or method parameters that are in the scope of the block that declares the class.

Local classes are most often used for *adapter classes*. An adapter class is a class that implements a particular interface, so that another class can call a particular method in the adapter class when a certain event occurs. In other words, an adapter class is Java's way of implementing a "callback" mechanism. Adapter classes are commonly used with the new event-handling model required by the Java 1.1 AWT and by the JavaBeans API.

Here is an example of a local class functioning as an adapter class:

```
public class Z extends Applet {
    public void init() {
        final Button b = new Button("Press Me");
        add(b);
        class ButtonNotifier implements ActionListener {
            public void actionPerformed(ActionEvent e) {
                b.setLabel("Press Me Again");
                doIt();
            }
        }
        b.addActionListener(new ButtonNotifier());
    }
    ...
}
```

The above example is from an applet that has a `Button` in its user interface. To tell a `Button` object that you want to be notified when it is pressed, you pass an instance of an adapter class that implements the `ActionListener` interface to its `addActionListener()` method. A class that implements the `ActionListener` interface is required to implement the `actionPerformed()` method. When the `Button` is pressed, it calls the adapter object's `actionPerformed()` method. The main advantage of declaring the `ButtonNotifier` class in the method that creates the `Button` is that it puts all of the code related to creating and setting up the `Button` in one place.

As the preceding example shows, a local class can access local variables of the block in which it is declared. However, any local variables that are accessed by a local class must be declared `final`. A local class can also access method parameters and the exception parameter of a `catch` statement that are accessible within the scope of its block, as long as the parameter is declared `final`. The Java compiler complains if a local class uses a non-`final` local variable or parameter. The lifetime of a parameter or local variable is extended indefinitely, as long as there is an instance of a local class that refers to it.

References Blocks; Class Declarations; Local Classes; Local Variables; Method formal parameters; Methods; The try Statement; Variables

Anonymous classes

An *anonymous class* is a kind of local class that does not have a name and is declared inside of an allocation expression. As such, an anonymous class is a more concise declaration of a local class that combines the declaration of the class with its instantiation.

Here is how you can rewrite the previous adapter class example to use an anonymous class instead of a local class:

```
public class Z extends Applet {
    public void init() {
        final Button b = new Button("Press Me");
        add(b);
        b.addActionListener(new ActionListener () {
            public void actionPerformed(ActionEvent e) {
                b.setLabel("Press Me Again");
            }
        } );
    }
    ...
}
```

As you can see, an anonymous class is declared as part of an allocation expression. If the name after `new` is the name of an interface, as is the case in the preceding example, the anonymous class is an immediate subclass of `Object` that implements the given interface. If the name after `new` is the name of a class, the anonymous class is an immediate subclass of the named class.

Obviously, an anonymous class doesn't have a name. The other restriction on an anonymous class is it can't have any constructors other than the default constructor. Any constructor-like initialization must be done using an instance initializer. Other than these differences, anonymous classes function just like local classes.

References Allocation Expressions; Class Declarations; Instance Initializers; Object

Implementation of inner classes

It is possible to use inner classes without knowing anything about how they are implemented. However, a high-level understanding can help you comprehend the filenames that the compiler produces, and also some of the restrictions associated with inner classes. The implementation of inner classes is less than transparent in a number of ways, primarily because the Java virtual machine does not know about inner classes. Instead, the Java compiler implements inner classes by rewriting them in a form that does not use inner classes. The advantage of this approach is that the Java virtual machine does not require any new features to be able to run programs that use inner classes.

Since a class declared inside another class is rewritten by the compiler as an external class, the compiler must give it a name unique outside of the class in which it is declared. The unique name is formed by prefixing the name of the inner class with the name of the class in which it is declared and a dollar sign (`$`). Thus, when the `Queue` class is compiled, the Java compiler produces four *.class* files:

- *Queue.class*
- *Queue$EmptyQueueException.class*
- *Queue$QueueEnumerator.class*
- *Queue$QueueNode.class*

Because anonymous classes do not have names, the Java compiler gives each anonymous class a number for a name; the numbers start at 1. When the version of the `Z` applet that uses an anonymous class is compiled, the Java compiler produces two *.class* files:

- *Z.class*
- *Z$1.class*

In order to give an inner class access to the variables of its enclosing instance, the compiler adds a `private` variable to the inner class that references the enclosing instance. The compiler also inserts a formal parameter into each constructor of the inner class and passes the reference to the enclosing instance using this parameter. Therefore, the `QueueEnumerator` class is rewritten as follows:

```
class Queue$QueueEnumerator implements Enumeration {
    private Queue this$0;
    private QueueNode start, end;
    QueueEnumerator(Queue this$0) {
        this.this$0 = this$0;
        synchronized (this$0) {
            if (queue != null) {
                start = queue.next;
                end = queue;
            }
        }
    }
    ...
}
```

As you can see, the compiler rewrites all references to the enclosing instance as `this$0`. One implication of this implementation is that you cannot pass the enclosing instance as an argument to its superclass's constructor because `this$0` is not available until after the superclass's constructor returns.

PREVIOUS | HOME | NEXT
Lexical Scope of Declarations | BOOK INDEX | Class Declarations

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

7.2 Packages

A package is a group of classes. If a class is not declared as `public`, it can only be referenced by other classes in the same package. A class is specified as being part of a particular package by a `package` directive at the beginning of its compilation unit:

[Graphic: Figure from the text]

[Graphic: Figure from the text]

A `package` directive can only occur at the beginning of a compilation unit (ignoring comments and white space). If there is no `package` directive in a compilation unit, the compilation unit is part of the default package. A package is identified by its name. However, the default package has no name. Here are some examples of package directives:

```
package tools.text;
package COM.geomaker;
```

A class or interface definition can refer to class and interface definitions in a different package by qualifying the class or interface name with the package name and a period. For example, you can refer to the `Socket` class as follows:

```
java.net.Socket
```

However, if you attempt to use a non-`public` class or interface defined in another package, the Java compiler issues an error message.

An `import` directive, described in the next section, makes the class and interface definitions in another package available by their simple names. In other words, if you use an `import` directive, you do not have to qualify the names of the classes and interfaces in the package with the package name.

In Sun's implementation of Java, the name of the package for a given compilation unit is used to determine the directories that the Java interpreter searches to find the compiled Java code (i.e., the *.class* file) for the compilation unit. The Java interpreter uses a two-step process to find the compiled code for a class in a named package:

- The name of the package is converted into a relative path. Each identifier in the package name becomes the name of a directory in this relative path. (This scheme assumes that the Java interpreter is operating in an environment that supports a hierarchical file system.)

- The relative path is appended to the directories specified in the `CLASSPATH` environment variable and the resulting paths are searched for the *.class* file.

If the Java interpreter is searching for the compiled code for a class that is in the default package, it simply searches the directories specified in the `CLASSPATH` environment variable.

For example, say that the value of the `CLASSPATH` environment variable is as follows:[1]

```
\java\classes;.\;
```

In this case, the Java interpreter searches for the *.class* files for classes in the package named `COM.geomaker` in the following directories:

```
\java\classes\COM\geomaker
.\COM\geomaker
```

If a package name contains a Unicode character that cannot directly appear in a directory name, the character is represented in the directory name by an "at" sign (`@`) followed by one to four hexadecimal digits. For example, the package name:

```
COM.geomaker.hg\u00f8
```

becomes the relative path:

```
\COM\geomaker\hg@f8
```

[1] This example uses Windows syntax for directory names. The syntax for directory names is different in other environments. In particular, the character used to separate directory names varies in other environments.

Java classes can also be retrieved out of a *.zip* file if the file is specified as part of the CLASSPATH. For instance, the value of CLASSPATH could be set as follows:

```
\java\classes;\java\classes.zip;.\;
```

When the Java interpreter finds a *.zip* file in the CLASSPATH, it searches the *.zip* file for the appropriate *.class* file. The core classes in the Java API are supplied in a file that is typically named something like *jdk1.1/lib/classes.zip*. As of Java 1.1, you do not normally need to put that *.zip* file in CLASSPATH because the Java interpreter automatically puts *startDir/../classes.zip* on the end of CLASSPATH (where *startDir* is the directory that contains the interpreter's executable file).

The Java language specification defines a scheme for creating package names that should be globally unique. Since Internet domain names are globally unique, the idea is to incorporate them into package names. This is done by reversing the order of the components of the domain name, capitalizing the top-level component of the domain name, and using the result as a prefix for the descriptive portion of a package name. For example, if different organizations were to create packages that they all wanted to call opinion_poll, they could use this scheme to ensure global uniqueness. The resulting package names might be:

```
COM.cnn.opinion_poll
GOV.whitehouse.opinion_poll
EDU.syracuse.newhouse.opinion_poll
```

Package names that begin with an identifier that does not contain all uppercase letters are reserved for use as local package names. The one exception is package names that begin with the identifier java, which are reserved for packages that are part of the standard Java distribution.

References Class Declarations; Identifiers; Interface Declarations; The import Directive

Class

Name

Class

Synopsis

Class Name:

`java.lang.Class`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

`java.io.Seriablizable`

Availability:

JDK 1.0 or later

Description

As of Java 1.1, instances of the `Class` class are used as run-time descriptions of all Java data types, both reference types and primitive types. The `Class` class has also been greatly expanded in 1.1 to provide support for the Reflection API. Prior to 1.1, `Class` just provided run-time descriptions of reference types.

A `Class` object provides considerable information about the data type. You can use the `isPrimitive()` method to find out if a `Class` object describes a primitive type, while `isArray()` indicates if the object describes an array type. If a `Class` object describes a class or interface type, there are numerous methods that return information about the fields, methods, and constructors of the type. This information is returned as `java.lang.reflect.Field`, `java.lang.reflect.Method`, and `java.lang.reflect.Constructor` objects.

There are a number of ways that you can get a `Class` object for a particular data type:

- If you have an object, you can get the `Class` object that describes the class of that object by calling the object's `getClass()`method. Every class inherits this method from the `Object` class.

- As of Java 1.1, you can get the `Class` object that describes any Java type using the new class literal syntax. A class literal is simply the name of a type (a class name or a primitive type name) followed by a period and the `class` keyword. For example:

  ```
  Class s = String.class;
  Class i = int.class;
  Class v = java.util.Vector.class;
  ```

- In Java 1.0, you can get the `Class` object from the name of a data type using the `forName()` class method of `Class`. For example:

  ```
  Class v = Class.forName("java.util.Vector");
  ```

  This technique still works in Java 1.1, but it is more cumbersome (and less efficient) than using a class literal.

You can create an instance of a class using the `newInstance()` method of a `Class` object, if the class has a constructor that takes no arguments.

The `Class` class has no `public` constructors; it cannot be explicitly instantiated. `Class` objects are normally created by the `ClassLoader` class or a `ClassLoader` object.

Class Summary

```
public final class java.lang.Class extends java.lang.Object
                                   implements java.io.Serializable {
    // Class Methods
    public static native Class forName(String className);
    // Instance Methods
    public Class[] getClasses();                              // New in 1.1
    public native ClassLoader getClassLoader();
    public native Class getComponentType();                   // New in 1.1
    public Constructor
           getConstructor(Class[] parameterTypes);            // New in 1.1
    public Constructor[] getConstructors();                   // New in 1.1
    public Class[] getDeclaredClasses();                      // New in 1.1
    public Constructor
           getDeclaredConstructor(Class[] parameterTypes); // New in 1.1
    public Constructor[] getDeclaredConstructors();           // New in 1.1
```

```
    public Field getDeclaredField(String name);             // New in 1.1
    public Field[] getDeclaredFields();                     // New in 1.1
    public Method getDeclaredMethod(String name,
                  Class[] parameterTypes)                   // New in 1.1
    public Method[] getDeclaredMethods()                    // New in 1.1
    public Class getDeclaringClass();                       // New in 1.1
    public Field getField(String name);                     // New in 1.1
    public Field[] getFields();                             // New in 1.1
    public native Class[] getInterfaces();
    public Method getMethod(String name,
                  Class[] parameterTypes);                  // New in 1.1
    public Method[] getMethods();                           // New in 1.1
    public native int getModifiers();                       // New in 1.1
    public native String getName();
    public URL getResource(String name);                    // New in 1.1
    public InputStream getResourceAsStream(String name);   // New in 1.1
    public native Object[] getSigners();                    // New in 1.1
    public native Class getSuperclass();
    public native boolean isArray();                        // New in 1.1
    public native boolean isAssignableFrom(Class cls);     // New in 1.1
    public native boolean isInstance(Object obj);          // New in 1.1
    public native boolean isInterface();
    public native boolean isPrimitive();                    // New in 1.1
    public native Object newInstance();
    public String toString();
}
```

Class Methods

forName

`public static Class forName(String className) throws ClassNotFoundException`

Parameters

`className`

Name of a class qualified by the name of its package. If the class is defined inside of another class, all dots (`.`) that separate the top-level class name from the class to load must be changed to dollar signs ($) for the name to be recognized.

Returns

A `Class` object that describes the named class.

Throws

`ClassNotFoundException`

If the class cannot be loaded because it cannot be found.

Description

This method dynamically loads a class if it has not already been loaded. The method returns a `Class` object that describes the named class.

The most common use of `forName()` is for loading classes on the fly when an application wants to use classes it wasn't built with. For example, a web browser uses this technique. When a browser needs to load an applet, the browser calls `Class.forName()` for the applet. The method loads the class if it has not already been loaded and returns the `Class` object that encapsulates the class. The browser then creates an instance of the applet by calling the `Class` object's `newInstance()` method.

When a class is loaded using a `ClassLoader` object, any classes loaded at the instigation of that class are also loaded using the same `ClassLoader` object. This method implements that security policy by trying to find a `ClassLoader` object to load the named class. The method searches the stack for the most recently invoked method associated with a class that was loaded using a `ClassLoader` object. If such a class is found, the `ClassLoader` object associated with that class is used.

Instance Methods

getClasses

`public Class[] getClasses()`

Availability

New as of JDK 1.1

Returns

An array of `Class` objects that contains the `public` classes and interfaces that are members of this class.

Description

If this `Class` object represents a reference type, this method returns an array of `Class` objects that lists all of the `public` classes and interfaces that are members of this class or interface. The list includes `public` classes and interfaces that are inherited from superclasses and that are defined by this class or interface. If there are no `public` member classes or interfaces, or if this `Class` represents a primitive type, the method returns an array of length `0`.

As of Java 1.1.1, this method always returns an array of length `0`, no matter how many `public` member classes this class or interface actually declares.

getClassLoader

`public native ClassLoader getClassLoader()`

Returns

The `ClassLoader` object used to load this class or `null` if this class was not loaded with a `ClassLoader`.

Description

This method returns the `ClassLoader` object that was used to load this class. If this class was not loaded with a `ClassLoader`, `null` is returned.

This method is useful for making sure that a class gets loaded with the same class loader as was used for loading this `Class` object.

getComponentType

`public native Class getComponentType()`

Availability

New as of JDK 1.1

Returns

A `Class` object that describes the component type of this class if it is an array type.

Description

If this `Class` object represents an array type, this method returns a `Class` object that describes the component type of the array. If this `Class` does not represent an array type, the method returns `null`.

getConstructor

`public Constructor getConstructor(Class[] parameterTypes) throws NoSuchMethodException, SecurityException`

Availability

New as of JDK 1.1

Parameters

`parameterTypes`

An array of `Class` objects that describes the parameter types, in declared order, of the constructor.

Returns

A `Constructor` object that reflects the specified `public` constructor of this class.

Throws

`NoSuchMethodException`

If the specified constructor does not exist.

`SecurityException`

If the `checkMemberAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

If this `Class` object represents a class, this method returns a `Constructor` object that reflects the specified `public` constructor of this class. The constructor is located by searching all of the constructors of the class for a `public` constructor that has exactly the same formal parameters as specified. If this `Class` does not represent a class, the method returns `null`.

getConstructors

```
public Constructor[] getConstructors() throws SecurityException
```

Availability

New as of JDK 1.1

Returns

An array of `Constructor` objects that reflect the `public` constructors of this class.

Throws

`SecurityException`

If the `checkMemberAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

If this `Class` object represents a class, this method returns an array of `Constructor` objects that reflect the `public` constructors of this class. If there are no `public` constructors, or if this `Class` does not represent a class, the method returns an array of length `0`.

getDeclaredClasses

```
public Class[] getDeclaredClasses() throws SecurityException
```

Availability

New as of JDK 1.1

Returns

An array of `Class` objects that contains all of the declared classes and interfaces that are members of this class.

Throws

`SecurityException`

If the `checkMemberAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

If this `Class` object represents a reference type, this method returns an array of `Class` objects that lists all of the classes and interfaces that are members of this class or interface. The list includes `public`, `protected`, default access, and `private` classes and interfaces that are defined by this class or interface, but it excludes classes and interfaces inherited from superclasses. If there are no such member classes or interfaces, or if this `Class` represents a primitive type, the method returns an array of length `0`.

As of Java 1.1.1, this method always returns an array of length `0`, no matter how many member classes this class or interface declares.

getDeclaredConstructor

```
 public Constructor getDeclaredConstructor(Class[] parameterTypes) throws
NoSuchMethodException, SecurityException
```

Availability

New as of JDK 1.1

Parameters

`parameterTypes`

An array of `Class` objects that describes the parameter types, in declared order, of the constructor.

Returns

A `Constructor` object that reflects the specified declared constructor of this class.

Throws

`NoSuchMethodException`

If the specified constructor does not exist.

`SecurityException`

If the `checkMemberAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

If this `Class` object represents a class, this method returns a `Constructor` object that reflects the specified declared constructor of this class. The constructor is located by searching all of the constructors of the class for a `public`, `protected`, default access, or `private` constructor that has exactly the same formal parameters as specified. If this `Class` does not represent a class, the method returns `null`.

getDeclaredConstructors

`public Constructor[] getDeclaredConstructors() throws SecurityException`

Availability

New as of JDK 1.1

Returns

An array of `Constructor` objects that reflect the declared constructors of this class.

Throws

`SecurityException`

If the `checkMemberAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

If this `Class` object represents a class, this method returns an array of `Constructor` objects that reflect the `public`, `protected`, default access, and `private` constructors of this class. If there are no declared constructors, or if this `Class` does not represent a class, the method returns an array of length `0`.

getDeclaredField

`public Field getDeclaredField(String name) throws NoSuchFieldException, SecurityException`

Availability

New as of JDK 1.1

Parameters

`name`

The simple name of the field.

Returns

A `Field` object that reflects the specified declared field of this class.

Throws

`NoSuchFieldException`

If the specified field does not exist.

`SecurityException`

If the `checkMemberAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

If this `Class` object represents a class or interface, this method returns a `Field` object that reflects the specified declared field of this class. The field is located by searching all of the fields of the class (but not inherited fields) for a `public`, `protected`, default access, or `private` field that has the specified simple name. If this `Class` does not represent a class or interface, the method returns `null`.

getDeclaredFields

public Field[] getDeclaredFields() throws SecurityException

Availability

New as of JDK 1.1

Returns

An array of `Field` objects that reflect the declared fields of this class.

Throws

`SecurityException`

If the `checkMemberAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

If this `Class` object represents a class or interface, this method returns an array of `Field` objects that reflect the `public`, `protected`, default access, and `private` fields declared by this class, but excludes inherited fields. If there are no declared fields, or if this `Class` does not represent a class or interface, the method returns an array of length `0`.

This method does not reflect the implicit `length` field for array types. The methods of the class `Array` should be used to manipulate array types.

getDeclaredMethod

`public Method getDeclaredMethod(String name, Class[] parameterTypes) throws NoSuchMethodException, SecurityException`

Availability

New as of JDK 1.1

Parameters

`name`

The simple name of the method.

`parameterTypes`

An array of `Class` objects that describes the parameter types, in declared order, of the method.

Returns

A `Method` object that reflects the specified declared method of this class.

Throws

`NoSuchMethodException`

If the specified method does not exist.

`SecurityException`

If the `checkMemberAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

If this `Class` object represents a class or interface, this method returns a `Method` object that reflects the specified declared method of this class. The method is located by searching all of the methods of the class (but not inherited methods) for a `public`, `protected`, default access, or `private` method that has the specified simple name and exactly the same formal parameters as specified. If this `Class` does not represent a class or interface, the method returns `null`.

getDeclaredMethods

public Method[] getDeclaredMethods() throws SecurityException

Availability

New as of JDK 1.1

Returns

An array of `Method` objects that reflect the declared methods of this class.

Throws

`SecurityException`

If the `checkMemberAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

If this `Class` object represents a class or interface, this method returns an array of `Method` objects that reflect the `public`, `protected`, default access, and `private` methods declared by this class, but excludes inherited methods. If there are no declared methods, or if this `Class` does not represent a class or interface, the method returns an array of length `0`.

getDeclaringClass

`public Class getDeclaringClass()`

Availability

New as of JDK 1.1

Returns

A `Class` object that represents the declaring class if this class is a member of another class.

Description

If this `Class` object represents a class or interface that is a member of another class or interface, this method returns a `Class` object that describes the declaring class or interface. If this class or interface is not a member of another class or interface, or if it represents a primitive type, the method returns `null`.

getField

`public Field getField(String name) throws NoSuchFieldException, SecurityException`

Availability

New as of JDK 1.1

Parameters

`name`

The simple name of the field.

Returns

A `Field` object that reflects the specified `public` field of this class.

Throws

`NoSuchFieldException`

If the specified field does not exist.

`SecurityException`

If the `checkMemberAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

If this `Class` object represents a class or interface, this method returns a `Field` object that reflects the specified `public` field of this class. The field is located by searching all of the fields of the class, including any inherited fields, for a `public` field that has the specified simple name. If this `Class` does not represent a class or interface, the method returns `null`.

getFields

`public Field[] getFields() throws SecurityException`

Availability

New as of JDK 1.1

Returns

An array of `Field` objects that reflect the `public` fields of this class.

Throws

`SecurityException`

If the `checkMemberAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

If this `Class` object represents a class or interface, this method returns an array of `Field` objects that reflect the `public` fields declared by this class and any inherited `public` fields. If there are no `public` fields, or if this `Class` does not represent a class or interface, the method returns an array of length `0`.

This method does not reflect the implicit `length` field for array types. The methods of the class `Array` should be used to manipulate array types.

getInterfaces

public native Class[] getInterfaces()

Returns

An array of the interfaces implemented by this class or extended by this interface.

Description

If the `Class` object represents a class, this method returns an array that refers to all of the interfaces that the class implements. The order of the interfaces referred to in the array is the same as the order in the class declaration's `implements` clause. If the class does not implement any interfaces, the length of the returned array is 0.

If the object represents an interface, this method returns an array that refers to all of the interfaces that this interface extends. The interfaces occur in the order they appear in the interface declaration's `extends` clause. If the interface does not extend any interfaces, the length of the returned array is 0.

If the object represents a primitive or array type, the method returns an array of length 0.

getMethod

public Method getMethod(String name, Class[] parameterTypes) throws NoSuchMethodException, SecurityException

Availability

New as of JDK 1.1

Parameters

`name`

The simple name of the method.

`parameterTypes`

An array of `Class` objects that describes the parameter types, in declared order, of the method.

Returns

A `Method` object that reflects the specified `public` method of this class.

Throws

`NoSuchMethodException`

If the specified method does not exist.

`SecurityException`

If the `checkMemberAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

If this `Class` object represents a class or interface, this method returns a `Method` object that reflects the specified `public` method of this class. The method is located by searching all of the methods of the class, including any inherited methods, for a `public` method that has the specified simple name and exactly the same formal parameters as specified. If this `Class` does not represent a class or interface, the method returns `null`.

getMethods

`public Method[] getMethods() throws SecurityException`

Availability

New as of JDK 1.1

Returns

An array of `Method` objects that reflect the `public` methods of this class.

Throws

`SecurityException`

If the `checkMemberAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

If this `Class` object represents a class or interface, this method returns an array of `Method` objects that reflect the `public` methods declared by this class and any inherited `public` methods. If there are no `public` methods or if this `Class` doesn't represent a class or interface, the method returns an array of length `0`.

getModifiers

`public native int getModifiers()`

Availability

New as of JDK 1.1

Returns

An integer that represents the modifier keywords used to declare this class.

Description

If this `Class` object represents a class or interface, this method returns an integer value that represents the modifiers used to declare the class or interface. The `Modifier` class should be used to decode the returned value.

getName

`public native String getName()`

Returns

The fully qualified name of this class or interface.

Description

This method returns the fully qualified name of the type represented by this `Class` object.

If the object represents the class of an array, the method returns a `String` that contains as many left square brackets as there are dimensions in the array, followed by a code that indicates the type of element contained in the base array. Consider the following:

```
(new int [3][4][5]).getClass().getName()
```

This code returns "`[[[I`". The codes used to indicate the element type are as follows:

Code	**Type**
`[`	array
`B`	`byte`
`C`	`char`
`d`	`double`
`F`	`float`
`I`	`int`
`J`	`long`

`L` *fully_qualified_class_name*	class or interface
`S`	`short`
`Z`	`boolean`

getResource

`public URL getResource(String name)`

Availability

New as of JDK 1.1

Parameters

`name`

A resource name.

Returns

A `URL` object that is connected to the specified resource or `null` if the resource cannot be found.

Description

This method finds a resource with the given name for this `Class` object and returns a `URL` object that is connected to the resource. The rules for searching for a resource associated with a class are implemented by the `ClassLoader` for the class; this method simply calls the `getResource()` method of the `ClassLoader`. If this class does not have a `ClassLoader` (i.e., it is a system class), the method calls the `ClassLoader.getSystemResource()` method.

getResourceAsStream

`public InputStream getResourceAsStream(String name)`

Availability

New as of JDK 1.1

Parameters

`name`

A resource name.

Returns

An `InputStream` object that is connected to the specified resource or `null` if the resource cannot be found.

Description

This method finds a resource with the given name for this `Class` object and returns an `InputStream` object that is connected to the resource. The rules for searching for a resource associated with a class are implemented by the `ClassLoader` for the class; this method simply calls the `getResourceAsStream()` method of the `ClassLoader`. If this class does not have a `ClassLoader` (i.e., it is a system class), the method calls the `ClassLoader.getSystemResourceAsStream()` method.

getSigners

`public native Object[] getSigners()`

Availability

New as of JDK 1.1

Returns

An array of `Object`s that represents the signers of this class.

Description

This method returns an array of objects that represents the digital signatures for this class.

getSuperclass

`public native Class getSuperclass()`

Returns

The superclass of this class or `null` if there is no superclass.

Description

If the `Class` object represents a class other than `Object`, this method returns the `Class` object that represents its superclass. If the object represents an interface, the `Object` class, or a primitive type, the method returns `null`.

isArray

`public native boolean isArray()`

Availability

New as of JDK 1.1

Returns

`true` if this object describes an array type; otherwise `false`.

isAssignableFrom

`public native boolean isAssignableFrom(Class cls)`

Availability

New as of JDK 1.1

Parameters

`cls`

A `Class` object to be tested.

Returns

`true` if the type represented by `cls` is assignable to the type of this class: otherwise `false`.

Throws

`NullPointerException`

If `cls` is `null`.

Description

This method determines whether or not the type represented by `cls` is assignable to the type of this class. If this class represents a class, this class must be the same as `cls` or a superclass of `cls`. If this class represents an interface, this class must be the same as `cls` or a superinterface of `cls`. If this class represents a primitive type, this class must be the same as `cls`.

isInstance

`public native boolean isInstance(Object obj)`

Availability

New as of JDK 1.1

Parameters

`obj`

An `Object` to be tested.

Returns

true if obj can be cast to the reference type specified by this class; otherwise false.

Throws

NullPointerException

If obj is null.

Description

This method determines whether or not the object represented by obj can be cast to the type of this class object without causing a ClassCastException. This method is the dynamic equivalent of the instanceof operator.

isInterface

```
public native boolean isInterface()
```

Returns

true if this object describes an interface; otherwise false.

isPrimitive

```
public native boolean isPrimitive()
```

Availability

New as of JDK 1.1

Returns

true if this object describes a primitive type; otherwise false.

newInstance

```
 public native Object newInstance () throws InstantiationException,
IllegalAccessException
```

Returns

A reference to a new instance of this class.

Throws

`InstantiationException`

If the `Class` object represents an interface or an `abstract` class.

`IllegalAccessException`

If the class or an initializer is not accessible.

Description

This method creates a new instance of this class by performing these steps:

1. It creates a new object of the class represented by the `Class` object.
2. It calls the constructor for the class that takes no arguments.
3. It returns a reference to the initialized object.

The `newInstance()` method is useful for creating an instance of a class that has been dynamically loaded using the `forName()` method.

The reference returned by this method is usually cast to the type of object that is instantiated.

The `newInstance()` method can throw objects that are not instances of the classes it is declared to throw. If the constructor invoked by `newInstance()` throws an exception, the exception is thrown by `newInstance()` regardless of the class of the object.

toString

`public String toString()`

Returns

A `String` that contains the name of the class with either `'class'` or `'interface'` prepended as appropriate.

Overrides

`Object.toString()`

Description

This method returns a string representation of the `Class` object.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals()`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

ClassLoader; Class Declarations; Constructors; Exceptions; Interface Declarations; Methods; Nested Top-Level and Member Classes; Object; Object Creation; Reference Types; SecurityManager; Variables

Void

Name

Void

Synopsis

Class Name:

```
    java.lang.Void
```

Superclass:

```
    java.lang.Object
```

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability

New as of JDK 1.1

Description

The `Void` class is an uninstantiable wrapper for the primitive type `void`. The class contains simply a reference to the `Class` object that represents the primitive type `void`. The `Void` class is necessary as of JDK 1.1 to support the Reflection API and class literals.

Class Summary

```
public final class java.lang.Void extends java.lang.Object {
  // Constants
  public static final Class TYPE;
}
```

Constants

TYPE

public static final Class TYPE

The `Class` object that represents the primitive type `void`. It is always true that `Void.TYPE` == `void.class`.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

Byte; Character; Class; Double; Float; Integer; Long; Short

5.2 Lexical Scope of Declarations

The lexical scope of a declaration determines where the named entity is a valid identifier. Every declaration is associated with a lexical level that corresponds to one of the following Java constructs:

Package

> The names at this level include all of the non-nested, outer-level class and interface declarations in files that belong to the same package as the file that is being compiled. This level also includes non-nested, outer-level class and interface declarations that are declared `public` in other packages.

.java file

> The names at this level include all of the class and interface declarations in the file, as well as all of the classes and interfaces that are imported by the file. The names declared directly in a file are defined from the beginning to the end of the file. An `import` statement defines simple identifiers as synonyms for names that are only fully qualified with the name of a package. These synonyms for fully qualified names are defined from the `import` statement that defines them to the end of the file.

Class or interface declaration

> The names at this level include the names of methods, variables, and classes or interfaces that are declared directly in the class or interface declaration, as well as names inherited from superclasses or super interfaces. The names declared in a class or interface are defined throughout the class or interface.

Method declaration

> The names at this level include the formal parameters of the method. The formal parameters are defined throughout the method.

Block

> The names at this level include the local variables, local classes, and statement labels declared in the block. Statement labels are defined throughout a block, while local variables and classes are defined from their declaration to the end of the block.

A nested block or a `for` statement

> The names at this level include local variables declared in the initialization of the `for` statement or the local variables, classes, and statement labels declared in a nested block. Local variables declared in the initialization of a `for` statement are defined from their declaration to the end of the `for` statement. Statement labels are defined throughout a nested block, while local variables and classes are defined from their declaration to the end of the nested block.

These lexical levels correspond to nested constructs. When the Java compiler encounters a name in a program, it finds the declaration for that name by first looking in the lexical level where the name is encountered. If the compiler does not find the name in that lexical level, it searches progressively higher lexical levels until it finds the declaration. If all of the lexical levels are exhausted, the compiler issues an error message.

If, however, an identifier is qualified by a class or package name, the compiler only searches that lexical level for a declaration.

References Blocks; Class Declarations; Interface Declarations; Packages; Methods; The for Statement

PREVIOUS | HOME | NEXT
Naming Conventions | BOOK INDEX | Object-Orientation Java Style

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

6.2 Labeled Statements

Zero or more labels can appear before a statement:

[Graphic: Figure from the text]

A label is defined throughout the block in which it occurs. The names of labels are independent of all other kinds of names. In other words, if a label has the same name as a local variable, formal parameter, class, interface, field variable, or method, there is never any confusion or interaction between those names.[1] For example, the following code works even though it contains a label and formal parameter with the same name:

> [1] Prior to version 1.0.2, Java required labels to have names that did not conflict with the names of local variables or formal parameters.

```
public static void main (String[] argv) {
  argv:
    while (true) {
        System.out.println(argv[0]);
        if ( Math.random() >.4)
            break argv;
        System.out.println("Again");
    }
}
```

Labels are used to mark statements, but a labeled statement does not affect the order of execution when it is defined. The statement following the label is executed as if the label were not present. However, a label can be used in a `break` or `continue` statement to transfer control to a labeled statement. Unlike C/C++, Java does not have a `goto` statement.

References Identifiers; *Statement* 6; The break Statement; The continue Statement

PREVIOUS | HOME | NEXT
Blocks | BOOK INDEX | The Empty Statement

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

PREVIOUS

NEXT

6.6 The switch Statement

A `switch` statement selects a specially labeled statement in its block as the next statement to be executed, based on the value of an expression:

[Graphic: Figure from the text]

In Java, the type of the expression in parentheses must be `byte`, `char`, `short`, or `int`. This is unlike C/C++, which allows the type of a `switch` statement to be any integer type, including `long`.

The body of a `switch` statement must be a block. The top-level statements inside a `switch` may contain `case` labels. The expression following a `case` label must be a constant expression that is assignable to the type of the `switch` expression. No two `case` labels in a `switch` can contain the same value. At most one of the top-level statements in a `switch` can contain a `default` label.

A `switch` statement does the following:

- Evaluates the expression in parentheses. If the type of the expression is not `int`, the value produced by the expression is converted to `int`.
- Compares the value produced by the expression to the values in the `case` labels. Prior to comparison, the value in the `case` label is converted to `int` if it is not already `int`.
- If a `case` label is found that has the same value as the expression, that label's statement is the next statement to be executed.

- If no `case` label is found with the same value as the expression, and a statement in the block has a `default` label, that statement is the next one to be executed.

- If there is no statement in the block that has a `default` label, the statement after the `switch` statement is the next statement to be executed.

Here's an example of a `switch` statement:

```
switch (rc) {
  case 1:
    msg = "Syntax error";
    break;
  case 2:
    msg = "Undefined variable";
    break;
  default:
    msg = "Unknown error";
    break;
}
```

After the `switch` statement has transferred control to a `case`-labeled statement, statements are executed sequentially in the normal manner. Any `case` labels and the `default` label have no further effect on the flow of control. If no statement inside the block alters the flow of control, each statement is executed in sequence with control flowing past each `case` label and out the bottom of the block. The following example illustrates this behavior:

```
void doInNTimes(int n){
    switch (n > 5 ? 5 : n) {
      case 5:
        doIt();
      case 4:
        doIt();
      case 3:
        doIt();
      case 2:
        doIt();
      case 1:
        doIt();
    }
}
```

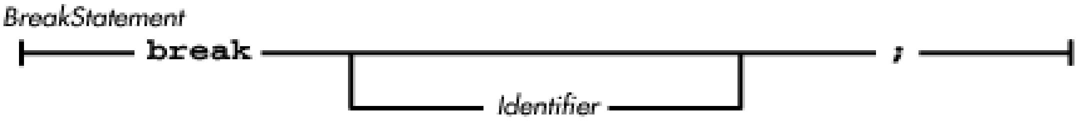
The above method calls the `doIt()` method up to 5 times.

To prevent control from flowing through `case` labels, it is common to end each `case` with a flow-altering statement such as a `break` statement. Other statements used for this purpose include the `continue` statement and the `return` statement.

References Constant Expressions; *Expression* 4; Identifiers; Integer types; Local Classes; Local Variables; *Statement* 6; The break Statement; The continue Statement; The return Statement

6.8 The break Statement

A break statement transfers control out of an enclosing statement:



If a break statement does not contain an identifier, the statement attempts to transfer control to the statement that follows the innermost enclosing while, for, do, or switch statement. The Java compiler issues an error message if a break statement without an identifier occurs without an enclosing while, for, do, or switch statement. Here is an example of a break statement that contains no identifier:

```
while (true) {
    c = in.read();
    if (Character.isSpace(c)
        break;
    s += (char)c;
}
```

In this example, the break statement is used to exit from the while loop.

The innermost while, for, do, or switch statement that encloses the break statement must be in the immediately enclosing method or initializer block. In other words, a break statement cannot be used to leave a method or initializer block. The break statement in the following example is used incorrectly and generates an error:

```
while (true) {
    class X {
        void doIt() {
            break;
        }
    }
    new X().doIt();
}
```

If a break statement contains an identifier, the identifier must be defined as the label of an enclosing statement. A break statement that contains an identifier attempts to transfer control to the statement that immediately follows the statement labeled with that identifier. Here's an example of a break statement that contains an identifier:

```
foo:{
    doIt();
    if (n > 4) break foo;
    doIt();
}
```

In this example, the break statement transfers control to the statement following the block labeled foo.

The label used in a break statement must be in the immediately enclosing method or initializer block. The break statement in the following example is used incorrectly and generates an error:

```
foo: {
    class X {
        void doIt() {
            break foo;
        }
    }
    new X().doIt();
}
```

The statement to which a break statement attempts to transfer control is called the *target statement*. If a break statement occurs inside a try statement, control may not immediately transfer to the target statement. If a try statement has a finally clause, the finally block is executed before control leaves the try statement for any reason. This means that if a break statement occurs inside a try statement (but not in its finally block) and the target statement is outside of the try statement, the finally block is executed first, before the control transfer can take place.

If the finally block contains a break, continue, return, or throw statement, the pending control transfer for the previously executed break statement is forgotten. Instead, control is transferred to the target of the break, continue, return, or throw statement in the finally block.

If the `finally` block does not contain a `break`, `continue`, `return`, or `throw` statement, the pending control transfer happens after the `finally` block is done executing, unless the target statement is enclosed by another `try` statement. If there is another enclosing `try` statement and it has a `finally` clause, that `finally` block is also executed before the control transfer can take place. Execution proceeds in this manner until the target statement of the `break` is executed.

Here is an example that illustrates a simple scenario:

```
ll:{
    try {
        f = new FileInputStream(fname);
        i = f.read();
        if (i != ' ')
            break ll;
        i = f.read();
    } catch (IOException e) {
        System.out.println("Got an IO Exception!");
        break ll;
    } finally {
        f.close();           // Always executed
    }
    // Only reached if we don't break out of the try
    System.out.println("No breaks");
}
```

In this example, a `break` statement is executed if one of two things happens. First, if an `IOException` is thrown, the `catch` clause prints a message and then executes a `break` statement. Otherwise, if the first call to `read()` does not return a space, a `break` statement is executed. In either case, the `finally` clause is executed before control is transferred to the statement following the statement labeled with `ll`.

References Identifiers; Labeled Statements; The continue Statement; The do Statement; The for Statement; The return Statement; The throw Statement; The try Statement; The while Statement

6.9 The continue Statement

A continue statement stops the current iteration of an iteration statement and transfers control to the start of the next iteration:

[Graphic: Figure from the text]

A continue statement must occur within a while, for, or do statement or the compiler issues an error message.

If a continue statement does not contain an identifier, the statement stops the current iteration in the innermost enclosing while, for, or do statement and attempts to transfer control to the start of the next iteration. This means that in a while or do statement, the continue statement transfers control to just after the contained statement of the while or do statement. In a for statement, the continue statement transfers control to the increment portion of the for statement. Here is an example of a continue statement that contains no identifier:

```
public static void main (String[] argv) {
    for (int i=0; i<=15; i++) {
        System.out.println(i);
        if ( (i&1) == 0 )
            continue;
        System.out.println("That's odd");
    }
}
```

The above example outputs the numbers 0 through 15, printing "That's odd" after each odd number.

The innermost `while`, `for`, `do`, or `switch` statement that encloses the `continue` statement must be in the immediately enclosing method or initializer block. The `continue` statement in the following example is used incorrectly and generates an error:

```
while (true) {
    class X {
        void doIt() {
            continue;
        }
    }
    new X().doIt();
}
```

If a `continue` statement contains an identifier, the identifier must be defined as the label of an enclosing `while`, `for`, or `do` statement. A `continue` statement that contains an identifier stops the current iteration of the labeled iteration statement and attempts to transfer control to the start of the next iteration of that loop. Here is an example of a `continue` statement that contains an identifier:

```
public boolean search(int x, int a[][]) {
    int count = 0;
  top:
    for (int i=0; i<a.length; i++) {
        int b[] = a[i];
        for (int j=0; j < b.length; j++) {
            if (x == b[j])
                return true;
            if ( x < b[j])
                continue top;
        } // for j
        count++;
        if (count > 100)
            return false;
    } // for i
    return false;
} // search()
```

The above method searches an array of arrays of integers for a specified value. The method assumes that the values in the sub-arrays are in descending order. The method gives up after checking 100 values.

The label used in a `continue` statement must be in the immediately enclosing method or initializer block.

The statement to which a `continue` statement attempts to transfer control is called the target statement. If a `continue` statement occurs inside a `try` statement, control may not immediately transfer to the target statement. If a `try` statement has a `finally` clause, the `finally` block is executed before control leaves the `try` statement for any reason. This means that if a `continue` statement occurs inside a `try` statement (but not in its `finally` block) and the target statement is outside of the `try` statement, the `finally` block is executed first, before the control transfer can take place.

If the `finally` block contains a `break`, `continue`, `return`, or `throw` statement, the pending control transfer for the previously executed `continue` statement is forgotten. Instead, control is transferred to the target of the `break`, `continue`, `return`, or `throw` statement in the `finally` block.

If the `finally` block does not contain a `break`, `continue`, `return`, or `throw` statement, the pending control transfer happens after the `finally` block is done executing, unless the target statement is enclosed by another `try` statement. If there is another enclosing `try` statement and it has a `finally` clause, that `finally` block is also executed before the control transfer can take place. Execution proceeds in this manner until the target statement of the `continue` is executed.

References Identifiers; Labeled Statements; The break Statement; The do Statement; The for Statement; The return Statement; The throw Statement; The try Statement; The while Statement

6.10 The return Statement

A `return` statement returns control of the current method or constructor to the caller:

[Graphic: Figure from the text]

If a `return` statement does not contain an expression, the statement must be in a method declared with the `void` return type or in a constructor. Otherwise, the compiler issues an error message. When a `return` statement does not contain an expression, the statement simply attempts to transfer control back to the method or constructor that invoked the current method or constructor.

If a `return` statement contains an expression, it must be in a method that returns a value or the compiler issues an error message. The type of the expression must be assignment-compatible with the declared return type of the method. The `return` statement attempts to transfer control back to the method or constructor that invoked the current method. The value produced by the expression is the return value of the current method.

Here's an example of a `return` statement:

```
int doubleIt (int k) {
    return k*2;
}
```

If a `return` statement occurs inside a `try` statement, control may not immediately transfer to the invoking method or constructor. If a `try` statement has a `finally` clause, the `finally` block is executed before control leaves the `try` statement for any reason. This means that if a `return` statement occurs inside a `try` statement (but not in its `finally` block), the `finally` block is executed first, before the control transfer can take place.

If the `finally` block contains a `break`, `continue`, `return`, or `throw` statement, the pending control transfer for the previously executed `return` statement is forgotten. Instead, control is transferred to the target of the `break`, `continue`, `return`, or `throw` statement in the `finally` block.

If the `finally` block does not contain a `break`, `continue`, `return`, or `throw` statement, the pending control transfer happens after the `finally` block is done executing, unless there is another enclosing `try` statement. If there is such a `try` statement and it has a `finally` clause, that `finally` block is also executed before the control transfer can take place. Execution proceeds in this manner until control is transferred to the invoking method or constructor.

References Constructors; *Expression* 4; Identifiers; Methods; The break Statement; The continue Statement; The throw Statement; The try Statement

6.11 The throw Statement

A `throw` statement is used to cause an exception to be thrown:

[Graphic: Figure from the text]

The expression in a `throw` statement must produce a reference to an object that is an instance of the `Throwable` class or one of its subclasses. Otherwise, the compiler issues an error message. You typically want the expression in a `throw` statement to produce an object that is an instance of a subclass of the `Exception` class.

Here is an example of a `throw` statement:

```
throw new ProtocolException();
```

A `throw` statement causes normal program execution to stop. Control is immediately transferred to the innermost enclosing `try` statement in the search for a `catch` clause that can handle the exception. If the innermost `try` statement cannot handle the exception, the exception propagates up through enclosing statements in the current method. If the current method does not contain a `try` statement that can handle the exception, the exception propagates up to the invoking method. If this method does not contain an appropriate `try` statement, the exception propagates up again, and so on. Finally, if no `try` statement is found to handle the exception, the currently running thread terminates. The termination of a thread is described in Stopping a thread.

As an exception propagates through enclosing `try` statements, any `finally` blocks associated with those `try` statements are executed until the exception is caught. If a `finally` block contains a `break`, continue, return, or throw statement, the pending control transfer initiated by the throw statement is forgotten. Instead, control is transferred to the target of the break, continue, return, or throw statement in the finally block.

References Exception Handling 9; *Expression* 4; The break Statement; The continue Statement; The return Statement; The try Statement; Throwable

PREVIOUS | HOME | NEXT
The return Statement | BOOK INDEX | The try Statement

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

6.12 The try Statement

A `try` statement provides a way to catch exceptions and execute clean-up code for a block:

[Graphic: Figure from the text]

[Graphic: Figure from the text]

A `try` statement contains a block of code to be executed. A `try` statement can have any number of optional `catch` clauses; these clauses act as exception handlers for the `try` block. A `try` statement can also have a `finally` clause. If present, the `finally` block is always executed before control leaves the `try` statement, so it is a good place to supply clean-up code for the `try` block. Note that a `try` statement must have either a `catch` clause or a `finally` clause.

Here is an example of a `try` statement that includes a `catch` clause and a `finally` clause:

```
try {
    out.write(b);
} catch (IOException e) {
    System.out.println("Output Error");
} finally {
    out.close();
}
```

If `out.write()` throws an `IOException`, the exception is caught by the `catch` clause. Regardless of whether `out.write()` returns normally or throws an exception, the `finally` block is executed, which ensures that `out.close()` is always called.

A `try` statement begins by executing the block that follows the keyword `try`. If an exception is thrown from within the `try` block and the `try` statement has any `catch` clauses, those clauses are searched in order for one that can handle the exception. A `catch` clause can handle an exception if the *ClassOrInterfaceName* specified in the clause is the same class as or a superclass of the object specified in the `throw` statement that caused the exception. The *ClassOrInterfaceName* specified in a catch clause must be `Throwable` or be one of its subclasses. If a `catch` clause handles an exception, that `catch` block is executed.

If an exception is thrown from within the `try` block and the `try` statement does not have any `catch` clauses that can handle the exception, the exception propagates up to the next enclosing `try` statement. An exception also propagates up if it is thrown from within a `catch`block in a `try` statement.

The identifier specified in parentheses for the `catch` clause is defined as a local variable within the `catch` block. The local variable is initialized to refer to the thrown object, in a manner that is similar to the way it which formal parameters for a method are handled. This means that an identifier in a `catch` clause cannot have the same name as a local variable or formal parameter that is defined in an enclosing block. If the `catch` parameter is declared as `final`, any assignment to that parameter in the `catch` block generates an error. The syntax for specifying `final catch` parameters is not supported prior to Java 1.1.

Any `catch` clauses in a `try` statement are checked in sequence to see if they can handle a given exception. Thus, the order in which `catch` clauses appear is important. In essence, more specific `catch` clauses should appear before more general `catch` clauses. Figure 6.1 shows the inheritance hierarchy for a few of the classes of objects that can be thrown in Java.

Figure 6.1: Some exception classes in Java

[Graphic: Figure 6-1]

Based on the classes shown in Figure 6.1, consider the following example:

```
try {
    System.out.write(b);
} catch (InterruptedIOException e) { ...
} catch (IOException e) { ...
} catch (Exception e) { ...
}
```

The `catch` clauses in this example appear in order from most specific to least specific. That means that if an `InterruptedIOException` were thrown, it would be caught by the first `catch` clause. Similarly, an `IOException` would be caught by the second `catch` clause and an `Exception` would be caught by the third clause. If, however, the `catch` clause for `Exception` appeared first, neither of the other `catch` clauses would ever be executed because the `catch` clause for `Exception` would catch all of the exceptions.

If a `try` statement includes a `finally` clause, the `finally` block is always executed before control leaves the `try` statement. There are two different ways that control can leave a `try` statement:

- The `try` statement completes normally. Normal completion occurs when all of the statements in the `try` block have been executed, so that control falls out of the bottom of the `try` block. Normal completion can also occur when an exception is thrown in the `try` block, as long as the exception is handled by a `catch` clause in the `try` statement.

- The `try` statement completes abruptly, due to an attempted control transfer out of the `try` block. A `break`, `continue`, or `return` statement in the `try` block causes an abrupt completion. In addition, abrupt completion can occur when an exception occurs and is not handled by a `catch` clause in the `try` statement, since the exception propagates out of the `try` block.

If a `try` statement completes normally and it does not have a `finally` clause, the statement following the `try` statement is the next statement to be executed. However, if the `try` statement does have a `finally` clause, the `finally` block is executed first, before control can be transferred to the statement following the `try` statement. If the `finally` block contains a `break`, `continue`, `return`, or `throw` statement, the pending control transfer is forgotten and control is instead transferred to the target of the `break`, `continue`, `return`, or `throw` statement in the `finally` block.

If a `try` statement completes abruptly and it does not have a `finally` clause, the control transfer out of the `try` block takes place immediately. However, if the `try` statement does have a `finally` clause, the `finally` block is executed first, before the control transfer can take place. If the `finally` block contains a `break`, `continue`, `return`, or `throw` statement, the pending control transfer is forgotten and control is instead transferred to the target of the `break`, `continue`, `return`, or `throw` statement in the `finally` block.

References Blocks; Exception Handling 9; *Expression* 4; Identifiers; The break Statement; The continue Statement; The return Statement; The throw Statement; Throwable *Type* 3;

7.3 The import Directive

You can refer to classes and interfaces defined in a particular package by qualifying their names with the package name and a period. For example, you can refer to the `Socket` class as `java.net.Socket`. Using this notation, you could write a declaration like the following:

```
java.net.Socket s = new java.net.Socket();
```

This declaration is rather verbose. As you can imagine, it would quickly become cumbersome to refer to classes this way in all of your programs.

The `import` directive provides an alternative to prefixing the names of classes and interfaces defined in particular packages with their package names. An `import` directive makes definitions from another package available by their simple names:

[Graphic: Figure from the text]

An `import` directive can only occur after the `package` directive in a compilation unit (if there is one) and before any class or interface declarations.

An `import` directive with an identifier after the package name defines that identifier to have the same meaning as the fully qualified class or interface name. When an identifier is defined using an `import` directive, the definition exists only from the `import` directive that defines it to the end of the compilation unit.

For example, you could use the following `import` directive:

```
import java.net.Socket;
```

Now the identifier `Socket` is defined to mean `java.net.Socket`. With the above `import` directive at the beginning of a compilation unit, you can rewrite the previous declaration as follows:

```
Socket s = new Socket();
```

If more than one `import` directive provides a definition for the same identifier, the compiler issues an error message.

An `import` directive can also be used to define an identifier as a synonym for the fully qualified name of a class that is declared inside of another class. For example, consider the following class declaration:

```
package COM.geomaker;
...
public class z {
    ...
    class zz {
        ...
    }
}
```

A class in another file can refer to the class `COM.geomaker.z.zz` as just `zz` if the file contains the following `import` directive:

```
import COM.geomaker.z.zz;
```

An `import` directive with an asterisk (`*`) after the package name tells the compiler to search the specified package when it cannot find a definition for an identifier. In other words, this type of `import` directive makes all of the classes and interfaces in the package available by their simple names. Here's an example of such an `import` directive:

```
import java.awt.*;
```

When the compiler is searching packages specified by this type of `import` directive, it issues an error message if it finds the same name defined in two different packages.

Every package in Java is considered separate and distinct, even if the name of a package begins with the name of another package. For example, the package `java.awt` is separate and distinct from the package `java.awt.image`. Even though the names imply a parent-child relationship, Java recognizes no such relationship between packages. Consider the following directive:

```
import java.awt.*;
```

This tells the Java compiler to search the `java.awt` package for class and interface names; it does not, however, tell the compiler to search `java.awt.image` for such names. For that to happen, a compilation unit must also include the following directive:

```
import Java.awt.image.*;
```

It is important to understand that an `import` directive does not cause the Java compiler to read any class or interface definitions. An `import` directive simply defines an identifier as a synonym for a fully qualified class or interface name or directs the compiler to search a package when it needs to find a definition. The compiler only reads a class or interface definition when its finds an actual reference to the class or interface.

References Compilation Units; Identifiers; Packages

PREVIOUS | HOME | NEXT
Packages | BOOK INDEX | Documentation Comments

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

7.4 Documentation Comments

Documentation comments are used to create stand-alone documentation for classes. Documentation comments are processed into Web pages by the *javadoc* program that is part of Sun's Java Development Kit (JDK). The *javadoc* program and the way that it processes *.java* files into Web pages is fully described in the documentation for *javadoc* provided by Sun. The remainder of this section describes the special formatting information that can be embedded in documentation comments.

Documentation comments are comments that begin with `/**`. If a documentation comment immediately precedes the declaration of a class, interface, method, or field variable, it is assumed to describe that class, interface, method, or field variable.

HTML tags can be included in a documentation comment; they are passed directly to the generated Web page. In addition to passing HTML tags, *javadoc* recognizes special tags that start with an "at" sign (@). These tags must appear as the first word on a line in order to be recognized. Here is an example of a documentation comment that includes these special *javadoc* tags:

```
/**
 * RomanNumeral is a class similar to Integer, except that
 * it uses roman numerals for its string based
 * representation.  It only represents positive numbers.
 *
 * @see        Integer
 * @see        Number
 * @see        Float
 * @see        Double
 * @version    1.1, 9/27/96
 * @author     Mark Grand
 */
```

Here are the special documentation comment tags recognized by *javadoc* :

`@author author-name`

Formats the given author name. This tag can only be used in a class or interface documentation comment.

`@exception name description`

Formats the given exception name and its description in the throws section of a method description. The name should be the fully qualified class name of the exception. This tag can only be used in a method documentation comment.

`@param name description`

Formats the given parameter name and its description in the parameters section of a method description. This tag can only be used in a method documentation comment.

`@return description`

Formats the given description in the returns section of a method description. This tag can only be used in a method documentation comment.

`@see classname`

Generates a hypertext link to the specified class. The class name may be qualified by its package name.

`@see classname#method-name`

Generates a hypertext link to the specified method in the specified class. The class name may be qualified by its package name.

`@version text`

Formats the given text as version information. This tag can only be used in a class or interface documentation comment.

`@deprecated`

Indicates that a class, method, or variable is deprecated, which means that it has been superceded by a newer, preferred class, method, or variable. Deprecated features should not be used in new Java programs. In addition, you should try to update existing code so that it does not rely on deprecated features. While the deprecated features in Java 1.1 are still supported, there is no guarantee that they will be supported in future releases. The `@deprecated` tag is new as of Java 1.1.

The documentation comment that immediately precedes a declaration is associated with the declaration. If two comments precede a declaration, only the one immediately preceding the declaration is processed by *javadoc*. The first comment is not considered to be associated with a declaration, so it is ignored. If there is anything but white space between a documentation comment and a declaration, the documentation comment is not considered to be associated with the declaration.

References Comments

PREVIOUS HOME NEXT
The import Directive BOOK INDEX Applications

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

7.5 Applications

For a Java program to run as an application, it must have at least one `public` class that contains a `public static` method called `main()` that takes exactly one parameter, an array of `String` objects.

Here is a very simple sample application that outputs "Hello World!" and then exits:

```
class DoIt {
    public static void main  (String argv[]){
         System.out.println ("Hello World!");
    }
}
```

The `main()` method must be `public` so that the Java virtual machine can find it. If the method is not `public`, its name is not included in the compiler's output. The system does not create any objects prior to the start of the application's `main()` method, so the `main()` method must be `static` because it cannot be associated with an object.

If an application has a graphical user interface, then it typically creates a `java.awt.Frame` object in `main()`. The `Frame` object acts as the top-level window for the application.

In Sun's implementation of Java, you run a Java application by running the Java interpreter with a command-line argument that specifies the name of the class that contains the `main()`. The name of the Java interpreter is *java*. Here's the command-line for our sample application:

```
C:\> java DoIt
```

The capitalization of the class name on the command line must match the capitalization of the class name within the program. If the class is part of a named package, the name of the class must be qualified with the package name. For example, if you have a package called `COM.geomaker` and it contains the class called DoIt, you would use the following command to run the application:

```
C:\> java COM.geomaker.DoIt
```

Any additional information that you provide on the command line is passed to the application as command line arguments. These arguments are passed to the application using the String array passed to main(). The number of elements in the array is equal to the number of arguments passed to the application. If there are no arguments to the application, the length of the array passed to main() is zero.

References Methods; Packages

7.6 Applets

A Java applet must be run from within another program, called a host application. At this point, most host applications are Web browsers. The interaction between an applet and its host application is rather involved.

From the viewpoint of an applet, the interaction involves defining a subclass of the `java.applet.Applet` class. The `Applet` class defines a number of methods that control the applet. A subclass of `Applet` overrides one or more of the methods:

`init()`

> The `init()` method is called to initialize the applet. Most initialization of an applet is done here instead of in a constructor because the constructor may be called before the hosting program is ready to provide all of the services needed for initialization.

`start()`

> The `start()` method is called in a separate thread to tell the applet to start doing whatever it is supposed to do.

`paint()`

> The `paint()` method is called at unpredictable times to draw the applet onto the screen.

`stop()`

> The `stop()` method is called to tell the applet to stop doing whatever it does.

`destroy()`

The `destroy()` method is called to tell the applet to release any resources that it holds.

From the viewpoint of the host application, the interaction typically follows a standard sequence of events. The host application usually does the following:

1. Installs a `SecurityManager` object to implement a security policy.
2. Creates a `ClassLoader` object to load the applet.
3. Loads the applet and calls its default constructor.
4. Passes an `AppletStub` object to the applet's `setStub()` method.
5. Calls the applet's `init()` method in a separate thread.
6. Marks the applet as active.
7. Starts a new thread to run the applet's `start()` method.
8. Calls the applet's `show()` method, which makes the applet visible and causes the applet's `paint()` method to be called for the first time.
9. Calls the applet's `paint()` method whenever the applet needs to be refreshed.
10. Calls the applet's `start()` and `stop()` methods when the host wants the applet to start or stop. These methods are typically called when the applet is exposed or hidden.
11. Calls the applet's `hide()` method followed by its `destroy()` method when the host wants to shut down the applet.

Embedding an Applet in a Web Page

Web pages are written in a language called HTML. This explanation of how to embed an applet in a Web page assumes that you have some knowledge of basic HTML. An applet is embedded in a Web page using an `<applet>` tag. A minimal `<applet>` tag looks as follows:

```
<applet code=Clock height=300 width=350>
</applet>
```

The `code` attribute of this sample `<applet>` tag specifies that the applet to be run is a class named `Clock`. The `width` and `height` attributes specify that the applet should be given a screen area that is

300 pixels high and 350 pixels wide.

The following list shows all of the attributes that can be specified in an `<applet>` tag. The attributes should be specified in the order in which they are listed. The `code`, `height`, and `width` attributes are required in an `<applet>` tag; the other attributes are optional:

`codebase`

> The `codebase` attribute should specify a URL that identifies the directory used to find the *.class* files needed for the applet. Files for classes that belong to the default package should be in this directory. Files for classes that belong to named packages should be in subdirectories of this directory, where the relative path is specified by individual identifiers in the package name. If `codebase` is not specified, the `<applet>` tag uses the directory that contains the HTML file as a default.

`code`

> The `code` attribute specifies the name of the class that implements the applet. If the applet is part of a named package, you must specify the fully qualified class name. So, if the name of the class is `DataPlot` and it is part of a package called `COM.geomaker.graph`, the value of the `code` attribute should be:

```
code=COM.geomaker.graph.DataPlot.class
```

> The browser locates the compiled code for the class by appending *.class* to the filename and searching the directory specified by the base URL for the document.

`object`

> The `object` attribute specifies the name of a file that contains a serialized representation of an applet. If this attribute is specified, the applet is created by deserialization, rather than by calling its default constructor. The serialization is assumed to have occurred after the applet's `init()` method has been invoked, so the `start()` method is called instead of the `init()` method. Any attributes specified when the applet was serialized are not restored; the applet sees the attributes specified for this invocation.

> The `object` attribute is new as of Java 1.1. An `<applet>` tag must include either the `code` attribute or the `object` attribute, but it cannot include both.

`archive`

> The `archive` attribute specifies a list of one or more archives that contain classes or other resources for an applet. Archives can be JAR or ZIP files. If this attribute is specified, the

resources in the archives are loaded before the applet is run. If multiple archives are listed, they should be separated by commas. The `archive` attribute is new for Java 1.1.

`alt`

The `alt` attribute specifies the text that should be displayed by Web browsers that understand the `<applet>` tag but cannot run Java applets. If the text contains space characters, it should be enclosed in quotation marks.

`name`

The `name` attribute specifies a name for a particular instance of an applet. An applet can get a reference to another applet on the same Web page using the `getApplet()` method.

`width`

The `width` attribute specifies the width of the applet in pixels.

`height`

The `height` attribute specifies the height of the applet in pixels.

`align`

The `align` attribute specifies the positioning of the applet. The possible values are: `left`, `right`, `top`, `texttop`, `middle`, `absmiddle`, `baseline`, `bottom`, or `absbottom`.

`vspace`

The `vspace` attribute specifies the amount of vertical space above and below the applet in pixels.

`hspace`

The `hspace` attribute specifies the amount of horizontal space to the left and right of the applet in pixels.

Applet-specific parameters can be provided to an applet using `<param>` tags inside the `<applet>` tag. A `<param>` tag must specify `name` and `value` attributes. For example:

```
<param name=speed value=65>
```

If a Web browser does not support the `<applet>` tag, it ignores the tag and simply displays any HTML content provided inside the tag. However, if the browser understands the `<applet>` tag, this HTML content is ignored. This means that you can provide HTML content inside an `<applet>` tag to inform users of non-Java-enabled browsers about what they are missing.

Here is an example that combines all of these elements:

```
<applet code=Compass height=400 width=300>
<param name=direction value=north>
<param name=speed value=65>
<p>
<i>If you can see this message, your Web browser is not Java enabled.
There is a Java applet on this Web page that you are not seeing.</i>
<p>
</applet>
```

If a non-Java-enabled browser is used to view this HTML file, the following text is displayed:

> If you can see this message, your Web browser is not Java-enabled. There is a Java applet on this Web page that you are not seeing.

8.2 Synchronizing Multiple Threads

The correct behavior of a multithreaded program generally depends on multiple threads cooperating with each other. This often involves threads not doing certain things at the same time or waiting for each other to perform certain tasks. This type of cooperation is called *synchronization*. This section discusses some common strategies for synchronization and how they can be implemented in Java.

The simplest strategy for ensuring that threads are correctly synchronized is to write code that works correctly when executed concurrently by any number of threads. However, this is more easily said than done. Most useful computations involve doing some activity, such as updating an instance variable or updating a display, that must be synchronized in order to happen correctly.

If a method only updates its local variables and calls other methods that only modify their local variables, the method can be invoked by multiple threads without any need for synchronization. `Math.sqrt()` and the `length()` method of the `String` class are examples of such methods.

A method that creates objects and meets the above criterion may not require synchronization. If the constructors invoked by the method do not modify anything but their own local variables and instance variables of the object they are constructing, and they only call methods that do not need to be synchronized, the method itself does not need to be synchronized. An example of such a method is the `substring()` in the `String` class.

Beyond these two simple cases, it is impossible to give an exhaustive list of rules that can tell you whether or not a method needs to be synchronized. You need to consider what the method is doing and think about any ill effects of concurrent execution in order to decide if synchronization is necessary.

Single-Threaded Execution

When more than one thread is trying to update the same data at the same time, the result may be wrong or inconsistent. Consider the following example:

```
class CountIt {
    int i = 0;
    void count() {
        i = i + 1;
    }
}
```

The method `count()` is supposed to increment the variable `i` by one. However, suppose that there are two threads, `A` and `B`, that call `count()` at the same time. In this case, it is possible that `i` could be incremented only once, instead of twice. Say the value of `i` is 7. Thread `A` calls the `count()` method and computes `i+1` as 8. Then thread `B` calls the `count()` method and computes `i+1` as 8 because thread `A` has not yet assigned the new value to `i`. Next, thread `A` assigns the value 8 to the variable `i`. Finally, thread `B` assigns the value 8 to the variable `i`. Thus, even though the `count()` method is called twice, the variable has only been incremented once when the sequence is finished.

Clearly, this code can fail to produce its intended result when it is executed concurrently by more than one thread. A piece of code that can fail to produce its intended result when executed concurrently is called a *critical section*. However, a critical section does behave correctly when it is executed by only one thread at a time. The strategy of single-threaded execution is to allow only one thread to execute a critical section of code at a time. If a thread wants to execute a critical section that another thread is already executing, the thread has to wait until the first thread is done and no other thread is executing that code before it can proceed.

Java provides the `synchronized` statement and the `synchronized` method modifier for implementing single-threaded execution. Before executing the block in a `synchronized` statement, the current thread must obtain a lock for the object referenced by the expression. If a method is declared with the `synchronized` modifer, the current thread must obtain a lock before it can invoke the method. If the method is not declared `static`, the thread must obtain a lock associated with the object used to access the method. If the method is declared `static`, the thread must obtain a lock associated with the class in which the method is declared. Because a thread must obtain a lock before executing a `synchronized` method, Java guarantees that `synchronized` methods are executed by only one thread at a time.

Modifying the `count()` method to make it a `synchronized` method ensures that it works as intended.

```
class CountIt {
    int i = 0;
    synchronized void count() {
        i = i + 1;
    }
}
```

The strategy of single-threaded execution can also be used when multiple methods update the same data. Consider the following example:

```
class CountIt2 {
    int i = 0;
    void count() {
        i = i + 1;
    }
    void count2() {
        i = i + 2;
    }
}
```

By the same logic used above, if the `count()` and `count2()` methods are executed concurrently, the result could be to increment `i` by 1, 2, or 3. Both the `count()` and `count2()` methods can be declared as `synchronized` to ensure that they are not executed concurrently with themselves or each other:

```
class CountIt2 {
    int i = 0;
    synchronized void count() {
        i = i + 1;
    }
    synchronized void count2() {
        i = i + 2;
    }
}
```

Sometimes it's necessary for a thread to make multiple method calls to manipulate an object without another thread calling that object's methods at the same time. Consider the following example:

```
System.out.print(new Date());
System.out.print(" : ");
System.out.println(foo());
```

If the code in the example is executed concurrently by multiple threads, the output from the two threads will be interleaved. The `synchronized` keyword provides a way to ensure that only one thread at a time can execute a block of code. Before executing the block in a `synchronized` statement, the current thread must obtain a lock for the object referenced by the expression. The above code can be modified to give a thread exclusive access to the `OutputStream` object referenced by `System.out`:

```
synchronized (System.out) {
    System.out.print(new Date());
    System.out.print(" : ");
    System.out.println(foo());
}
```

Note that this approach only works if other code that wants to call methods in the same object also uses similar `synchronized` statements, or if the methods in question are all `synchronized` methods. In this case, the `print()` and `println()` methods are `synchronized`, so other pieces of code that need to use these methods do not need to use a `synchronized` statement.

When an inner class is updating fields in its enclosing instance, simply making a method `synchronized` does not provide the needed single-threaded execution. Consider the following code:

```
public class Z extends Frame {
    int pressCount = 0;
    ...
    private class CountButton extends Button
                              implements ActionListener {
        public void actionPerformed(ActionEvent evt) {
            pressCount ++;
        }
    }
    ...
}
```

If a `Z` object instantiates more than one instance of `CountButton`, you need to use single-threaded execution to ensure that updates to `pressCount` are done correctly. Unfortunately, declaring the `actionPerformed()` method of `CountButton` to be `synchronized` does not accomplish that goal because it only forces the method to acquire a lock on the instance of `CountButton` it is associated with before it executes. The object you need to acquire a lock for is the enclosing instance of `Z`.

One way to have a `CountButton` object capture a lock on its enclosing instance of `Z` is to update `pressCount` inside of a `synchronized` statement. For example:

```
synchronized (Z.this) {
    pressCount ++;
}
```

The drawback to this approach is that every piece of code that accesses `pressCount` in any inner class of `Z` must be in a similar `synchronized` statement. Otherwise, it is possible for `pressCount` to be updated incorrectly. The more pieces of code that need to be inside of `synchronized` statements, the more places there are to introduce bugs in your program.

A more robust approach is to have the inner class update a field in its enclosing instance by calling a `synchronized` method in the enclosing instance. For example:

```
public class Z extends Frame {
    int pressCount = 0;
    synchronized incrementPressCount() {
        pressCount++;
    }
    ...
    private class CountButton extends Button
                              implements ActionListener {
        public void actionPerformed(ActionEvent evt) {
            incrementPressCount();
        }
    }
    ...
}
```

References Inner Classes; Method modifiers; The synchronized Statement

Optimistic Single-Threaded Execution

When multiple threads are updating a data structure, single-threaded execution is the obvious strategy to use to ensure correctness of the operations on the data structure. However, single-threaded execution can cause some problems of its own. Consider the following example:

```
public class Queue extends java.util.Vector {
    synchronized public void put(Object obj) {
        addElement(obj);
    }
    synchronized public Object get() throws EmptyQueueException {
        if (size() == 0)
            throw new EmptyQueueException();
        Object obj = elementAt(0);
        removeElementAt(0);
        return obj;
    }
}
```

This example implements a first-in, first-out (FIFO) queue. If the `get()` method of a `Queue` object is called when the queue is empty, the method throws an exception. Now suppose that you want to write the `get()` method so that when the queue is empty, the method waits for an item to be put in the queue, rather than throwing an exception. In order for an item to be put in the queue, the `put()` method of the queue must be invoked. But using the single-threaded execution strategy, the `put()` method will never be able to run while the `get()` method is waiting for the queue to receive an item. A good way to solve this dilemma is to use a strategy called *optimistic single-threaded execution*.

The optimistic single-threaded execution strategy is similar to the single-threaded execution strategy. They both begin by getting a lock on an object to ensure that the currently executing thread is the only thread that can execute a piece of code, and they both end by releasing that lock. The difference is what happens in between. Using the optimistic single-threaded execution strategy, if a piece of code discovers that conditions are not right to proceed, the code releases the lock it has on the object that enforces single-threaded execution and waits. When another piece of code changes things in such a way that might allow the first piece of code to proceed, it notifies the first piece of code that it should try to regain the lock and proceed.

To implement this strategy, the `Object` class provides methods called `wait()`, `notify()`, and `notifyAll()`. These methods are inherited by every other class in Java. The following example shows how to implement a queue that uses the optimistic single-threaded execution strategy, so that when the queue is empty, its `get()` method waits for the queue to have an item put in it:

```
public class Queue extends java.util.Vector {
    synchronized public void put(Object obj) {
        addElement(obj);
        notify();
    }
    synchronized public Object get() throws EmptyQueueException {
        while (size() == 0)
            wait();
        Object obj = elementAt(0);
        removeElementAt(0);
        return obj;
    }
}
```

In the above implementation of the `Queue` class, the `get()` method calls `wait()` when the queue is empty. The `wait()` method releases the lock that excludes other threads from executing methods in the `Queue` object, and then waits until another thread calls the `put()` method. When `put()` is called, it adds an item to the queue and calls `notify()`. The `notify()` method tells a thread that is waiting to return from a `wait()` method that it should attempt to regain its lock and proceed. If there is more than one thread waiting to regain the lock on the object, `notify()` chooses one of the threads arbitrarily. The `notifyAll()` method is similar to `notify()`, but instead of choosing one thread to notify, it notifies all of the threads that are waiting to regain the lock on the object.

Notice that the `get()` method calls `wait()` inside a `while` loop. Between the time that `wait()` is notified that it should try to regain its lock and the time it actually does regain the lock, another thread may have called the `get()` method and emptied the queue. The `while` loop guards against this situation.

References Method modifiers; Object; The synchronized Statement

Rendezvous

Sometimes it is necessary to have a thread wait to continue until another thread has completed its work and died. This type of synchronization uses the rendezvous strategy. The `Thread` class provides the `join()` method for implementing this strategy. When the `join()` method is called on a `Thread` object, the method returns immediately if the thread is dead. Otherwise, the method waits until the thread dies and then returns.

References Thread

Balking

Some methods should not be executed concurrently, and have a time-sensitive nature that makes postponing calls to them a bad idea. This is a common situation when software is controlling real-world devices. Suppose you have a Java program that is embedded in an electronic control for a toilet. There is a method called `flush()` that is responsible for flushing a toilet, and `flush()` can be called from more than one thread. If a thread calls `flush()` while another thread is already executing `flush()`, the second call should do nothing. A toilet is capable of only one flush at a time, and having a concurrent call to the `flush()` method result in a second flush would only waste water.

This scenario suggests the use of the balking strategy. The balking strategy allows no more than one thread to execute a method at a time. If another thread attempts to execute the method, the method simply returns without doing anything. Here is an example that shows what such a `flush()` method might look like:

```
boolean busy;
void flush() {
    synchronized (this) {
        if (busy)
            return;
        busy = true;
    }
    // code to make flush happen goes here
    busy = false;
}
```

Explicit Synchronization

When the synchronization needs of a thread are not known in advance, you can use a strategy called explicit synchronization. The explicit synchronization strategy allows you to explicitly tell a thread when it can and cannot run. For example, you may want an animation to start and stop in response to external events that happen at unpredictable times, so you need to be able to tell the animation when it can run.

To implement this strategy, the `Thread` class provides methods called `suspend()` and `resume()`. You can suspend the execution of a thread by calling the `suspend()` method of the `Thread` object that controls the thread. You can later resume execution of the thread by calling the `resume()` method on the `Thread` object.

References Thread

PREVIOUS | HOME | NEXT
Using Thread Objects | BOOK INDEX | Exception Handling

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Thread

Name

Thread

Synopsis

Class Name:

`java.lang.Thread`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

`java.lang.Runnable`

Availability:

JDK 1.0 or later

Description

The Thread class encapsulates all of the information about a single thread of control running in a Java environment. Thread objects are used to control threads in a multithreaded program.

The execution of Java code is always under the control of a Thread object. The Thread class provides a static method called currentThread() that can be used to get a reference to the Thread object that controls the current thread of execution.

In order for a Thread object to be useful, it must be associated with a method that it is supposed to run. Java provides two ways of associating a Thread object with a method:

- Declare a subclass of Thread that defines a run() method. When such a class is instantiated and the object's start() method is called, the thread invokes this run() method.

- Pass a reference to an object that implements the Runnable interface to a Thread constructor. When the start() method of such a Thread object is called, the thread invokes the run() method of the Runnable object.

After a thread is started, it dies when one of the following things happens:

- The run() method called by the Thread returns.

- An exception is thrown that causes the run() method to be exited.

- The stop() method of the Thread is called.

Class Summary

```
public class java.lang.Thread extends java.lang.Object
                              implements java.lang.Runnable {
    // Constants
    public final static int MAX_PRIORITY;
    public final static int MIN_PRIORITY;
    public final static int NORM_PRIORITY;
    // Constructors
    public Thread();
    public Thread(Runnable target);
    public Thread(Runnable target, String name);
    public Thread(String name);
    public Thread(ThreadGroup group, Runnable target);
    public Thread(ThreadGroup group, Runnable target, String name);
    public Thread(ThreadGroup group, String name);
    // Class Methods
    public static int activeCount();
    public static native Thread currentThread();
    public static void dumpStack();
    public static int enumerate(Thread tarray[]);
    public static boolean interrupted();
    public static native void sleep(long millis);
    public static void sleep(long millis, int nanos);
    public static native void yield();
    // Instance Methods
    public void checkAccess();
    public native int countStackFrames();
    public void destroy();
    public final String getName();
    public final int getPriority();
    public final ThreadGroup getThreadGroup();
    public void interrupt();
    public final native boolean isAlive();
    public final boolean isDaemon();
    public boolean isInterrupted();
    public final void join();
    public final synchronized void join(long millis);
    public final synchronized void join(long millis, int nanos);
    public final void resume();
    public void run();
    public final void setDaemon(boolean on);
    public final void setName(String name);
    public final void setPriority(int newPriority);
    public synchronized native void start();
    public final void stop();
    public final synchronized void stop(Throwable o);
    public final void suspend();
    public String toString();
}
```

Constants

MAX_PRIORITY

`public final static int MAX_PRIORITY = 10`

Description

The highest priority a thread can have.

MIN_PRIORITY

`public final static int MIN_PRIORITY = 1`

Description

The lowest priority a thread can have.

NORM_PRIORITY

`public final static int NORM_PRIORITY = 5`

Description

The default priority assigned to a thread.

Constructors

Thread

`public Thread()`

Throws

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

Creates a `Thread` object that belongs to the same `ThreadGroup` object as the current thread, has the same daemon attribute as the current thread, has the same priority as the current thread, and has a default name.

A `Thread` object created with this constructor invokes its own `run()` method when the `Thread` object's `start()` method is called. This is not useful unless the object belongs to a subclass of the `Thread` class that overrides the `run()` method.

Calling this constructor is equivalent to:

```
Thread(null, null, genName)
```

`genName` is an automatically generated name of the form `"Thread-"+n`, where `n` is an integer incremented each time a `Thread` object is created.

public Thread(String name)

Parameters

`name`

The name of this `Thread` object.

Throws

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

Creates a `Thread` object that belongs to the same `ThreadGroup` object as the current thread, has the same daemon attribute as the current thread, has the same priority as the current thread, and has the specified name.

A `Thread` object created with this constructor invokes its own `run()` method when the `Thread` object's `start()` method is called. This is not useful unless the object belongs to a subclass of the `Thread` class that overrides the `run()` method.

Calling this constructor is equivalent to:

```
Thread(null, null, name)
```

The uniqueness of the specified `Thread` object's name is not checked, which may be a problem for programs that attempt to identify `Thread` objects by their name.

`public Thread(ThreadGroup group, Runnable target)`

Parameters

`group`

The `ThreadGroup` object that this `Thread` object is to be added to.

`target`

A reference to an object that implements the `Runnable` interface.

Throws

SecurityException

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

Creates a `Thread` object that belongs to the specified `ThreadGroup` object, has the same daemon attribute as the current thread, has the same priority as the current thread, and has a default name.

A `Thread` object created with this constructor invokes the `run()` method of the specified `Runnable` object when the `Thread` object's `start()` method is called.

Calling this constructor is equivalent to:

```
Thread(group, target, genName)
```

`genName` is an automatically generated name of the form `"Thread-"+n`, where `n` is an integer that is incremented each time a `Thread` object is created.

`public Thread(ThreadGroup group, Runnable target, String name)`

Parameters

`group`

The `ThreadGroup` object that this `Thread` object is to be added to.

`target`

A reference to an object that implements the `Runnable` interface.

`name`

The name of this `Thread` object.

Throws

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

Creates a `Thread` object that belongs to the specified `ThreadGroup` object, has the same daemon attribute as the current thread, has the same priority as the current thread, and has the specified name.

A `Thread` object created with this constructor invokes the `run()` method of the specified `Runnable` object when the `Thread` object's `start()` method is called.

The uniqueness of the specified `Thread` object's name is not checked, which may be a problem for programs that attempt to identify `Thread` objects by their names.

`public Thread(ThreadGroup group, String name)`

Parameters

`group`

The `ThreadGroup` object that this `Thread` object is to be added to.

`name`

The name of this `Thread` object.

Throws

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

Creates a `Thread` object that belongs to the specified `ThreadGroup` object, has the same daemon attribute as the current thread, has the same priority as the current thread, and has the specified name.

A `Thread` object created with this constructor invokes its own `run()` method when the `Thread` object's `start()` method is called. This is not useful unless the object belongs to a subclass of the `Thread` class that overrides the `run()` method. Calling this constructor is equivalent to:

```
Thread(group, null, name)
```

The uniqueness of the specified `Thread` object's name is not checked, which may be a problem for programs that attempt to identify `Thread` objects by their name.

Class Methods

activeCount

`public static int activeCount()`

Returns

The current number of threads in the `ThreadGroup` of the currently running thread.

Description

This method returns the number of threads in the `ThreadGroup` of the currently running thread for which the `isAlive()` method returns `true`.

currentThread

`public static native Thread currentThread()`

Returns

A reference to the `Thread` object that controls the currently executing thread.

Description

This method returns a reference to the `Thread` object that controls the currently executing thread.

dumpStack

`public static void dumpStack()`

Description

This method outputs a stack trace of the currently running thread.

enumerate

`public static int enumerate(Thread tarray[])`

Parameters

`tarray`

A reference to an array of `Thread` objects.

Returns

The number of `Thread` objects stored in the array.

Description

This method stores a reference in the array for each of the `Thread` objects in the `ThreadGroup` of the currently running thread for which the `isAlive()` method returns `true`.

Calling this method is equivalent to:

```
currentThread().getThreadGroup().enumerate(tarray)
```

If the array is not big enough to contain references to all the `Thread` objects, only as many references as will fit are put into the array. No indication is given that some `Thread` objects were left out, so it is a good idea to call `activeCount()` before calling this method, to get an idea of how large to make the array.

interrupted

`public static boolean interrupted()`

Returns

`true` if the currently running thread has been interrupted; otherwise `false`.

Description

This method determines whether or not the currently running thread has been interrupted.

sleep

`public static native void sleep(long millis)`

Parameters

`millis`

The number of milliseconds that the currently running thread should sleep.

Throws

`InterruptedException`

If another thread interrupts the currently running thread.

Description

This method causes the currently running thread to sleep. The method does not return until at least the specified number of milliseconds have elapsed.

While a thread is sleeping, it retains ownership of all locks. The `Object` class defines a method called `wait()` that is similar to `sleep()` but causes the currently running thread to temporarily relinquish its locks.

`public static void sleep(long millis, int nanos)`

Parameters

`millis`

The number of milliseconds that the currently running thread should sleep.

`nanos`

An additional number of nanoseconds to sleep.

Throws

`InterruptedException`

If another thread interrupts the currently running thread.

Description

This method causes the currently running thread to sleep. The method does not return until at least the specified number of milliseconds have elapsed.

While a thread is sleeping, it retains ownership of all locks. The `Object` class defines a method called `wait()` that is similar to `sleep()` but causes the currently running thread to temporarily relinquish its locks.

Note that Sun's reference implementation of Java does not attempt to implement the precision implied by this method. Instead, it rounds to the nearest millisecond (unless `millis` is `0`, in which case it rounds up to 1 millisecond) and calls `sleep(long)`.

yield

`public static native void yield()`

Description

> This method causes the currently running thread to yield control of the processor so that another thread can be scheduled.

Instance Methods

checkAccess

`public void checkAccess()`

Throws

> `SecurityException`
>
> > If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

> This method determines if the currently running thread has permission to modify this `Thread` object.

countStackFrames

`public native int countStackFrames()`

Returns

> The number of pending method invocations on this thread's stack.

Description

> This method returns the number of pending method invocations on this thread's stack.

destroy

`public void destroy()`

Description

> This method is meant to terminate this thread without any of the usual cleanup (i.e., any locks held by the thread are not released). This method provides a last-resort way to terminate a thread. While a thread can defeat its `stop()` method by catching objects thrown from it, there is nothing that a thread can do to stop itself from being destroyed.
>
> Note that Sun's reference implementation of Java does not implement the documented functionality of this method. Instead, the implementation of this method just throws a `NoSuchMethodError`.

getName

`public final String getName()`

Returns

> The name of this thread.

Description

> This method returns the name of this `Thread` object.

getPriority

`public final int getPriority()`

Returns

> The priority of this thread.

Description

> This method returns the priority of this `Thread` object.

getThreadGroup

`public final ThreadGroup getThreadGroup()`

Returns

The `ThreadGroup` of this thread.

Description

This method returns a reference to the `ThreadGroup` that this `Thread` object belongs to.

interrupt

`public void interrupt()`

Description

This method interrupts this `Thread` object.

Note that prior to version 1.1, Sun's reference implementation of Java does not implement the documented functionality of this method. Instead, the method just sets a `private` flag that indicates that an interrupt has been requested. None of the methods that should throw an `InterruptedException` currently do. However, the `interrupted()` and `isInterrupted()` methods do return `true` after this method has been called.

isAlive

`public final native boolean isAlive()`

Returns

`true` if this thread is alive; otherwise `false`.

Description

This method determines whether or not this `Thread` object is alive. A `Thread` object is alive if it has been started and has not yet died. In other words, it has been scheduled to run before and can still be scheduled to run again. A thread is generally alive after its `start()` method is called and until its `stop()` method is called.

isDaemon

`public final boolean isDaemon()`

Returns

> `true` if the thread is a daemon thread; otherwise `false`.

Description

> This method determines whether or not this thread is a daemon thread, based on the value of the daemon attribute of this `Thread` object.

isInterrupted

`public boolean isInterrupted()`

Returns

> `true` if this thread has been interrupted; otherwise `false`.

Description

> This method determines whether or not this `Thread` object has been interrupted.

join

`public final void join()`

Throws

> `InterruptedException`
>
> > If another thread interrupts this thread.

Description

> This method allows the thread that calls it to wait for the `Thread` associated with this method to die. The method returns when the `Thread` dies. If this thread is already dead, then this method returns immediately.

`public final synchronized void join(long millis)`

Parameters

`millis`

The maximum number of milliseconds to wait for this thread to die.

Throws

`InterruptedException`

If another thread interrupts this thread.

Description

This method causes a thread to wait to die. The method returns when this `Thread` object dies or after the specified number of milliseconds has elapsed, whichever comes first. However, if the specified number of milliseconds is zero, the method will wait forever for this thread to die. If this thread is already dead, the method returns immediately.

`public final synchronized void join(long millis, int nanos)`

Parameters

`millis`

The maximum number of milliseconds to wait for this thread to die.

`nanos`

An additional number of nanoseconds to wait.

Throws

`InterruptedException`

If another thread interrupts this thread.

Description

This method causes a thread to wait to die. The method returns when this `Thread` object dies or after the specified amount of time has elapsed, whichever comes first. However, if `millis` and `nanos` are zero, the method will wait forever for this thread to die. If this thread is already dead, the method returns immediately.

Note that Sun's reference implementation of Java does not attempt to implement the precision implied by this method. Instead, it rounds to the nearest millisecond (unless `millis` is `0`, in which case it rounds up to 1 millisecond) and calls `join(long)`.

resume

`public final void resume()`

Throws

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method resumes a suspended thread. The method causes this `Thread` object to once again be eligible to be run. Calling this method for a thread that is not suspended has no effect.

run

`public void run()`

Implements

`Runnable.run()`

Description

A `Thread` object's `start()` method causes the thread to invoke a `run()` method. If this `Thread` object was created without a specified `Runnable` object, the `Thread` object's own `run()` method is executed. This behavior is only useful in a subclass of `Thread` that overrides this `run()` method, since the `run()` method of the `Thread` class does not do anything.

setDaemon

`public final void setDaemon(boolean on)`

Parameters

`on`

The new value for this thread's daemon attribute.

Throws

`IllegalThreadStateException`

If this method is called after this thread has been started and while it is still alive.

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method sets the daemon attribute of this `Thread` object to the given value. This method must be called before the thread is started. If a thread dies and there are no other threads except daemon threads alive, the Java virtual machine stops.

setName

`public final void setName(String name)`

Parameters

`name`

The new name for this thread.

Throws

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a

`SecurityException`.

Description

This method sets the name of this `Thread` object to the given value. The uniqueness of the specified `Thread` object's name is not checked, which may be a problem for programs that attempt to identify `Thread` objects by their name.

setPriority

`public final void setPriority(int newPriority)`

Parameters

`newPriority`

The new priority for this thread.

Throws

`IllegalArgumentException`

If the given priority is less than `MIN_PRIORITY` or greater than `MAX_PRIORITY`.

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method sets the priority of this `Thread` to the given value.

start

`public synchronized native void start()`

Throws

`IllegalThreadStateException`

If this `Thread` object's `start()` method has been called before.

Description

This method starts this `Thread` object, allowing it to be scheduled for execution. The top-level code that is executed by the thread is the `run()` method of the `Runnable` object specified in the constructor that was used to create this object. If no such object was specified, the top-level code executed by the thread is this object's `run()` method.

It is not permitted to start a thread more than once.

stop

`public final void stop()`

Throws

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method causes this `Thread` object to stop executing by throwing a `ThreadDeath` object. The object is thrown in this thread, even if the method is called from a different thread. This thread is forced to stop whatever it is doing and throw a newly created `ThreadDeath` object. If this thread was suspended, it is resumed; if it was sleeping, it is awakened. Normally, you should not catch `ThreadDeath` objects in a `try` statement. If you need to catch `ThreadDeath` objects to detect a `Thread` is about to die, the `try` statement that catches `ThreadDeath` objects should rethrow them.

When an object is thrown out of the `run()` method associated with a `Thread`, the `uncaughtException()` method of the `ThreadGroup` for that `Thread` is called. The `uncaughtException()` method normally outputs a stack trace. However, `uncaughtException()` treats a `ThreadDeath` object as a special case by not outputting a stack trace. When the `uncaughtException()` method returns, the thread is dead. The thread is never scheduled to run again.

If this `Thread` object's `stop()` method is called before this thread is started, the `ThreadDeath` object is thrown as soon as the thread is started.

`public final synchronized void stop(Throwable o)`

Parameters

`o`

The object to be thrown.

Throws

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method causes this `Thread` object to stop executing by throwing the given object. Normally, the `stop()` method that takes no arguments and throws a `ThreadDeath` object should be called instead of this method. However, if it is necessary to stop a thread by throwing some other type of object, this method can be used.

The object is thrown in this thread, even if the method is called from a different thread. This thread is forced to stop whatever it is doing and throw the `Throwable` object `o`. If this thread was suspended, it is resumed; if it was sleeping, it is awakened.

When an object is thrown out of the `run()` method associated with a `Thread`, the `uncaughtException()` method of the `ThreadGroup` for that `Thread` is called. If the thrown object is not an instance of the `ThreadDeath` class, `uncaughtException()` calls the thrown object's `printStackTrace()` method and then the thread dies. The thread is never scheduled to run again.

If this `Thread` object's `stop()` method is called before this thread is started, the `ThreadDeath` object is thrown as soon as the thread is started.

suspend

`public final void suspend()`

Throws

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method suspends a thread. The method causes this `Thread` object to temporarily be ineligible to be run. The thread becomes eligible to be run again after its `resume()` method is called. Calling this method for a thread that is already suspended has no effect.

toString

`public String toString()`

Returns

A string representation of this `Thread` object.

Overrides

`Object.toString()`

Description

This method returns a string representation of this `Thread` object.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

Exceptions; Object; Runnable; SecurityManager; ThreadGroup; Threads 8

PREVIOUS | HOME | NEXT
System | BOOK INDEX | ThreadGroup

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

ThreadGroup

Name

ThreadGroup

Synopsis

Class Name:

```
java.lang.ThreadGroup
```

Superclass:

```
java.lang.Object
```

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The ThreadGroup class implements a grouping scheme for threads. A ThreadGroup object can own Thread objects and other ThreadGroup objects. The ThreadGroup class provides methods that allow a ThreadGroup object to control its Thread and ThreadGroup objects as a group. For example, suspend() and resume() methods of a ThreadGroup object call the suspend() and resume() methods of each of the Thread and ThreadGroup objects that belong to the particular ThreadGroup.

When a Java program starts, a ThreadGroup object is created to own the first Thread. Any additional ThreadGroup objects are explicitly created by the program.

Class Summary

```
public class java.lang.ThreadGroup extends java.lang.Object {
    // Constructors
    public ThreadGroup(String name);
    public ThreadGroup(ThreadGroup parent, String name;
    // Instance Methods
    public int activeCount();
    public int activeGroupCount();
    public boolean allowThreadSuspension(boolean b);       // New in 1.1
    public final void checkAccess();
    public final void destroy();
    public int enumerate(Thread list[]);
    public int enumerate(Thread list[], boolean recurse);
    public int enumerate(ThreadGroup list[]);
    public int enumerate(ThreadGroup list[], boolean recurse);
    public final int getMaxPriority();
    public final String getName();
    public final ThreadGroup getParent();
    public final boolean isDaemon();
    public synchronized boolean isDestroyed();             // New in 1.1
    public void list();
    public final boolean parentOf(ThreadGroup g);
    public final void resume();
    public final void setDaemon(boolean daemon);
    public final void setMaxPriority(int pri);
    public final void stop();
    public final void suspend();
    public String toString();
    public void uncaughtException(Thread t, Throwable e);
}
```

Constructors

ThreadGroup

`public ThreadGroup(String name)`

Parameters

`name`

The name of this `ThreadGroup` object.

Throws

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

Creates a `ThreadGroup` object that has the specified name and the same parent `ThreadGroup` as the current thread.

`public ThreadGroup(ThreadGroup parent, String name)`

Parameters

`parent`

The `ThreadGroup` object that this `ThreadGroup` object is to be added to.

`name`

The name of this `ThreadGroup` object.

Throws

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a

`SecurityException`.

Description

Creates a `ThreadGroup` object with the specified name and parent `ThreadGroup` object.

Instance Methods

activeCount

`public int activeCount()`

Returns

An approximation of the current number of threads in this `ThreadGroup` object and any child `ThreadGroup` objects.

Description

This method returns an approximation of the number of threads that belong to this `ThreadGroup` object and any child `ThreadGroup` objects. The count is approximate because a thread can die after it is counted, but before the complete count is returned. Also, after a child `ThreadGroup` is counted but before the total count is returned, additional `Thread` and `ThreadGroup` objects can be added to a child `ThreadGroup`.

activeGroupCount

`public int activeGroupCount()`

Returns

An approximation of the current number of child `ThreadGroup` objects in this `ThreadGroup` object.

Description

This method returns an approximation of the number of child `ThreadGroup` objects that belong to this `ThreadGroup` object. The count is approximate because after a child `ThreadGroup` is counted but before the total count is returned, additional `ThreadGroup` objects can be added to a child `ThreadGroup`.

allowThreadSuspension

`public boolean allowThreadSuspension(boolean b)`

Availability

New as of JDK 1.1

Parameters

`b`

A `boolean` value that specifies whether or not the run-time system is allowed to suspend threads due to low memory.

Returns

The `boolean` value `true`.

Description

This method specifies whether or not the Java virtual machine is allowed to suspend threads due to low memory.

checkAccess

`public final void checkAccess()`

Throws

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method determines if the currently running thread has permission to modify this `ThreadGroup` object.

destroy

`public final void destroy()`

Throws

`IllegalThreadStateException`

If this `ThreadGroup` object is not empty, or if it has already been destroyed.

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method destroys this `ThreadGroup` object and any child `ThreadGroup` objects. The `ThreadGroup` must not contain any `Thread` objects. This method also removes the `ThreadGroup` object from its parent `ThreadGroup` object.

enumerate

`public int enumerate(Thread list[])`

Parameters

`list`

A reference to an array of `Thread` objects.

Returns

The number of `Thread` objects stored in the array.

Description

This method stores a reference in the array for each of the `Thread` objects that belongs to this `ThreadGroup` or any of its child `ThreadGroup` objects.

If the array is not big enough to contain references to all the `Thread` objects, only as many references as will fit are put into the array. No indication is given that some `Thread` objects were left out, so it is a good idea to call `activeCount()` before calling this method, to get an idea of how large to make the array.

`public int enumerate(Thread list[], boolean recurse)`

Parameters

`list`

A reference to an array of `Thread` objects.

`recurse`

A `boolean` value that specifies whether or not to include `Thread` objects that belong to child `ThreadGroup` objects of this `ThreadGroup` object.

Returns

The number of `Thread` objects stored in the array.

Description

This method stores a reference in the array for each of the `Thread` objects that belongs to this `ThreadGroup` object. If `recurse` is `true`, the method also stores a reference for each of the `Thread` objects that belongs to a child `ThreadGroup` object of this `ThreadGroup`.

If the array is not big enough to contain references to all the `Thread` objects, only as many references as will fit are put into the array. No indication is given that some `Thread` objects were left out, so it is a good idea to call `activeCount()` before calling this method, to get an idea of how large to make the array.

`public int enumerate(ThreadGroup list[])`

Parameters

`list`

A reference to an array of `ThreadGroup` objects.

Returns

The number of `ThreadGroup` objects stored in the array.

Description

This method stores a reference in the array for each `ThreadGroup` object that belongs to this `ThreadGroup` or any of its child `ThreadGroup` objects.

If the array is not big enough to contain references to all the `ThreadGroup` objects, only as many references as will fit are put into the array. No indication is given that some `ThreadGroup` objects were left out, so it is a good idea to call `activeGroupCount()` before calling this method, to get an idea of how large to make the array.

`public int enumerate(Thread list[], boolean recurse)`

Parameters

`list`

A reference to an array of `ThreadGroup` objects.

`recurse`

A `boolean` value that specifies whether or not to include `ThreadGroup` objects that belong to child `ThreadGroup` objects of this `ThreadGroup` object.

Returns

The number of `ThreadGroup` objects stored in the array.

Description

This method stores a reference in the array for each of the `ThreadGroup` objects that belongs to this `ThreadGroup` object. If `recurse` is `true`, the method also stores a reference for each of the `ThreadGroup` objects that belongs to a child `ThreadGroup` object of this `ThreadGroup`.

If the array is not big enough to contain references to all the `ThreadGroup` objects, only as many references as will fit are put into the array. No indication is given that some `ThreadGroup` objects were left out, so it is a good idea to call `activeGroupCount()` before calling this method, to get an idea of how large to make the array.

getMaxPriority

`public final int getMaxPriority()`

Returns

> The maximum priority that can be assigned to `Thread` objects that belong to this `ThreadGroup` object.

Description

> This method returns the maximum priority that can be assigned to `Thread` objects that belong to this `ThreadGroup` object.
>
> It is possible for a `ThreadGroup` to contain `Thread` objects that have higher priorities than this maximum, if they were given that higher priority before the maximum was set to a lower value.

getName

```
public final String getName()
```

Returns

> The name of this `ThreadGroup` object.

Description

> This method returns the name of this `ThreadGroup` object.

getParent

```
public final ThreadGroup getParent()
```

Returns

> The parent `ThreadGroup` object of this `ThreadGroup`, or `null` if this `ThreadGroup` is the root of the thread group hierarchy.

Description

> This method returns the parent `ThreadGroup` object of this `ThreadGroup` object. If this `ThreadGroup` is at the root of the thread group hierarchy and has no parent, the method returns `null`.

isDaemon

`public final boolean isDaemon()`

Returns

> `true` if this `ThreadGroup` is a daemon thread group; otherwise `false`.

Description

> This method determines whether or not this `ThreadGroup` is a daemon thread group, based on the value of daemon attribute of this `ThreadGroup` object. A daemon thread group is destroyed when the last `Thread` in it is stopped, or the last `ThreadGroup` in it is destroyed.

isDestroyed

`public synchronized boolean isDestroyed()`

Availability

> New as of JDK 1.1

Returns

> `true` if this `ThreadGroup` has been destroyed; otherwise `false`.

Description

> This method determines whether or not this `ThreadGroup` has been destroyed.

list

`public void list()`

Description

> This method outputs a listing of the contents of this `ThreadGroup` object to `System.out`.

parentOf

`public final boolean parentOf(ThreadGroup g)`

Parameters

`g`

A `ThreadGroup` object.

Returns

`true` if this `ThreadGroup` object is the same `ThreadGroup`, or a direct or indirect parent of the specified `ThreadGroup`; otherwise `false`.

Description

This method determines if this `ThreadGroup` object is the same as the specified `ThreadGroup` or one of its ancestors in the thread-group hierarchy.

resume

`public final void resume()`

Throws

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method resumes each `Thread` object that directly or indirectly belongs to this `ThreadGroup` object by calling its `resume()` method.

setDaemon

`public final void setDaemon(boolean daemon)`

Parameters

`daemon`

The new value for this `ThreadGroup` object's daemon attribute.

Throws

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method sets the daemon attribute of this `ThreadGroup` object to the given value. A daemon thread group is destroyed when the last `Thread` in it is stopped, or the last `ThreadGroup` in it is destroyed.

setMaxPriority

`public final void setMaxPriority(int pri)`

Parameters

`pri`

The new maximum priority for `Thread` objects that belong to this `ThreadGroup` object.

Description

This method sets the maximum priority that can be assigned to `Thread` objects that belong to this `ThreadGroup` object.

It is possible for a `ThreadGroup` to contain `Thread` objects that have higher priorities than this maximum, if they were given that higher priority before the maximum was set to a lower value.

stop

`public final void stop()`

Throws

`SecurityException`

If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

> This method stops each `Thread` object that directly or indirectly belongs to this `ThreadGroup` object by calling its `stop()` method.

suspend

`public final void suspend()`

Throws

> `SecurityException`
>
> > If the `checkAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

> This method suspends each `Thread` object that directly or indirectly belongs to this `ThreadGroup` object by calling its `suspend()` method.

toString

`public String toString()`

Returns

> A string representation of this `ThreadGroup` object.

Overrides

> `Object.toString()`

Description

> This method returns a string representation of this `ThreadGroup` object.

uncaughtException

`public void uncaughtException(Thread t, Throwable e)`

Parameters

`t`

A reference to a `Thread` that just died because of an uncaught exception.

`e`

The uncaught exception.

Description

This method is called when a `Thread` object that belongs to this `ThreadGroup` object dies because of an uncaught exception. If this `ThreadGroup` object has a parent `ThreadGroup` object, this method just calls the parent's `uncaughtException()` method. Otherwise, this method must determine whether the uncaught exception is an instance of `ThreadDeath`. If it is, nothing is done. If it is not, the method calls the `printStackTrace()` method of the exception object.

If this method is overridden, the overriding method should end with a call to `super.uncaughtException()`.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

Exceptions; Object; Runnable; SecurityManager; Thread; Threads 8; Throwable

PREVIOUS HOME NEXT

Thread BOOK INDEX Throwable

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Object

Name

Object

Synopsis

Class Name:

`java.lang.Object`

Superclass:

None

Immediate Subclasses:

Too many to be listed here

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The `Object` class is the ultimate superclass of all other classes in Java. Because every other class is a subclass of `Object`, all of the methods accessible from `Object` are inherited by every other class. In other words, all objects in Java, including arrays, have access to implementations of the methods in `Object`.

The methods of `Object` provide some basic object functionality. The `equals()` method compares two objects for equality, while the `hashCode()` method returns a hashcode for an object. The `getClass()` method returns the `Class` object associated with an object. The `wait()`, `notify()`, and `notifyAll()` methods support thread synchronization for an object. The `toString()` method provides a string representation of an object.

Some of these methods should be overridden by subclasses of `Object`. For example, every class should provide its own implementation of the `toString()` method, so that it can provide an appropriate string representation.

Although it is possible to create an instance of the `Object` class, this is rarely done because it is more useful to create specialized objects. However, it is often useful to declare a variable that contains a reference to an `Object` because such a variable can contain a reference to an object of any other class.

Class Summary

```
public class java.lang.Object {
    // Constructors
    public Object();
    // Public Instance Methods
    public boolean equals(Object obj);
    public final native Class getClass();
    public native int hashCode();
    public final native void notify();
    public final native void notifyAll();
    public String toString();
    public final native void wait();
    public final native void wait(long millis);
    public final native void wait(long millis, int nanos);
    // Protected Instance Methods
    protected native Object clone();
    protected void finalize() throws Throwable;
}
```

Constructors

Object

`public Object()`

Description

> Creates an instance of the `Object` class.

Public Instance Methods

equals

`public boolean equals(Object obj)`

Parameters

`obj`

The object to be compared with this object.

Returns

`true` if the objects are equal; `false` if they are not.

Description

The `equals()` method of `Object` returns `true` if the `obj` parameter refers to the same object as the object this method is associated with. This is equivalent to using the `==` operator to compare two objects.

Some classes, such as `String`, override the `equals()` method to provide a comparison based on the contents of the two objects, rather than on the strict equality of the references. Any subclass can override the `equals()` method to implement an appropriate comparison, as long as the overriding method satisfies the following rules for an equivalence relation:

- The method is *reflexive* : given a reference `x`, `x.equals(x)` returns `true`.
- The method is *symmetric* : given references `x` and `y`, `x.equals(y)` returns `true` if and only if `y.equals(x)` returns `true`.
- The method is *transitive* : given references `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` returns `true`.
- The method is *consistent* : given references `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided that no information contained by the objects referenced by `x` or `y` changes.
- A comparison with `null` returns `false`: given a reference `x` that is non-`null`, `x.equals(null)` returns `false`.

getClass

`public final native Class getClass()`

Returns

A reference to the `Class` object that describes the class of this object.

Description

The `getClass()` method of `Object` returns the `Class` object that describes the class of this object. This method is `final`, so it cannot be overridden by subclasses.

hashCode

`public native int hashCode()`

Returns

A relatively unique value that should be the same for all objects that are considered equal.

Description

The `hashCode()` method of `Object` calculates a hashcode value for this object. The method returns an integer value that should be relatively unique to the object. If the `equals()` method for the object bases its result on the contents of the object, the `hashcode()` method should also base its result on the contents. The `hashCode()` method is provided for the benefit of hashtables, which store and retrieve elements using key values called *hashcodes*. The internal placement of a particular piece of data is determined by its hashcode; hashtables are designed to use hashcodes to provide efficient retrieval.

The `java.util.Hashtable` class provides an implementation of a hashtable that stores values of type `Object`. Each object is stored in the hashtable based on the hash code of its key object. It is important that each object have the most unique hash code possible. If two objects have the same hash code but they are not equal (as determined by `equals()`), a `Hashtable` that stores these two objects may need to spend additional time searching when it is trying to retrieve objects. The implementation of `hashCode()` in `Object` tries to make sure that every object has a distinct hash code by basing its result on the internal representation of a reference to the object.

Some classes, such as `String`, override the `hashCode()` method to produce values based on the contents of individual objects, instead of the objects themselves. In other words, two `String` objects that contain the exact same strings have the same hash code. If `String` did not override the `hashCode()` method inherited from `Object`, these two `String` objects would have different hash code values and it would be impossible to use `String` objects as keys for hashtables.

Any subclass can override the `hashCode()` method to implement an appropriate way of producing hash code values, as long as the overriding method satisfies the following rules:

- If the `hashCode()` method is called on the same object more than once during the execution of a Java application, it must consistently return the same integer value. The integer does not, however, need to be consistent between Java applications, or from one execution of an application to another execution of the same application.

- If two objects compare as equal according to their `equals()` methods, calls to the `hashCode()` methods for the objects must produce the same result.

- If two objects compare as not equal according to their `equals()` methods, calls to the `hashCode()` methods for the two objects are not required to produce distinct results. However, implementations of `hashCode()` that produce distinct results for unequal objects may improve the performance of hashtables.

In general, if a subclass overrides the `equals()` method of `Object`, it should also override the `hashCode()` method.

notify

`public final native void notify()`

Throws

`IllegalMonitorStateException`

If the method is called from a thread that does not hold this object's lock.

Description

The `notify()` method wakes up a thread that is waiting to return from a call to this object's `wait()` method. The awakened thread can resume executing as soon as it regains this object's lock. If more than one thread is waiting, the `notify()` method arbitrarily awakens just one of the threads.

The `notify()` method can be called only by a thread that is the current owner of this object's lock. A thread holds the lock on this object while it is executing a `synchronized` instance method of the object or executing the body of a `synchronized` statement that synchronizes on the object. A thread can also hold the lock for a `Class` object if it is executing a `synchronized static` method of that class.

This method is `final`, so it cannot be overridden by subclasses.

notifyAll

`public final native void notifyAll()`

Throws

`IllegalMonitorStateException`

If the method is called from a thread that does not hold this object's lock.

Description

The `notifyAll()` method wakes up all the threads that are waiting to return from a call to this object's `wait()` method. Each awakened thread can resume executing as soon as it regains this object's lock.

The `notifyAll()` method can be called only by a thread that is the current owner of this object's lock. A thread holds the lock on this object while it is executing a `synchronized` instance method of the object or executing the body of a `synchronized` statement that synchronizes on the object. A thread can also hold the lock for a `Class` object if it is executing a `synchronized static` method of that class.

This method is `final`, so it cannot be overridden by subclasses.

toString

`public String toString()`

Returns

The string representation of this object.

Description

The `toString()` method of `Object` returns a generic string representation of this object. The method returns a `String` that consists of the object's class name, an "at" sign, and the unsigned hexadecimal representation of the value returned by the object's `hashCode()` method.

Many classes override the `toString()` method to provide a string representation that is specific to that type of object. Any subclass can override the `toString()` method; the overriding method should simply return a `String` that represents the contents of the object with which it is associated.

wait

`public final native void wait() throws InterruptedException`

Throws

`IllegalMonitorStateException`

If the method is called from a thread that does not hold this object's lock.

`InterruptedException`

If another thread interrupted this thread.

Description

The `wait()` method causes a thread to wait until it is notified by another thread to stop waiting. When `wait()` is called, the thread releases its lock on this object and waits until another thread notifies it to wake up through a call to either `notify()` or `notifyAll()`. After the thread is awakened, it has to regain the lock on this object before it can resume executing.

The `wait()` method can be called only by a thread that is the current owner of this object's lock. A thread holds the lock on this object while it is executing a `synchronized` instance method of the object or executing the body of a `synchronized` statement that synchronizes on the object. A thread can also hold the lock for a `Class` object if it is executing a `synchronized` `static` method of that class.

This method is `final`, so it cannot be overridden by subclasses.

`public final native void wait(long timeout) throws InterruptedException`

Parameters

`timeout`

The maximum number of milliseconds to wait.

Throws

`IllegalMonitorStateException`

If the method is called from a thread that does not hold this object's lock.

`InterruptedException`

If another thread interrupted this thread.

Description

The `wait()` method causes a thread to wait until it is notified by another thread to stop waiting or until the specified amount of time has elapsed, whichever comes first. When `wait()` is called, the thread releases its lock on this object and waits until another thread notifies it to wake up through a call to either `notify()` or `notifyAll()`. If the thread is not notified within the specified `timeout` period, it is automatically awakened when that amount of time has elapsed. If `timeout` is zero, the thread waits indefinitely, just as if `wait()` had been called without a `timeout` argument. After the thread is awakened, it has to regain the lock on this object before it can resume executing.

The `wait()` method can be called only by a thread that is the current owner of this object's lock. A thread holds the lock on this object while it is executing a `synchronized` instance method of the object or executing the body of a `synchronized` statement that synchronizes on the object. A thread can also hold the lock for a `Class` object if it is executing a `synchronized static` method of that class.

This method is `final`, so it cannot be overridden by subclasses.

`public final native void wait(long timeout, int nanos) throws InterruptedException`

Parameters

`timeout`

The maximum number of milliseconds to wait.

`nanos`

An additional number of nanoseconds to wait.

Throws

`IllegalMonitorStateException`

If the method is called from a thread that does not hold this object's lock.

`InterruptedException`

If another thread interrupted this thread.

Description

> The `wait()` method causes a thread to wait until it is notified by another thread to stop waiting or until the specified amount of time has elapsed, whichever comes first. When `wait()` is called, the thread releases its lock on this object and waits until another thread notifies it to wake up through a call to either `notify()` or `notifyAll()`. If the thread is not notified within the specified time period, it is automatically awakened when that amount of time has elapsed. If `timeout` and `nanos` are zero, the thread waits indefinitely, just as if `wait()` had been called without any arguments. After the thread is awakened, it has to regain the lock on this object before it can resume executing.
>
> The `wait()` method can be called only by a thread that is the current owner of this object's lock. A thread holds the lock on this object while it is executing a `synchronized` instance method of the object or executing the body of a `synchronized` statement that synchronizes on the object. A thread can also hold the lock for a `Class` object if it is executing a `synchronized static` method of that class.
>
> Note that Sun's reference implementation of Java does not attempt to implement the precision implied by this method. Instead, it rounds to the nearest millisecond (unless `timeout` is `0`, in which case it rounds up to 1 millisecond) and calls `wait(long)`.
>
> This method is `final`, so it cannot be overridden by subclasses.

Protected Instance Methods

clone

`protected native Object clone() throws CloneNotSupportedException`

Returns

> A clone of this object.

Throws

> `OutOfMemoryError`
>
> > If there is not enough memory to create the new object.
>
> `CloneNotSupportedException`
>
> > If the object is of a class that does not support `clone()`.

Description

> A *clone* of an object is another object of the same type that has all of its instance variables set to the same values as the object being cloned. In other words, a clone is an exact copy of the original object.
>
> The `clone()` method of `Object` creates a new object that is a clone of this object. No constructor is used in creating the clone. The `clone()` method only clones an object if the class of that object indicates that its instances

can be cloned. A class indicates that its objects can be cloned by implementing the `Cloneable` interface.

Although array objects do not implement the `Cloneable` interface, the `clone()` method works for arrays. The clone of an array is an array that has the same number of elements as the original array, and each element in the clone array has the same value as the corresponding element in the original array. Note that if an array element contains an object reference, the clone array contains a reference to the same object, not a copy of the object.

A subclass of `Object` can override the `clone()` method in `Object` to provide any additional functionality that is needed. For example, if an object contains references to other objects, the `clone()` method should recursively call the `clone()` methods of all the referenced objects. An overriding `clone()` method can throw a `CloneNotSupportedException` to indicate that particular objects cannot be cloned.

finalize

`protected void finalize() throws Throwable`

Throws

`Throwable`

For any reason that suits an overriding implementation of this method.

Description

The `finalize()` method is called by the garbage collector when it decides that an object can never be referenced again. The method gives an object a chance to perform any cleanup operations that are necessary before it is destroyed by the garbage collector.

The `finalize()` method of `Object` does nothing. A subclass overrides the `finalize()` method to perform any necessary cleanup operations. The overriding method should call `super.finalize()` as the very last thing it does, so that any `finalize()` method in the superclass is called.

When the garbage collector calls an object's `finalize()` method, the garbage collector does not immediately destroy the object because the `finalize()` method might do something that results in a reference to the object. Thus the garbage collector waits to destroy the object until it can again prove it is safe to do so. The next time the garbage collector decides it is safe to destroy the object, it does so without calling `finalize()` again. In other words, the garbage collector never calls the `finalize()` method more than once for a particular object.

A `finalize()` method can throw any kind of exception. An exception causes the `finalize()` method to stop running. The garbage collector then catches and ignores the exception, so it has no further effect on a program.

See Also

Equality Comparison Operators; Exceptions; Object Destruction; The finalize method; String; Threads 8; Throwable

PREVIOUS | HOME | NEXT
Number | BOOK INDEX | Process

System

Name

System

Synopsis

Class Name:

`java.lang.System`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The `System` class provides access to various information about the operating system environment in which a program is running. For example, the `System` class defines variables that allow access to the standard I/O streams and methods that allow a program to run the garbage collector and stop the Java virtual machine.

All of the variables and methods in the `System` class are `static`. In other words, it is not necessary to create an instance of the `System` class in order to use its variables and methods. In fact, the `System` class does not define any `public` constructors, so it cannot be instantiated.

The `System` class supports the concept of system properties that can be queried and set. The following properties are guaranteed always to be defined:

Property Name	**Description**
`file.encoding`	The character encoding for the default locale (Java 1.1 only)
`file.encoding.pkg`	The package that contains converters between local encodings and Unicode (Java 1.1 only)
`file.separator`	File separator (`'/'` on UNIX, `' \'` on Windows)
`java.class.path`	The class path
`java.class.version`	Java class version number
`java.compiler`	The just-in-time compiler to use, if any (Java 1.1 only)
`java.home`	Java installation directory
`java.vendor`	Java vendor-specific string
`java.vendor.url`	Java vendor URL
`java.version`	Java version number
`line.separator`	Line separator(`' \n'` on UNIX, `' \r\n'` on Windows)
`os.arch`	Operating system architecture
`os.name`	Operating system name
`os.version`	Operating system version
`path.separator`	Path separator (`':'` on UNIX, `','` on Windows)
`user.dir`	User's current working directory when the properties were initialized
`user.home`	User's home directory
`user.language`	The two-letter language code of the default locale (Java 1.1 only)
`user.name`	User's account name
`user.region`	The two-letter country code of the default locale (Java 1.1 only)
`user.timezone`	The default time zone (Java 1.1 only)

Additional properties may be defined by the run-time environment. The `-D` command-line option can be used to define system properties when a program is run.

The `Runtime` class is related to the `System` class; it provides access to information about the environment in which a program is running.

Class Summary

```
public final class java.lang.System extends java.lang.Object {
    // Constants
    public static final PrintStream err;
    public static final InputStream in;
    public static final PrintStream out;
    // Class Methods
    public static void arraycopy(Object src, int srcOffset,
                                 Object dst, int dstOffset, int length);
    public static long currentTimeMillis();
    public static void exit(int status);
    public static void gc();
    public static Properties getProperties();
    public static String getProperty(String key);
    public static String getProperty(String key, String default);
    public static SecurityManager getSecurityManager();
    public static String getenv(String name);           // Deprecated in 1.1
    public static native int identityHashCode(Object x);   // New in 1.1
    public static void load(String filename);
    public static void loadLibrary(String libname);
    public static void runFinalization();
    public static void runFinalizersOnExit(boolean value); // New in 1.1
    public static void setErr(PrintStream err);            // New in 1.1
    public static void setIn(InputStream in);              // New in 1.1
    public static void setOut(PrintStream out);            // New in 1.1
    public static void setProperties(Properties props);
    public static void setSecurityManager(SecurityManager s);
}
```

Variables

err

`public static final PrintStream err`

Description

> The standard error stream. In an application environment, this variable refers to a `java.io.PrintStream` object that is associated with the standard error output for the process running the Java virtual machine. In an applet environment, the `PrintStream` is likely to be associated with a separate window, although this is not guaranteed.
>
> The value of `err` can be set using the `setErr()` method. The value of `err` can only be set if the

currenly installed `SecurityManager` does not throw a `SecurityException` when the request is made.

Prior to to Java 1.1, `err` was not `final`. It has been made `final` as of Java 1.1 because the unchecked ability to set `err` is a security hole.

in

`public static final InputStream in`

Description

The standard input stream. In an application environment, this variable refers to a `java.io.InputStream` object that is associated with the standard input for the process running the Java virtual machine.

The value of `in` can be set using the `setIn()` method. The value of `in` can only be set if the currenly installed `SecurityManager` does not throw a `SecurityException` when the request is made.

Prior to to Java 1.1, `in` was not `final`. It has been made `final` as of Java 1.1 because the unchecked ability to set `in` is a security hole.

out

`public static final PrintStream out`

Description

The standard output stream. In an application environment, this variable refers to a `java.io.PrintStream` object that is associated with the standard output for the process running the Java virtual machine. In an applet environment, the `PrintStream` is likely to be associated with a separate window, although this is not guaranteed.

`out` is the most commonly used of the three I/O streams provided by the `System` class. Even in GUI-based applications, sending output to this stream can be useful for debugging. The usual idiom for sending output to this stream is:

```
System.out.println("Some text");
```

The value of `out` can be set using the `setOut()` method. The value of `out` can only be set if the currenly installed `SecurityManager` does not throw a `SecurityException` when the request is made.

Prior to to Java 1.1, `out` was not `final`. It has been made `final` as of Java 1.1 because the unchecked ability to set `out` is a security hole.

Class Methods

arraycopy

`public static void arraycopy(Object src, int src_position, Object dst, int dst_position, int length)`

Parameters

`src`

The source array.

`src_position`

An index into the source array.

`dst`

The destination array.

`dst_position`

An index into the destination array.

`length`

The number of elements to be copied.

Throws

`ArrayIndexOutOfBoundsException`

If the values of the `src_position`, `dst_position`, and `length` arguments imply accessing either array with an index that is less than zero or an index greater than or equal to the number of elements in the array.

`ArrayStoreException`

If the type of value stored in the `src` array cannot be stored in the `dst` array.

`NullPointerException`

If `src` or `dst` is `null`.

Description

This method copies a range of array elements from the `src` array to the `dst` array. The number of elements that are copied is specified by `length`. The elements at positions `src_position` through `src_position+length-1` in `src` are copied to the positions `dst_position` through `dst_position+length-1` in `dst`, respectively.

If `src` and `dst` refer to the same array, the copying is done as if the array elements were first copied to a temporary array and then copied to the destination array.

Before this method does any copying, it performs a number of checks. If either `src` or `dst` are `null`, the method throws a `NullPointerException` and `dst` is not modified.

If any of the following conditions are true, the method throws an `ArrayStoreException`, and `dst` is not modified:

- Either `src` or `dst` refers to an object that is not an array.
- `src` and `dst` refer to arrays whose element types are different primitive types.
- `src` refers to an array that has elements that contain a primitive type, while `dst` refers to an array that has elements that contain a reference type, or vice versa.

If any of the following conditions are true, the method throws an `ArrayIndexOutOfBoundsException`, and `dst` is not modified:

- `srcOffset`, `dstOffset`, or `length` is negative.
- `srcOffset+length` is greater than `src.length()`.
- `dstOffset+length` is greater than `dst.length()`.

Otherwise, if an element in the source array being copied cannot be converted to the type of the destination array using the rules of the assignment operator, the method throws an `ArrayStoreException` when the problem occurs. Since the problem is discovered during the copy operation, the state of the `dst` array reflects the incomplete copy operation.

currentTimeMillis

```
public static native long currentTimeMillis()
```

Returns

The current time as the number of milliseconds since 00:00:00 UTC, January 1, 1970.

Description

This method returns the current time as the number of milliseconds since 00:00:00 UTC, January 1, 1970. It will not overflow until the year 292280995.

The `java.util.Date` class provides more extensive facilities for dealing with times and dates.

exit

`public static void exit(int status)`

Parameters

`status`

The exit status code to use.

Throws

`SecurityException`

If the `checkExit()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method causes the Java virtual machine to exit with the given status code. This method works by calling the `exit()` method of the current `Runtime` object. By convention, a nonzero status code indicates abnormal termination. This method never returns.

gc

`public static void gc()`

Description

This method causes the Java virtual machine to run the garbage collector in the current thread. This method works by calling the `gc()` method of the current `Runtime` object.

The garbage collector finds objects that will never be used again because there are no live references to them. After it finds these objects, the garbage collector frees the storage occupied by these objects.

The garbage collector is normally run continuously in a thread with the lowest possible priority, so that it works intermittently to reclaim storage. The `gc()` method allows a program to invoke the garbage collector explicitly when necessary.

getProperties

`public static Properties getProperties()`

Returns

A `Properties` object that contains the values of all the system properies.

Throws

`SecurityException`

If the `checkPropertiesAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method returns all of the defined system properties encapsulated in a `java.util.Properties` object. If there are no system properties currently defined, a set of default system properties is created and initialized. As discussed in the description of the `System` class, some system properties are guaranteed always to be defined.

getProperty

`public static String getProperty(String key)`

Parameters

`key`

The name of a system property.

Returns

The value of the named system property or `null` if the named property is not defined.

Throws

`SecurityException`

If the `checkPropertyAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method returns the value of the named system property. If there is no definition for the named property, the method returns `null`. If there are no system properties currently defined, a set of default system properties is created and initialized. As discussed in the description of the `System` class, some system properties are guaranteed always to be defined.

`public static String getProperty(String key, String def)`

Parameters

`key`

The name of a system property.

`def`

A default value for the property.

Returns

The value of the named system property, or the default value if the named property is not defined.

Throws

`SecurityException`

If the `checkPropertyAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method returns the value of the named system property. If there is no definition for the named property, the method returns the default value as specified by the `def` parameter. If there are no system properties currently defined, a set of default system properties is created and initialized. As discussed earlier in the description of the `System` class, some system properties are guaranteed to always be defined.

getSecurityManager

`public static SecurityManager getSecurityManager()`

Returns

A reference to the installed `SecurityManager` object or `null` if there is no `SecurityManager` object installed.

Description

This method returns a reference to the installed `SecurityManager` object. If there is no `SecurityManager` object installed, the method returns `null`.

getenv

`public static String getenv(String name)`

Availability

Deprecated as of JDK 1.1

Parameters

`name`

The name of a system-dependent environment variable.

Returns

The value of the environment variable or `null` if the variable is not defined.

Description

This method is obsolete; it always throws an error. Use `getProperties()` and the `-D` option instead.

identityHashCode

`public static native int identityHashCode(Object x)`

Availability

New as of JDK 1.1

Parameters

`x`

An object.

Returns

The identity hashcode value for the specified object.

Description

This method returns the same hashcode value for the specified object as would be returned by the default `hashCode()` method of `Object`, regardless of whether or not the object's class overrides `hashCode()`.

load

`public void load(String filename)`

Parameters

`filename`

A string that specifies the complete path of the file to be loaded.

Throws

`SecurityException`

If the `checkLink()` method of the `SecurityManager` throws a `SecurityException`.

`UnsatisfiedLinkError`

If the method is unsuccessful in loading the specified dynamically linked library.

Description

This method loads the specified dynamically linked library. This method works by calling the `load()` method of the current `Runtime` object.

loadLibrary

`public void loadLibrary(String libname)`

Parameters

`libname`

A string that specifies the name of a dynamically linked library.

Throws

`SecurityException`

If the `checkLink()` method of the `SecurityManager` throws a `SecurityException`.

`UnsatisfiedLinkError`

If the method is unsuccessful in loading the specified dynamically linked library.

Description

This method loads the specified dynamically linked library. It looks for the specified library in a platform-specific way. This method works by calling the `loadLibrary()` method of the current `Runtime` object.

runFinalization

`public static void runFinalization()`

Description

This method causes the Java virtual machine to run the `finalize()` methods of any objects in the finalization queue in the current thread. This method works by calling the `runFinalization()` method of the current `Runtime` object.

When the garbage collector discovers that there are no references to an object, it checks to see if the object has a `finalize()` method that has never been called. If the object has such a `finalize()` method, the object is placed in the finalization queue. While there is a reference to the object in the finalization queue, the object is no longer considered garbage collectable.

Normally, the objects in the finalization queue are handled by a separate finalization thread that runs continuously at a very low priority. The finalization thread removes an object from the queue and calls its `finalize()` method. As long as the `finalize()` method does not generate a reference to the object, the object again becomes available for garbage collection.

Because the finalization thread runs at a very low priority, there may be a long delay from the time that an object is put on the finalization queue until the time that its `finalize()` method is called. The runFinalization() method allows a program to run the finalize() methods explicitly. This can be useful when there is a shortage of some resource that is released by a finalize() method.

runFinalizersOnExit

public static void runFinalizersOnExit(boolean value)

Availability

New as of JDK 1.1

Parameters

value

A boolean value that specifies whether or not finalization occurs on exit.

Throws

SecurityException

If the checkExit() method of the SecurityManager throws a SecurityException.

Description

This method specifies whether or not the finalize() methods of all objects that have finalize() methods are run before the Java virtual machine exits. By default, the finalizers are not run on exit. This method works by calling the runFinalizersOnExit() method of the current Runtime object.

setErr

public static void setErr(PrintStream err)

Availability

New as of JDK 1.1

Parameters

err

A PrintStream object to use for the standard error stream.

Throws

`SecurityException`

If the `checkExec()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method sets the standard error stream to be this `PrintStream` object.

setIn

`public static void setIn(InputStream in)`

Availability

New as of JDK 1.1

Parameters

`in`

A `InputStream` object to use for the standard input stream.

Throws

`SecurityException`

If the `checkExec()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method sets the standard input stream to be this `InputStream` object.

setOut

`public static void setOut(PrintStream out)`

Availability

New as of JDK 1.1

Parameters

`out`

A `PrintStream` object to use for the standard output stream.

Throws

`SecurityException`

If the `checkExec()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method sets the standard output stream to be this `PrintStream` object.

setProperties

`public static void setProperties(Properties props)`

Parameters

`props`

A reference to a `Properties` object.

Throws

`SecurityException`

If the `checkPropertiesAccess()` method of the `SecurityManager` throws a `SecurityException`.

Description

This method replaces the current set of system property definitions with a new set of system property definitions that are encapsulated by the given `Properties` object. As discussed in the description of the `System` class, some system properties are guaranteed to always be defined.

setSecurityManager

`public static void setSecurityManager(SecurityManager s)`

Parameters

`s`

A reference to a `SecurityManager` object.

Throws

`SecurityException`

If a `SecurityManager` object has already been installed.

Description

This method installs the given `SecurityManager` object. If `s` is `null`, then no `SecurityManager` object is installed. Once a `SecurityManager` object is installed, any subsequent calls to this method throw a `SecurityException`.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

Assignment Compatibility; Errors; Exceptions; Object; Object Destruction; Process; Runtime; SecurityManager

PREVIOUS
StringBuffer

HOME
BOOK INDEX

NEXT
Thread

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

28.22 java.net.URL (JDK 1.0)

This class represents a URL (a Uniform Resource Locator) and allows the data referred to by the URL to be downloaded. A URL may be specified as a single string or with separate protocol, host, port, and file specifications. Relative URLs may also be specified with a `String` and the `URL` object that it is relative to.

`getFile()`, `getHost()`, `getPort()`, `getProtocol()`, and `getRef()` return the various portions of the URL specified by a `URL` object. `sameFile()` determines whether a `URL` object refers to the same file as this one.

The data or object referred to by a URL may be downloaded from the Internet in three ways: through a `URLConnection` created with `openConnection()`, through an `InputStream` created by `openStream()`, or through `getContent()`, which returns the URL contents directly, if an appropriate `ContentHandler` can be found.

```
public final class URL extends Object implements Serializable {
    // Public Constructors
            public URL(String protocol, String host, int port, String file) throws 
MalformedURLException;
            public URL(String protocol, String host, String file) throws 
MalformedURLException;
            public URL(String spec) throws MalformedURLException;
            public URL(URL context, String spec) throws MalformedURLException;
    // Class Methods
            public static synchronized void 
setURLStreamHandlerFactory(URLStreamHandlerFactory fac);
    // Public Instance Methods
            public boolean equals(Object obj);  // Overrides Object
            public final Object getContent() throws IOException;
            public String getFile();
            public String getHost();
            public int getPort();
            public String getProtocol();
            public String getRef();
            public int hashCode();  // Overrides Object
            public URLConnection openConnection() throws IOException;
            public final InputStream openStream() throws IOException;
            public boolean sameFile(URL other);
            public String toExternalForm();
            public String toString();  // Overrides Object
    // Protected Instance Methods
            protected void set(String protocol, String host, int port, String file, 
String ref);
}
```

Passed To:

Applet.getAudioClip(), Applet.getImage(), Applet.play(), AppletContext.getAudioClip(), AppletContext.getImage(), AppletContext.showDocument(), HttpURLConnection(), Toolkit.getImage(), URL(default.htm), URL.sameFile(), URLConnection(), URLStreamHandler.openConnection(), URLStreamHandler.parseURL(default.htm), URLStreamHandler.setURL(default.htm), URLStreamHandler.toExternalForm()

Returned By:

Applet.getCodeBase(), Applet.getDocumentBase(), AppletStub.getCodeBase(), AppletStub.getDocumentBase(), Class.getResource(), ClassLoader.getResource(), ClassLoader.getSystemResource(), URLConnection.getURL(default.htm)

Type Of:

URLConnection.url

PREVIOUS HOME NEXT
java.net.UnknownServiceException (JDK 1.0) BOOK INDEX java.net.URLConnection (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

28.26 java.net.URLStreamHandlerFactory (JDK 1.0)

This interface defines a method that creates a `URLStreamHandler` object for a specified protocol.

Normal applications never need to use or implement this interface.

```
public abstract interface URLStreamHandlerFactory {
    // Public Instance Methods
            public abstract URLStreamHandler createURLStreamHandler(String protocol);
}
```

Passed To:

URL.setURLStreamHandlerFactory()

13. Java Syntax

Contents:

13.1 Primitive Data Types

Java supports a complete set of primitive data types, listed in Table 13.1. In Java, the size of each type is defined by the language, and is *not* implementation dependent, as it is in C.

Table 13.1: Java Primitive Data Types

				Min Value
Type	**Contains**	**Default**	**Size**	**Max Value**
boolean	`true` or `false`	`false`	1 bit	N.A.
				N.A.
char	Unicode character	`\u0000`	16 bits	`\u0000`
				`\uFFFF`
byte	signed integer	`0`	8 bits	-128
				127
short	signed integer	`0`	16 bits	-32768
				32767

int	signed integer	`0`	32 bits	-2147483648
				2147483647
long	signed integer	`0`	64 bits	-9223372036854775808
				9223372036854775807
float	IEEE 754	`0.0`	32 bits	+/-3.40282347E+38
	floating-point			+/-1.40239846E-45
double	IEEE 754	`0.0`	64 bits	+/-1.79769313486231570E+308
	floating-point			+/-4.94065645841246544E-324

PREVIOUS
Invoking a Named Method

HOME
BOOK INDEX

NEXT
Character Escape Sequences

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

String

Name

String

Synopsis

Class Name:

`java.lang.String`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

`java.io.Serializable`

Availability:

JDK 1.0 or later

Description

The `String` class represents sequences of characters. Once a `String` object is created, it is immutable. In other words, the sequence of characters that a `String` represents cannot be changed after it is created. The `StringBuffer` class, on the other hand, represents a sequence of characters that can be changed. `StringBuffer` objects are used to perform computations on `String` objects.

The `String` class includes a number of utility methods, such as methods for fetching individual characters or ranges of contiguous characters, for translating characters to upper- or lowercase, for searching strings, and for parsing numeric values in strings.

`String` literals are compiled into `String` objects. Where a `String` literal appears in an expression, the compiled code contains a `String` object. If `s` is declared as `String`, the following two expressions are identical:

```
s.equals("ABC")
"ABC".equals(s)
```

The string concatenation operator implicitly creates `String` objects.

Class Summary

```
public final class java.lang.String extends java.lang.Object {
    // Constructors
    public String();
    public String(byte[] bytes);                          // New in 1.1
    public String(byte[] bytes, String enc);              // New in 1.1
    public String(byte[] bytes, int offset, int length);  // New in 1.1
    public String(byte[] bytes, int offset,
                  int length, String enc);                // New in 1.1
    public String(byte[] lowbytes, int hibyte);        // Deprecated in 1.1
    public String(byte[] lowbytes, int hibyte,
                  int offset, int count);              // Deprecated in 1.1
    public String(char[] value);
    public String(char[] value, int offset, int;
    public String(String value);
    public String(StringBuffer buffer);
    // Class Methods
    public static String copyValueOf(char data[]);
    public static String copyValueOf(char data[], int offset, int count);
    public static String valueOf(boolean b);
    public static String valueOf(char c);
    public static String valueOf(char[] data);
    public static String valueOf(char[] data, int offset, int count);
    public static String valueOf(double d);
    public static String valueOf(float f);
    public static String valueOf(int i);
    public static String valueOf(long l);
    public static String valueOf(Object obj);
    // Instance Methods
    public char charAt(int index);
    public int compareTo(String anotherString);
    public String concat(String str);
    public boolean endsWith(String suffix);
    public boolean equals(Object anObject);
    public boolean equalsIgnoreCase(String anotherString);
    public byte[] getBytes();                               // New in 1.1
    public byte[] getBytes(String enc);                     // New in 1.1
    public void getBytes(int srcBegin, int srcEnd,
                         byte[] dst, int dstBegin);     // Deprecated in 1.1
    public void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin);
    public int hashCode();
    public int indexOf(int ch);
    public int indexOf(int ch, int fromIndex);
    public int indexOf(String str);
    public int indexOf(String str, int fromIndex);
    public native String intern();
    public int lastIndexOf(int ch);
    public int lastIndexOf(int ch, int fromIndex);
    public int lastIndexOf(String str);
    public int lastIndexOf(String str, int fromIndex;
    public int length();
    public boolean regionMatches(boolean ignoreCase, int toffset,
                                 String other, int ooffset, int len);
    public boolean regionMatches(int toffset, String other,
                                 int ooffset, int len);
    public String replace(char oldChar, char newChar);
    public boolean startsWith(String prefix);
    public boolean startsWith(String prefix, int toffset);
    public String substring(int beginIndex);
    public String substring(int beginIndex, int endIndex);
    public char[] toCharArray();
    public String toLowerCase();
    public String toLowerCase(Locale locale);               // New in 1.1
    public String toString();
    public String toUpperCase();
    public String toUpperCase(Locale locale);               // New in 1.1
    public String trim();
}
```

Constructors

String

`public String()`

Description

Creates a new `String` object that represents the empty string (i.e., a string with zero characters).

`public String(byte[] bytes)`

Availability

New as of JDK 1.1

Parameters

`bytes`

An array of `byte` values.

Description

Creates a new `String` object that represents the sequence of characters stored in the given `bytes` array. The bytes in the array are converted to characters using the system's default character encoding scheme.

`public String(byte[] bytes, String enc)`

Availability

New as of JDK 1.1

Parameters

`bytes`

An array of `byte` values.

`enc`

The name of an encoding scheme.

Throws

`UnsupportedEncodingException`

If `enc` is not a supported encoding scheme.

Description

Creates a new `String` object that represents the sequence of characters stored in the given `bytes` array. The bytes in the array are converted to characters using the specified character encoding scheme.

`public String(byte[] bytes, int offset, int length)`

Availability

New as of JDK 1.1

Parameters

`bytes`

An array of `byte` values.

`offset`

An offset into the array.

`length`

The number of bytes to be included.

Throws

`StringIndexOutOfBoundsException`

If `offset` or `length` indexes an element that is outside the bounds of the `bytes` array.

Description

Creates a new `String` object that represents the sequence of characters stored in the specified portion of the given `bytes` array. The bytes in the array are converted to characters using the system's default character encoding scheme.

`public String(byte[] bytes, int offset, int length, String enc)`

Availability

New as of JDK 1.1

Parameters

`bytes`

An array of `byte` values.

`offset`

An offset into the array.

`length`

The number of bytes to be included.

`enc`

The name of an encoding scheme.

Throws

`StringIndexOutOfBoundsException`

If `offset` or `length` indexes an element that is outside the bounds of the `bytes` array.

`UnsupportedEncodingException`

If `enc` is not a supported encoding scheme.

Description

Creates a new `String` object that represents the sequence of characters stored in the specified portion of the given `bytes` array. The bytes in the array are converted to characters using the specified character encoding scheme.

`public String(byte[] lowbytes, int hibyte)`

Availability

Deprecated as of JDK 1.1

Parameters

`lowbytes`

An array of `byte` values.

`hibyte`

The value to be put in the high-order byte of each 16-bit character.

Description

Creates a new `String` object that represents the sequence of characters stored in the given `lowbytes` array. The type of the array elements is `byte`, which is an 8-bit data type, so each element must be converted to a `char`, which is a 16-bit data type. The value of the `hibyte` argument is used to provide the value of the high-order byte when the `byte` values in the array are converted to `char` values.

More specifically, for each element `i` in the array `lowbytes`, the character at position `i` in the created `String` object is:

```
((hibyte & 0xff)<<8) | lowbytes[i]
```

This method is deprecated as of JDK 1.1 because it does not convert bytes into characters properly. You should instead use one of the constructors that takes a specific character encoding argument or that uses the default encoding.

`public String(byte[] lowbytes, int hibyte, int offset, int count)`

Availability

Deprecated as of JDK 1.1

Parameters

`lowbytes`

An array of `byte` values.

`hibyte`

The value to be put in the high-order byte of each 16-bit character.

`offset`

An offset into the array.

`count`

The number of bytes from the array to be included in the string.

Throws

`StringIndexOutOfBoundsException`

If `offset` or `count` indexes an element that is outside the bounds of the `lowbytes` array.

Description

Creates a new `String` object that represents the sequence of characters stored in the specified portion of the `lowbytes` array. That is, the portion of the array that starts at `offset` elements from the beginning of the array and is `count` elements long.

The type of the array elements is `byte`, which is an 8-bit data type, so each element must be converted to a `char`, which is a 16-bit data type. The value of the `hibyte` argument is used to provide the value of the high-order byte when the `byte` values in the array are converted to `char` values.

More specifically, for each element `i` in the array `lowbytes` that is included in the `String` object, the character at position `i` in the created `String` is:

```
 ((hibyte & 0xff)<<8) | lowbytes[I]
```

This method is deprecated as of JDK 1.1 because it does not convert bytes into characters properly. You should instead use one of the constructors that takes a specific character encoding argument or that uses the default encoding.

`public String(char[] value)`

Parameters

`value`

An array of `char` values.

Description

Creates a new `String` object that represents the sequence of characters stored in the given array.

`public String(char[] value, int offset, int count)`

Parameters

`value`

An array of `char` values.

`offset`

An offset into the array.

`count`

The number of characters from the array to be included in the string.

Throws

`StringIndexOutOfBoundsException`

If `offset` or `count` indexes an element that is outside the bounds of the `value` array.

Description

Creates a new `String` object that represents the sequence of characters stored in the specified portion of the given array. That is, the portion of the given array that starts at `offset` elements from the beginning of the array and is `count` elements long.

`public String(String value)`

Parameters

`value`

A `String` object.

Description

Creates a new `String` object that represents the same sequence of characters as the given `String` object.

`public String(StringBuffer value)`

Parameters

`value`

A `StringBuffer` object.

Description

Creates a new `String` object that represents the same sequence of characters as the given object.

Class Methods

copyValueOf

`public static String copyValueOf(char data[])`

Parameters

`data`

An array of `char` values.

Returns

A new `String` object that represents the sequence of characters stored in the given array.

Description

This method returns a new `String` object that represents the character sequence contained in the given array. The `String` object produced by this method is guaranteed not to refer to the given array, but instead to use a copy. Because the `String` object uses a copy of the array, subsequent changes to the array do not change the contents of this `String` object.

This method is now obsolete. The same result can be obtained using the `valueOf()` method that takes an array of `char` values.

`public static String copyValueOf(char data[], int offset, int count)`

Parameters

`data`

An array of `char` values.

`offset`

An offset into the array.

`count`

The number of characters from the array to be included in the string.

Returns

A new `String` object that represents the sequence of characters stored in the specified portion of the given array.

Throws

`StringIndexOutOfBoundsException`

If `offset` or `count` indexes an element that is outside the bounds of the `data` array.

Description

This method returns a new `String` object that represents the character sequence contained in the specified portion of the given array. That is, the portion of the given array that starts at `offset` elements from the beginning of the array and is `count` elements long. The `String` object produced by this method is guaranteed not to refer to the given array, but instead to use a copy. Because the `String` object uses a copy of the array, subsequent changes to the array do not change the contents of this `String` object.

This method is obsolete. The same result can be obtained by using the `valueOf()` method that takes an array of `char` values, an offset, and a count.

valueOf

`public static String valueOf(boolean b)`

Parameters

`b`

A `boolean` value.

Returns

A new `String` object that contains `'true'` if `b` is `true` or `'false'` if `b` is `false`.

Description

This method returns a string representation of a `boolean` value. In other words, it returns `'true'` if `b` is `true` or `'false'` if `b` is `false`.

`public static String valueOf(char c)`

Parameters

`c`

A `char` value.

Returns

A new `String` object that contains just the given character.

Description

This method returns a string representation of a `char` value. In other words, it returns a `String` object that contains just the given character.

`public static String valueOf(char[] data)`

Parameters

`data`

An array of `char` values.

Returns

A new `String` object that contains the sequence of characters stored in the given array.

Description

This method returns a string representation of an array of `char` values. In other words, it returns a `String` object that contains the sequence of characters stored in the given array.

`public static String valueOf(char[] data, int offset, int count)`

Parameters

`data`

An array of `char` values.

`offset`

An offset into the array.

`count`

The number of characters from the array to be included in the string.

Returns

A new `String` object that contains the sequence of characters stored in the specified portion of the given array.

Throws

`StringIndexOutOfBoundsException`

If `offset` or `count` indexes an element that is outside the bounds of the `data` array.

Description

This method returns a string representation of the specified portion of an array of char values. In other words, it returns a `String` object that contains the sequence of characters in the given array that starts `offset` elements from the beginning of the array and is `count` elements long.

`public static String valueOf(double d)`

Parameters

`d`

A `double` value.

Returns

A new `String` object that contains a string representation of the given `double` value.

Description

This method returns a string representation of a `double` value. In other words, it returns the `String` object returned by `Double.toString(d)`.

`public static String valueOf(float f)`

Parameters

`f`

A `float` value.

Returns

A new `String` object that contains a string representation of the given `float` value.

Description

This method returns a string representation of a `float` value. In other words, it returns the `String` object returned by `Float.toString(f)`.

`public static String valueOf(int i)`

Parameters

`i`

An `int` value.

Returns

A new `String` object that contains a string representation of the given `int` value.

Description

This method returns a string representation of an `int` value. In other words, it returns the `String` object returned by `Integer.toString(i)`.

`public static String valueOf(long l)`

Parameters

`l`

A `long` value.

Returns

A new `String` object that contains a string representation of the given `long` value.

Description

This method returns a string representation of a `long` value. In other words, it returns the `String` object returned by `Long.toString(l)`.

`public static String valueOf (Object obj)`

Parameters

`obj`

A reference to an object.

Returns

A new `String` that contains a string representation of the given object.

Description

This method returns a string representation of the given object. If `obj` is `null`, the method returns`'null'`. Otherwise, the method returns the `String` object returned by the `toString()` method of the object.

Instance Methods

charAt

`public char charAt(int index)`

Parameters

`index`

An index into the string.

Returns

The character at the specified position in this string.

Throws

`StringIndexOutOfBoundsException`

If `index` is less than zero or greater than or equal to the length of the string.

Description

This method returns the character at the specified position in the `String` object; the first character in the string is at position `0`.

compareTo

`public int compareTo(String anotherString)`

Parameters

`anotherString`

The `String` object to be compared.

Returns

A positive value if this string is greater than `anotherString`, `0` if the two strings are the same, or a negative value if this string is less than `anotherString`.

Description

This method lexicographically compares this `String` object to `anotherString`.

Here is how the comparison works: the two `String` objects are compared character-by-character, starting at index `0` and continuing until a position is found in which the two strings contain different characters or until all of the characters in the shorter string have been compared. If the characters at `k` are different, the method returns:

```
this.charAt(k)-anotherString.charAt(k)
```

Otherwise, the comparison is based on the lengths of the strings and the method returns:

```
this.length()-anotherString.length()
```

concat

`public String concat(String str)`

Parameters

`str`

The `String` object to be concatenated.

Returns

A new `String` object that contains the character sequences of this string and `str` concatenated together.

Description

This method returns a new `String` object that concatenates the characters from the argument string `str` onto the characters from this `String` object. Although this is a good way to concatenate two strings, concatenating more than two strings can be done more efficiently using a `StringBuffer` object.

endsWith

`public boolean endsWith(String suffix)`

Parameters

`suffix`

The `String` object suffix to be tested.

Returns

`true` if this string ends with the sequence of characters specified by `suffix`; otherwise `false`.

Description

This method determines whether or not this `String` object ends with the specified `suffix`.

equals

`public boolean equals(Object anObject)`

Parameters

`anObject`

The `Object` to be compared.

Returns

`true` if the objects are equal; `false` if they are not.

Overrides

`Object.equals()`

Description

This method returns `true` if `anObject` is an instance of `String` and it contains the same sequence of characters as this `String` object.

Note the difference between this method and the `==` operator, which only returns `true` if both of its arguments are references to the same object.

equalsIgnoreCase

`public boolean equalsIgnoreCase(String anotherString)`

Parameters

`anotherString`

The `String` object to be compared.

Returns

`true` if the strings are equal, ignoring case; otherwise `false`.

Description

This method determines whether or not this `String` object contains the same sequence of characters, ignoring case, as `anotherString`. More specifically, corresponding characters in the two strings are considered equal if any of the following conditions are true:

- The two characters compare as equal using the `==` operator.
- The `Character.toUppercase()` method returns the same result for both characters.
- The `Character.toLowercase()` method returns the same result for both characters.

getBytes

`public byte[] getBytes()`

Availability

New as of JDK 1.1

Returns

A `byte` array that contains the characters of this `String`.

Description

This method converts the characters in this `String` object to an array of `byte` values. The characters in the string are converted to bytes using the system's default character encoding scheme.

`public byte[] getBytes(String enc)`

Availability

New as of JDK 1.1

Parameters

`enc`

The name of an encoding scheme.

Returns

A `byte` array that contains the characters of this `String`.

Throws

`UnsupportedEncodingException`

If `enc` is not a supported encoding scheme.

Description

This method converts the characters in this `String` object to an array of `byte` values. The characters in the string are converted to bytes using the specified character encoding scheme.

`public void getBytes(int srcBegin, int srcEnd, byte[] dst, int dstBegin)`

Availability

Deprecated as of JDK 1.1

Parameters

`srcBegin`

The index of the first character to be copied.

`srcEnd`

The index after the last character to be copied.

`dst`

The destination `byte` array.

`dstBegin`

An offset into the destination array.

Throws

`StringIndexOutOfBoundsException`

If `srcBegin`, `srcEnd`, or `dstBegin` is out of range.

Description

This method copies the low-order byte of each character in the specified range of this `String` object to the given array of `byte` values. More specifically, the first character to be copied is at index `srcBegin`; the last character to be copied is at index `srcEnd-1`. The low-order bytes of these characters are copied into `dst`, starting at index `dstBegin` and ending at index:

```
dstBegin + (srcEnd-srcBegin) - 1
```

This method is deprecated as of JDK 1.1 because it does not convert characters into bytes properly. You should instead use the `getBytes()` method that takes a specific character encoding argument or the one that uses the default encoding.

getChars

```
public void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)
```

Parameters

`srcBegin`

The index of the first character to be copied.

`srcEnd`

The index after the last character to be copied.

`dst`

The destination `char` array.

`dstBegin`

An offset into the destination array.

Throws

`StringIndexOutOfBoundsException`

If `srcBegin`, `srcEnd`, or `dstBegin` is out of range.

Description

This method copies each character in the specified range of this `String` object to the given array of `char` values. More specifically, the first character to be copied is at index `srcBegin`; the last character to be copied is at index `srcEnd-1`. These characters are copied into `dst`, starting at index `dstBegin` and ending at index:

```
dstBegin + (srcEnd-srcBegin) - 1
```

hashCode

`public int hashCode()`

Returns

A hashcode based on the sequence of characters in this string.

Overrides

`Object.hashCode()`

Description

This method returns a hashcode based on the sequence of characters this `String` object represents.

More specifically, one of two algorithms is used to compute a hash code for the string, depending on its length. If *n* is the length of the string and *S_i* is the character at position *i* in the string, then if *n* = 15 the method returns:

If *n* > 15, the method returns:

indexOf

`public int indexOf(int ch)`

Parameters

`ch`

A `char` value.

Returns

The index of the first occurrence of the given character in this string or `-1` if the character does not occur.

Description

This method returns the index of the first occurrence of the given character in this `String` object. If there is no such occurrence, the method returns the value `-1`.

`public int indexOf(int ch, int fromIndex)`

Parameters

`ch`

A `char` value.

`fromIndex`

The index where the search is to begin.

Returns

The index of the first occurrence of the given character in this string after `fromIndex` or `-1` if the character does not occur.

Description

This method returns the index of the first occurrence of the given character in this `String` object after ignoring the first `fromIndex` characters. If there is no such occurrence, the method returns the value `-1`.

`public int indexOf(String str)`

Parameters

`str`

A `String` object.

Returns

The index of the first occurrence of `str` in this string or `-1` if the substring does not occur.

Description

This method returns the index of the first character of the first occurrence of the substring `str` in this `String` object. If there is no such occurrence, the method returns the value `-1`.

`public int indexOf(String str, int fromIndex)`

Parameters

`str`

A `String` object.

`fromIndex`

The index where the search is to begin.

Returns

The index of the first occurrence of `str` in this string after `fromIndex` or `-1` if the substring does not occur.

Description

This method returns the index of the first character of the first occurrence of the substring `str` in this `String` object after ignoring the first `fromIndex` characters. If there is no such occurrence, the method returns the value `-1`.

intern

`public native String intern()`

Returns

A `String` object that is guaranteed to be the same object for every `String` that contains the same character sequence.

Description

This method returns a canonical representation for this `String` object. The returned `String` object is guaranteed to be the same `String` object for every `String` object that contains the same character sequence. In other words, if:

```
s1.equals(s2)
```

then:

```
s1.intern() == s2.intern()
```

The `intern`() method is used by the Java environment to ensure that `String` literals and constant-value `String` expressions that contain the same sequence of characters are all represented by a single `String` object.

lastIndexOf

`public int lastIndexOf(int ch)`

Parameters

`ch`

A `char` value.

Returns

The index of the last occurrence of the given character in this string or `-1` if the character does not occur.

Description

This method returns the index of the last occurrence of the given character in this `String` object. If there is no such occurrence, the method returns the value `-1`.

`public int lastIndexOf(int ch, int fromIndex)`

Parameters

`ch`

A `char` value.

`fromIndex`

The index where the search is to begin.

Returns

The index of the last occurrence of the given character in this string after `fromIndex` or `-1` if the character does not occur.

Description

This method returns the index of the last occurrence of the given character in this `String` object after ignoring the first `fromIndex` characters. If there is no such occurrence, the method returns the value `-1`.

`public int lastIndexOf(String str)`

Parameters

`str`

A `String` object.

Returns

The index of the last occurrence of `str` in this string or `-1` if the substring does not occur.

Description

This method returns the index of the first character of the last occurrence of the substring `str` in this `String` object. If there is no such occurrence, the method returns the value `-1`.

`public int lastIndexOf(String str, int fromIndex)`

Parameters

`str`

A `String` object.

`fromIndex`

The index where the search is to begin.

Returns

The index of the last occurrence of `str` in this string after `fromIndex` or `-1` if the substring does not occur.

Description

This method returns the index of the first character of the last occurrence of the substring `str` in this `String` object after ignoring the first `fromIndex` characters. If there is no such occurrence, the method returns the value `-1`.

length

`public int length()`

Returns

The length of the character sequence represented by this string.

Description

This method returns the number of characters in the character sequence represented by this `String` object.

regionMatches

`public boolean regionMatches(int toffset, String other, int ooffset, int len)`

Parameters

`toffset`

The index of the first character in this string.

`other`

The `String` object to be used in the comparison.

`ooffset`

The index of the first character in `other`.

`len`

The length of the sub-sequences to be compared.

Returns

`true` if the sub-sequences are identical; otherwise `false`.

Description

This method determines whether or not the specified sub-sequences in this `String` object and `other` are identical. The method returns false if `toffset` is negative, if `ooffset` is negative, if `toffset+len` is greater than the length of this string, or if `ooffset+len` is greater than the length of `other`. Otherwise, the method returns `true` if for all nonnegative integers `k` less than `len` it is true that:

```
this.charAt(toffset+k) == other.charAt(ooffset+k)
```

public boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)

Parameters

`ignoreCase`

A `boolean` value that indicates whether case should be ignored.

`toffset`

The index of the first character in this string.

`other`

The `String` object to be used in the comparison.

`ooffset`

The index of the first character in `other`.

`len`

The length of the sub-sequences to be compared.

Returns

`true` if the sub-sequences are identical; otherwise `false`. The `ignoreCase` argument controls whether or not case is ignored in the comparison.

Description

This method determines whether or not the specified sub-sequences in this `String` object and `other` are identical. The method returns false if `toffset` is negative, if `ooffset` is negative, if `toffset+len` is greater than the length of this string, or if `ooffset+len` is greater than the length of `other`. Otherwise, if `ignoreCase` is `false`, the method returns `true` if for all nonnegative integers `k` less than `len` it is true that:

```
this.charAt(toffset+k) == other.charAt(ooffset+k)
```

If `ignoreCase` is `true`, the method returns `true` if for all nonnegative integers `k` less than `len` it is true that:

```
Character.toLowerCase(this.charAt(toffset+k))
== Character.toLowerCase(other.charAt(ooffset+k))
```

or:

```
Character.toUpperCase(this.charAt(toffset+k))
== Character.toUpperCase(other.charAt(ooffset+k))
```

replace

`public String replace(char oldChar, char newChar)`

Parameters

`oldChar`

The character to be replaced.

`newChar`

The replacement character.

Returns

A new `String` object that results from replacing every occurrence of `oldChar` in the string with `newChar`.

Description

This method returns a new `String` object that results from replacing every occurrence of `oldChar` in this `String` object with `newChar`. If there are no occurrences of `oldChar`, the method simply returns this `String` object.

startsWith

`public boolean startsWith(String prefix)`

Parameters

`prefix`

The `String` object prefix to be tested.

Returns

`true` if this string begins with the sequence of characters specified by `prefix`; otherwise `false`.

Description

This method determines whether or not this `String` object begins with the specified `prefix`.

`public boolean startsWith(String prefix, int toffset)`

Parameters

`prefix`

The `String` object prefix to be tested.

`toffset`

The index where the search is to begin.

Returns

`true` if this string contains the sequence of characters specified by `prefix` starting at the index `toffset`; otherwise `false`.

Description

This method determines whether or not this `String` object contains the specified `prefix` at the index specified by `toffset`.

substring

`public String substring(int beginIndex)`

Parameters

`beginIndex`

The index of the first character in the substring.

Returns

A new `String` object that contains the sub-sequence of this string that starts at `beginIndex` and extends to the end of the string.

Throws

`StringIndexOutOfBoundsException`

If `beginIndex` is less than zero or greater than or equal to the length of the string.

Description

This method returns a new `String` object that represents a sub-sequence of this `String` object. The sub-sequence consists of the characters starting at `beginIndex` and extending through the end of this `String` object.

`public String substring(int beginIndex, int endIndex)`

Parameters

`beginIndex`

The index of the first character in the substring.

`endIndex`

The index after the last character in the substring.

Returns

A new `String` object that contains the sub-sequence of this string that starts at `beginIndex` and extends to the character at `endindex-1`.

Throws

`StringIndexOutOfBoundsException`

If `beginIndex` or `endIndex` is less than zero or greater than or equal to the length of the string.

Description

This method returns a new `String` object that represents a sub-sequence of this `String` object. The sub-sequence consists of the characters starting at `beginIndex` and extending through `endIndex-1` of this `String` object.

toCharArray

`public char[] toCharArray()`

Returns

A new `char` array that contains the same sequence of characters as this string.

Description

This method returns a new `char` array that contains the same sequence of characters as this `String`object. The length of the array is the same as the length of this `String` object.

toLowerCase

`public String toLowerCase()`

Returns

A new `String` object that contains the characters of this string converted to lowercase.

Description

This method returns a new `String` that represents a character sequence of the same length as this `String` object, but with each character replaced by its lowercase equivalent if it has one. If no character in the string has a lowercase equivalent, the method returns this `String` object.

`public String toLowerCase(Locale locale)`

Availability

New as of JDK 1.1

Parameters

`locale`

The `Locale` to use.

Returns

A new `String` object that contains the characters of this string converted to lowercase using the rules of the specified locale.

Description

This method returns a new `String` that represents a character sequence of the same length as this `String` object, but with each character replaced by its lowercase equivalent if it has one according to the rules of the specified locale. If no character in the string has a lowercase equivalent, the method returns this `String` object.

toString

`public String toString()`

Returns

This `String` object.

Overrides

`Object.toString()`

Description

This method returns this `String` object.

toUpperCase

`public String toUpperCase()`

Returns

A new `String` object that contains the characters of this string converted to uppercase.

Description

This method returns a new `String` that represents a character sequence of the same length as this `String` object, but with each character replaced by its uppercase equivalent if it has one. If no character in the string has an uppercase equivalent, the method returns this `String` object.

`public String toUpperCase(Locale locale)`

Availability

New as of JDK 1.1

Parameters

`locale`

The `Locale` to use.

Returns

A new `String` object that contains the characters of this string converted to uppercase using the rules of the specified locale.

Description

This method returns a new `String` that represents a character sequence of the same length as this `String` object, but with each character replaced by its uppercase equivalent if it has one according to the rules of the specified locale. If no character in the string has an uppercase equivalent, the method returns this `String` object.

trim

`public String trim()`

Returns

A new `String` object that represents the same character sequence as this string, but with leading and trailing whitespace and control characters removed.

Description

If the first and last character in this `String` object are greater than `'\u0020'` (the space character), the method returns this `String` object. Otherwise, the method returns a new `String` object that contains the same character sequence as this `String` object, but with leading and trailing characters that are less than `'\u0020''` removed.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`finalize()`	`Object`
`getClass()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`
`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

Character; Class; Double; Exceptions; Float; Integer; Long; Object; StringBuffer; String Concatenation Operator +; String literals

StringBuffer

Name

StringBuffer

Synopsis

Class Name:

```
java.lang.StringBuffer
```

Superclass:

```
java.lang.Object
```

Immediate Subclasses:

None

Interfaces Implemented:

```
java.io.Serializable
```

Availability:

JDK 1.0 or later

Description

The `StringBuffer` class represents a variable-length sequence of characters. `StringBuffer` objects are used in computations that involve creating new `String` objects. The `StringBuffer` class provides a number of utility methods for working with StringBuffer objects, including append() and insert() methods that add characters to a StringBuffer and methods that fetch the contents of StringBuffer objects.

When a StringBuffer object is created, the constructor determines the initial contents and capacity of the StringBuffer. The capacity of a StringBuffer is the number of characters that its internal data structure can hold. This is distinct from the length of the contents of a StringBuffer, which is the number of characters that are actually stored in the StringBuffer object. The capacity of a StringBuffer can vary. When a StringBuffer object is asked to hold more characters than its current capacity allows, the StringBuffer enlarges its internal data structure. However, it is more costly in terms of execution time and memory when a StringBuffer has to repeatedly increase its capacity than when a StringBuffer object is created with sufficient capacity.

Because the intended use of StringBuffer objects involves modifying their contents, all methods of the StringBuffer class that modify StringBuffer objects are synchronized. This means that is it safe for multiple threads to try to modify a StringBuffer object at the same time.

StringBuffer objects are used implicitly by the string concatenation operator. Consider the following code:

```
String s, s1, s2;
s = s1 + s2;
```

To compute the string concatenation, the Java compiler generates code like:

```
s = new StringBuffer().append(s1).append(s2).toString();
```

Class Summary

```
public class java.lang.StringBuffer extends java.lang.Object {
    // Constructors
    public StringBuffer();
    public StringBuffer(int length);
    public StringBuffer(String str);
    // Instance Methods
    public StringBuffer append(boolean b);
    public synchronized StringBuffer append(char c);
    public synchronized StringBuffer append(char[] str);
    public synchronized StringBuffer append(char[] str, int offset, int len);
    public StringBuffer append(double d);
    public StringBuffer append(float f);
    public StringBuffer append(int i);
    public StringBuffer append(long l);
    public synchronized StringBuffer append(Object obj);
    public synchronized StringBuffer append(String str);
    public int capacity();
    public synchronized char charAt(int index);
    public synchronized void ensureCapacity(int minimumCapacity);
    public synchronized void getChars(int srcBegin, int srcEnd,
                             char[] dst, int dstBegin);
    public StringBuffer insert(int offset, boolean b);
    public synchronized StringBuffer insert(int offset, char c);
    public synchronized StringBuffer insert(int offset, char[] str);
    public StringBuffer insert(int offset, double d);
    public StringBuffer insert(int offset, float f);
    public StringBuffer insert(int offset, int i);
    public StringBuffer insert(int offset, long l);
    public synchronized StringBuffer insert(int offset, Object obj);
    public synchronized StringBuffer insert(int offset, String str);
    public int length();
    public synchronized StringBuffer reverse();
    public synchronized void setCharAt(int index, char ch);
    public synchronized void setLength(int newLength);
    public String toString();
}
```

Constructors

StringBuffer

`public StringBuffer()`

Description

> Creates a `StringBuffer` object that does not contain any characters and has a capacity of 16 characters.

`public StringBuffer(int capacity)`

Parameters

> `capacity`
>
> > The initial capacity of this `StringBufffer` object.

Throws

> `NegativeArraySizeException`
>
> > If `capacity` is negative.

Description

Creates a `StringBuffer` object that does not contain any characters and has the specified capacity.

`public StringBuffer(String str)`

Parameters

`str`

A `String` object.

Description

Creates a `StringBuffer` object that contains the same sequence of characters as the given `String` object and has a capacity 16 greater than the length of the `String`.

Instance Methods

append

`public StringBuffer append(boolean b)`

Parameters

`b`

A `boolean` value.

Returns

This `StringBuffer` object.

Description

This method appends either `"true"` or `"false"` to the end of the sequence of characters stored in ths `StringBuffer` object, depending on the value of `b`.

`public synchronized StringBuffer append(char c)`

Parameters

`c`

A `char` value.

Returns

This `StringBuffer` object.

Description

This method appends the given character to the end of the sequence of characters stored in this `StringBuffer` object.

`public synchronized StringBuffer append(char str[])`

Parameters

`str`

An array of `char` values.

Returns

This `StringBuffer` object.

Description

This method appends the characters in the given array to the end of the sequence of characters stored in this `StringBuffer` object.

`public synchronized StringBuffer append(char str[], int offset, int len)`

Parameters

`str`

An array of `char` values.

`offset`

An offset into the array.

`len`

The number of characters from the array to be appended.

Returns

This `StringBuffer` object.

Throws

`StringIndexOutOfBoundsException`

If `offset` or `len` are out of range.

Description

This method appends the specified portion of the given array to the end of the character sequence stored in this `StringBuffer` object. The portion of the array that is appended starts `offset` elements from the beginning of the array and is `len` elements long.

`public StringBuffer append(double d)`

Parameters

`d`

A `double` value.

Returns

This `StringBuffer` object.

Description

This method converts the given `double` value to a string using `Double.toString(d)` and appends the resulting string to the end of the sequence of characters stored in this `StringBuffer` object.

`public StringBuffer append(float f)`

Parameters

`f`

A `float` value.

Returns

This `StringBuffer` object.

Description

This method converts the given `float` value to a string using `Float.toString(f)` and appends the resulting string to the end of the sequence of characters stored in this `StringBuffer` object.

`public StringBuffer append(int i)`

Parameters

`i`

An `int` value.

Returns

This `StringBuffer` object.

Description

This method converts the given `int` value to a string using `Integer.toString(i)` and appends the resulting string to the end of the sequence of characters stored in this `StringBuffer` object.

`public StringBuffer append(long l)`

Parameters

`l`

A `long` value.

Returns

This `StringBuffer` object.

Description

This method converts the given `long` value to a string using `Long.toString(l)` and appends the resulting string to the end of the sequence of characters stored in this `StringBuffer` object.

`public synchronized StringBuffer append(Object obj)`

Parameters

`obj`

A reference to an object.

Returns

This `StringBuffer` object.

Description

This method gets the string representation of the given object by calling `String.valueOf(obj)` and appends the resulting string to the end of the character sequence stored in this `StringBuffer` object.

`public synchronized StringBuffer append(String str)`

Parameters

`str`

A `String` object.

Returns

This `StringBuffer` object.

Description

This method appends the sequence of characters represented by the given `String` to the characters in this `StringBuffer` object. If `str` is `null`, the string `"null"` is appended.

capacity

`public int capacity()`

Returns

The capacity of this `StringBuffer` object.

Description

This method returns the current capacity of this object. The capacity of a `StringBuffer` object is the number of characters that its internal data structure can hold. A `StringBuffer` object automatically increases its capacity when it is asked to hold more characters than its current capacity allows.

charAt

`public synchronized char charAt(int index)`

Parameters

index

An index into the StringBuffer.

Returns

The character stored at the specified position in this StringBuffer object.

Throws

StringIndexOutOfBoundsException

If index is less than zero or greater than or equal to the length of the StringBuffer object.

Description

This method returns the character at the specified position in the StringBuffer object. The first character in the StringBuffer is at index 0.

ensureCapacity

public synchronized void ensureCapacity(int minimumCapacity)

Parameters

minimumCapacity

The minimum desired capacity.

Description

This method ensures that the capacity of this StringBuffer object is at least the specified number of characters. If necessary, the capacity of this object is increased to the greater of minimumCapacity or double its current capacity plus two.

It is more efficient to ensure that the capacity of a StringBuffer object is sufficient to hold all of the additions that will be made to its contents, rather than let the StringBuffer increase its capacity in multiple increments.

getChars

public synchronized void getChars(int srcBegin, int srcEnd, char dst[], int

`dstBegin)`

Parameters

`srcBegin`

The index of the first character to be copied.

`srcEnd`

The index after the last character to be copied.

`dst`

The destination `char` array.

`dstBegin`

An offset into the destination array.

Throws

`StringIndexOutOfBoundsException`

If `srcBegin`, `srcEnd`, or `dstBegin` is out of range.

Description

This method copies each character in the specified range of this `StringBuffer` object to the given array of `char` values. More specifically, the first character to be copied is at index `srcBegin`; the last character to be copied is at index `srcEnd-1`.

These characters are copied into `dst`, starting at index `dstBegin` and ending at index:

```
dstBegin + (srcEnd-srcBegin) - 1
```

insert

public StringBuffer insert(int offset, boolean b)

Parameters

`offset`

An offset into the `StringBuffer`.

`b`

A `boolean` value.

Returns

This `StringBuffer` object.

Throws

`StringIndexOutOfBoundsException`

If `offset` is out of range.

Description

This method inserts either `"true"` or `"false"` into the sequence of characters stored in this `StringBuffer` object, depending on the value of `b`. The string is inserted at a position `offset` characters from the beginning of the sequence. If `offset` is `0`, the string is inserted before the first character in the `StringBuffer`.

`public synchronized StringBuffer insert(int offset, char c)`

Parameters

`offset`

An offset into the `StringBuffer`.

`c`

A `char` value.

Returns

This `StringBuffer` object.

Throws

`StringIndexOutOfBoundsException`

If `offset` is less than zero or greater than or equal to the length of the `StringBuffer` object.

Description

This method inserts the given character into the sequence of characters stored in this `StringBuffer` object. The character is inserted at a position `offset` characters from the beginning of the sequence. If `offset` is `0`, the character is inserted before the first character in the `StringBuffer`.

public synchronized StringBuffer insert(int offset, char str[])

Parameters

`offset`

An offset into the `StringBuffer`.

`str`

An array of `char` values.

Returns

This `StringBuffer` object.

Throws

`StringIndexOutOfBoundsException`

If `offset` is less than zero or greater than or equal to the length of the `StringBuffer` object.

Description

This method inserts the characters in the given array into the sequence of characters stored in this `StringBuffer` object. The characters are inserted at a position `offset` characters from the beginning of the sequence. If `offset` is `0`, the characters are inserted before the first character in the `StringBuffer`.

`public StringBuffer insert(int offset, double d)`

Parameters

`offset`

An offset into the `StringBuffer`.

`d`

A `double` value.

Returns

This `StringBuffer` object.

Throws

`StringIndexOutOfBoundsException`

If `offset` is less than zero or greater than or equal to the length of the `StringBuffer` object.

Description

This method converts the given `double` value to a string using `Double.toString(d)` and inserts the resulting string into the sequence of characters stored in this `StringBuffer` object. The string is inserted at a position `offset` characters from the beginning of the sequence. If `offset` is `0`, the string is inserted before the first character in the `StringBuffer`.

`public StringBuffer insert(int offset, float f)`

Parameters

`offset`

An offset into the `StringBuffer`.

`f`

A `float` value.

Returns

This `StringBuffer` object.

Throws

`StringIndexOutOfBoundsException`

If `offset` is less than zero or greater than or equal to the length of the `StringBuffer` object.

Description

This method converts the given `float` value to a string using `Float.toString(f)` and inserts the resulting string into the sequence of characters stored in this `StringBuffer` object. The string is inserted at a position `offset` characters from the beginning of the sequence. If `offset` is `0`, the string is inserted before the first character in the `StringBuffer`.

`public StringBuffer insert(int offset, int i)`

Parameters

`offset`

An offset into the `StringBuffer`.

`i`

An `int` value.

Returns

This `StringBuffer` object.

Throws

`StringIndexOutOfBoundsException`

If `offset` is less than zero or greater than or equal to the length of the `StringBuffer` object.

Description

This method converts the given `int` value to a string using `Integer.toString(i)` and inserts the resulting string into the sequence of characters stored in this `StringBuffer` object. The string is inserted at a position `offset` characters from the beginning of the sequence. If `offset` is `0`, the string is inserted before the first character in the `StringBuffer`.

`public StringBuffer insert(int offset, long l)`

Parameters

`offset`

An offset into the `StringBuffer`.

`l`

A `long` value.

Returns

This `StringBuffer` object.

Throws

`StringIndexOutOfBoundsException`

If `offset` is less than zero or greater than or equal to the length of the `StringBuffer` object.

Description

This method converts the given `long` value to a string using `Long.toString(l)` and inserts the resulting string into the sequence of characters stored in this `StringBuffer` object. The string is inserted at a position `offset` characters from the beginning of the sequence. If `offset` is `0`, the string is inserted before the first character in the `StringBuffer`.

public synchronized StringBuffer insert(int offset, Object obj)

Parameters

`offset`

An offset into the `StringBuffer`.

`obj`

A reference to an object.

Returns

This `StringBuffer` object.

Throws

`StringIndexOutOfBoundsException`

If `offset` is less than zero or greater than or equal to the length of the `StringBuffer` object.

Description

This method gets the string representation of the given object by calling `String.valueOf(obj)` and inserts the resulting string into the sequence of characters stored in this `StringBuffer` object. The string is inserted at a position `offset` characters from the beginning of the sequence. If `offset` is `0`, the string is inserted before the first character in the `StringBuffer`.

public synchronized StringBuffer insert(int offset, String str)

Parameters

`offset`

An offset into the `StringBuffer`.

`str`

A `String` object.

Returns

This `StringBuffer` object.

Throws

`StringIndexOutOfBoundsException`

If `offset` is less than zero or greater than or equal to the length of the `StringBuffer` object.

Description

This method inserts the sequence of characters represented by the given `String` into the sequence of characters stored in this `StringBuffer` object. If `str` is `null`, the string `"null"` is inserted. The string is inserted at a position `offset` characters from the beginning of the sequence. If `offset` is `0`, the string is inserted before the first character in the `StringBuffer`.

length

`public int length()`

Returns

The number of characters stored in this `StringBuffer` object.

Description

This method returns the number of characters stored in this `StringBuffer` object. The length is distinct from the capacity of a `StringBuffer`, which is the number of characters that its internal data structure can hold.

reverse

`public synchronized StringBuffer reverse()`

Returns

> This `StringBuffer` object.

Description

> This method reverses the sequence of characters stored in this `StringBuffer` object.

setCharAt

`public synchronized void setCharAt(int index, char ch)`

Parameters

> `index`
>
> > The index of the character to be set.
>
> `ch`
>
> > A `char` value.

Throws

> `StringIndexOutOfBoundsException`
>
> > If `index` is less than zero or greater than or equal to the length of the `StringBuffer` object.

Description

> This method modifies the character located `index` characters from the beginning of the sequence of characters stored in this `StringBuffer` object. The current character at this position is replaced by the character `ch`.

setLength

`public synchronized void setLength(int newLength)`

Parameters

> `newLength`

The new length for this `StringBuffer`.

Throws

`StringIndexOutOfBoundsException`

If `index` is less than zero.

Description

This method sets the length of the sequence of characters stored in this `StringBuffer` object. If the length is set to be less than the current length, characters are lost from the end of the character sequence. If the length is set to be more than the current length, NUL characters (`\u0000`) are added to the end of the character sequence.

toString

`public String toString()`

Returns

A new `String` object that represents the same sequence of characters as the sequence of characters stored in this `StringBuffer` object.

Overrides

`Object.toString()`

Description

This method returns a new `String` object that represents the same sequence of characters as the sequence of characters stored in this `StringBuffer` object. Note that any subsequent changes to the contents of this `StringBuffer` object do not affect the contents of the `String` object created by this method.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`wait()`	`Object`

`wait(long)`	`Object`	`wait(long, int)`	`Object`

See Also

Character; Double; Exceptions; Float; Integer; Long; Object; String; String Concatenation Operator +

PREVIOUS String | HOME BOOK INDEX | NEXT System

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Number

Name

Number

Synopsis

Class Name:

`java.lang.Number`

Superclass:

`java.lang.Object`

Immediate Subclasses:

`java.lang.Byte`, `java.lang.Double`,

`java.lang.Float`, `java.lang.Integer`,

`java.lang.Long`, `java.lang.Short`,

`java.math.BigDecimal`,

`java.math.BigInteger`

Interfaces Implemented:

```
java.io.Serializable
```

Availability:

JDK 1.0 or later

Description

The `Number` class is an `abstract` class that serves as the superclass for all of the classes that provide object wrappers for primitive numeric values: `byte`, `short`, `int`, `long`, `float`, and `double`. Wrapping a primitive value is useful when you need to treat such a value as an object. For example, there are a number of utility methods that take a reference to an `Object` as one of their arguments. You cannot specify a primitive value for one of these arguments, but you can provide a reference to an object that encapsulates the primitive value. Furthermore, as of JDK 1.1, these wrapper classes are necessary to support the Reflection API and class literals.

The `Number` class defines six methods that must be implemented by its subclasses: `byteValue()`, `shortValue()`, `intValue()`, `longValue()`, `floatValue()`, and `doubleValue()`. This means that a `Number` object can be fetched as an `byte`, `short`, `int`, `long`, `float`, or `double` value, without regard for its actual class.

Class Summary

```
public abstract class java.lang.Number extends java.lang.Number
                                       implements java.io.Serializable {
    // Instance Methods
    public abstract byte byteValue();                  // New in 1.1
    public abstract double doubleValue();
    public abstract float floatValue();
    public abstract int intValue();
    public abstract long longValue();
    public abstract short shortValue();                // New in 1.1
}
```

Instance Methods

byteValue

`public abstract byte byteValue()`

Availability

New as of JDK 1.1

Returns

The value of this object as a `byte`.

Description

This method returns the value of this object as a `byte`. If the data type of the value is not `byte`, rounding may occur.

doubleValue

`public abstract double doubleValue()`

Returns

The value of this object as a `double`.

Description

This method returns the value of this object as a `double`. If the data type of the value is not `double`, rounding may occur.

floatValue

`public abstract float floatValue()`

Returns

The value of this object as a `float`.

Description

This method returns the value of this object as a `float`. If the data type of the value is not `float`, rounding may occur.

intValue

`public abstract int intValue()`

Returns

> The value of this object as an `int`.

Description

> This method returns the value of this object as an `int`. If the type of value is not an `int`, rounding may occur.

longValue

`public abstract long longValue()`

Returns

> The value of this object as a `long`.

Description

> This method returns the value of this object as a `long`. If the type of value is not a `long`, rounding may occur.

shortValue

`public abstract short shortValue()`

Availability

> New as of JDK 1.1

Returns

> The value of this object as a `short`.

Description

> This method returns the value of this object as a `short`. If the type of value is not a `short`, rounding may occur.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

Byte; Double; Float; Integer; Long; Object; Short

Math

Name

Math

Synopsis

Class Name:

`java.lang.Math`

Superclass:

`java.lang.Object`

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The `Math` class contains constants for the mathematical values pi and *e*. The class also defines methods that compute various mathematical functions, such as trigonometric and exponential functions. All of these constants and methods are `static`. In other words, it is not necessary to create an instance of the `Math` class in order to use its constants and methods. In fact, the `Math` class does not define any `public` constructors, so it cannot be instantiated.

To ensure that the methods in this class return consistent results under different implementations of Java, all of the methods use the algorithms from the well-known Freely-Distributable Math Library package, *fdlibm*. This package is part of the network library *netlib*. The library can be obtained through the URL *http://netlib.att.com*. The algorithms used in this class are from the version of *fdlibm* dated January 4, 1995. *fdlibm* provides more than one definition for some functions. In those cases, the "IEEE 754 core function" version is used.

Class Summary

```
public final class java.lang.Math extends java.lang.Object {
    // Constants
    public static final double E;
    public static final double PI;
    // Class Methods
    public static int abs(int a);
    public static long abs(long a);
    public static float abs(float a);
    public static double abs(double a);
    public static native double acos(double a);
    public static native double asin(double a);
    public static native double atan(double a);
    public static native double atan2(double a, double b);
    public static native double ceil(double a);
    public static native double cos(double a);
    public static native double exp(double a);
    public static native double floor(double a);
    public static native double IEEEremainder(double f1, double f2);
    public static native double log(double a);
    public static int max(int a, int b);
    public static long max(long a, long b);
    public static float max(float a, float b);
    public static double max(double a, double b);
    public static int min(int a, int b);
    public static long min(long a, long b);
    public static float min(float a, float b);
    public static double min(double a, double b);
    public static native double pow(double a, double b);
    public static synchronized double random();
    public static native double rint(double a);
    public static int round(float a);
    public static long round(double a);
    public static native double sin(double a);
    public static native double sqrt(double a);
    public static native double tan(double a);
}
```

Constants

E

`public static final double E = 2.7182818284590452354`

Description

> The value of this constant is *e*, the base for natural logarithms.

PI

`public static final double PI = 3.14159265358979323846`

Description

> The value for this constant is pi.

Class Methods

abs

`public static double abs(double a)`

Parameters

`a`

A `double` value.

Returns

The absolute value of its argument.

Description

This method returns the absolute value of its argument.

If the argument to this method is negative or positive zero, the method should return positive zero. If the argument is positive or negative infinity, the method returns positive infinity. If the argument is NaN, the method returns NaN.

`public static float abs(float a)`

Parameters

`a`

A `float` value.

Returns

The absolute value of its argument.

Description

This method returns the absolute value of its argument.

If the argument to this method is negative or positive zero, the method should return positive zero. If the argument is positive or negative infinity, the method returns positive infinity. If the argument is NaN, the method returns NaN.

`public static int abs(int a)`

Parameters

`a`

An `int` value.

Returns

The absolute value of its argument.

Description

This method returns the absolute value of its argument.

If the argument is `Integer.MIN_VALUE`, the method actually returns `Integer.MIN_VALUE` because the true absolute value of `Integer.MIN_VALUE` is one greater than the largest positive value that can be represented by an `int`.

`public static long abs(long a)`

Parameters

`a`

A `long` value.

Returns

The absolute value of its argument.

Description

This method returns the absolute value of its argument.

If the argument is `Long.MIN_VALUE`, the method actually returns `Long.MIN_VALUE` because the true absolute value of `Long.MIN_VALUE` is one greater than the largest positive value represented by a `long`.

acos

`public static native double acos(double a)`

Parameters

`a`

A `double` value greater than or equal to `-1.0` and less than or equal to `1.0`.

Returns

The arc cosine measured in radians; the result is greater than or equal to `0.0` and less than or equal to pi.

Description

This method returns the arc cosine of the given value.

If the value is NaN or its absolute value is greater than `1.0`, the method returns NaN.

asin

`public static native double asin(double a)`

Parameters

`a`

A `double` value greater than or equal to `-1.0` and less than or equal to `1.0`.

Returns

The arc sine measured in radians; the result is greater than or equal to -pi/2 and less than or equal to pi/2.

Description

This method returns the arc sine of the given value.

If the value is NaN or its absolute value is greater than `1.0`, the method returns NaN. If the value is positive zero, the method returns positive zero. If the value is negative zero, the method returns negative zero.

atan

`public static native double atan(double a)`

Parameters

`a`

A `double` value greater than or equal to `-1.0` and less than or equal to `1.0`.

Returns

The arc tangent measured in radians; the result is greater than or equal to -pi/2 and less than or equal to pi/2.

Description

This method returns the principle value of the arc tangent of the given value.

If the value is NaN, the method returns NaN. If the value is positive zero, the method returns positive zero. If the value is negative zero, the method returns negative zero.

atan2

`public static native double atan2(double a, double b)`

Parameters

`a`

A `double` value.

`b`

A `double` value.

Returns

The theta component of the polar coordinate (*r*,theta) that corresponds to the cartesian coordinate (`a`,`b`); the result is measured in radians and is greater than or equal to -pi and less than or equal to pi.

Description

This method returns the theta component of the polar coordinate (*r*,theta) that corresponds to the cartesian coordinate (`a`,`b`). It computes theta as the principle value of the arc tangent of `b`/`a`, using the signs of both arguments to determine the quadrant (and sign) of the return value.

If either argument is NaN, the method returns NaN.

If the first argument is positive zero and the second argument is positive, then the method returns positive zero. If the first argument is positive zero and the second argument is negative, then the method returns the `double` value closest to pi.

If the first argument is negative zero and the second argument is positive, the method returns negative zero. If the first argument is negative zero and the second argument is negative, the method returns the `double` value closest to -pi.

If the first argument is positive and finite and the second argument is positive infinity, the method returns positive zero. If the first argument is positive and finite and the second argument is negative infinity, the method returns the `double` value closest to pi.

If the first argument is negative and finite and the second argument is positive infinity, the method returns negative zero. If the first argument is negative and finite and the second argument is negative infinity, the method returns the `double` value closest to -pi.

If the first argument is positive and the second argument is positive zero or negative zero, the method returns the `double` value closest to pi/2. If the first argument is negative and the second argument is positive or negative zero, the method returns the `double` value closest to -pi/2.

If the first argument is positive infinity and the second argument is finite, the method returns the `double` value closest to pi/2. If the first argument is negative infinity and the second argument is finite, the method returns the `double` value closest to -pi/2.

If both arguments are positive infinity, the method returns the `double` value closest to pi/4. If the first argument is positive infinity and the second argument is negative infinity, the method returns the `double` value closest to 3pi/4. If the first argument is negative infinity and the second argument is positive infinity, the method returns the `double` value closest to -pi/4. If both arguments are negative infinity, the method returns the `double` value closest to -3pi/4.

ceil

```
public static native double ceil(double a)
```

Parameters

`a`

A `double` value.

Returns

The smallest integer greater than or equal to the given value.

Description

This method performs the ceiling operation. It returns the smallest integer that is greater than or equal to its argument.

If the argument is NaN, an infinity value, or a zero value, the method returns that same value. If the argument is less than zero but greater than `-1.0`, the method returns negative zero.

cos

`public static native double cos(double a)`

Parameters

`a`

A `double` value that's an angle measured in radians.

Returns

The cosine of the given angle.

Description

This method returns the cosine of the given angle measured in radians.

If the angle is NaN or an infinity value, the method returns NaN.

exp

`public static native double exp(double a)`

Parameters

`a`

A `double` value.

Returns

`e^a`

Description

This method returns the exponential function of `a`. In other words, *e* is raised to the value specified by the parameter `a`, where *e* is the base of the natural logarithms.

If the value is NaN, the method returns NaN. If the value is positive infinity, the method returns positive infinity. If the value is negative infinity, the method returns positive zero.

floor

`public static native double floor(double a)`

Parameters

`a`

A `double` value.

Returns

The greatest integer less than or equal to the given value.

Description

This method performs the floor operation. It returns the largest integer that is less than or equal to its argument.

If the argument is NaN, an infinity value, or a zero value, the method returns that same value.

IEEEremainder

`public static native double IEEEremainder(double a, double b)`

Parameters

`a`

A `double` value.

`b`

A `double` value.

Returns

The remainder of `a` divided by `b` as defined by the IEEE 754 standard.

Description

This method returns the remainder of `a` divided by `b` as defined by the IEEE 754 standard. This operation involves first determining the mathematical quotient of `a`/`b` rounded to the nearest integer. If the quotient is equally close to two integers, it is rounded to the even integer. The method then returns `a-(b x `*`Q`*`)`, where *`Q`* is the rounded quotient.

If either argument is NaN, the method returns NaN. If the first argument is positive or negative infinity and the second argument is positive or negative zero, the method also returns NaN. If the first argument is a finite value and the second argument is positive or negative infinity, the method returns its first argument.

log

`public static native double log(double a)`

Parameters

`a`

A `double` value that is greater than `0.0`.

Returns

The natural logarithm of `a`.

Description

This method returns the natural logarithm (base *e*) of its argument.

In particular, if the argument is positive infinity, the method returns positive infinity. If the argument is positive or negative zero, the method returns negative infinity. If the argument is less than zero, the method returns NaN. If the argument is NaN, the method returns NaN.

max

`public static double max(double a, double b)`

Parameters

`a`

A `double` value.

`b`

A `double` value.

Returns

The greater of `a` and `b`.

Description

This method returns the greater of its two arguments. In other words, it returns the one that is closer to positive infinity.

If one argument is positive zero and the other is negative zero, the method returns positive zero. If either argument is NaN, the method returns NaN.

`public static float max(float a, float b)`

Parameters

`a`

A `float` value.

`b`

A `float` value.

Returns

The greater of `a` and `b`.

Description

This method returns the greater of its two arguments. In other words, it returns the one that is closer to positive infinity.

If one argument is positive zero and the other is negative zero, the method returns positive zero. If either argument is NaN, the method returns NaN.

`public static int max(int a, int b)`

Parameters

`a`

An `int` value.

`b`

An `int` value.

Returns

The greater of `a` and `b`.

Description

This method returns the greater of its two arguments. In other words, it returns the one that is closer to `Integer.MAX_VALUE`.

`public static long max(long a, long b)`

Parameters

`a`

A `long` value.

`b`

A `long` value.

Returns

The greater of `a` and `b`.

Description

This method returns the greater of its two arguments. In other words, it returns the one that is closer to `Long.MAX_VALUE`.

min

`public static double min(double a, double b)`

Parameters

`a`

A `double` value.

`b`

A `double` value.

Returns

The lesser of `a` and `b`.

Description

This method returns the lesser of its two arguments. In other words, it returns the one that is closer to negative infinity.

If one argument is positive zero and the other is negative zero, the method returns negative zero. If either argument is NaN, the method returns NaN.

`public static float min(float a, float b)`

Parameters

`a`

A `float` value.

`b`

A `float` value.

Returns

The lesser of `a` and `b`.

Description

This method returns the lesser of its two arguments. In other words, it returns the one that is closer to negative infinity.

If one argument is positive zero and the other is negative zero, the method returns negative zero. If either argument is NaN, the method returns NaN.

`public static int min(int a, int b)`

Parameters

`a`

An `int` value.

`b`

An `int` value.

Returns

The lesser of `a` and `b`.

Description

This method returns the lesser of its two arguments. In other words, it returns the one that is closer to `Integer.MIN_VALUE`.

`public static long min(long a, long b)`

Parameters

`a`

A `long` value.

`b`

A `long` value.

Returns

The lesser of `a` and `b`.

Description

This method returns the lesser of its two arguments. In other words, it returns the one that is closer to `Long.MIN_VALUE`.

pow

`public static native double pow(double a, double b)`

Parameters

`a`

A `double` value.

`b`

A `double` value.

Returns

`a^b`

Description

This method computes the value of raising `a` to the power of `b`.

If the second argument is positive or negative zero, the method returns `1.0`. If the second argument is `1.0`, the method returns its first argument. If the second argument is NaN, the method returns NaN. If the first argument is NaN and the second argument is nonzero, the method returns NaN.

If the first argument is positive zero and the second argument is greater than zero, the method returns positive zero. If the first argument is positive zero and the second argument is less than zero, the method returns positive infinity.

If the first argument is positive infinity and the second argument is less than zero, the method returns positive zero. If the first argument is positive infinity and the second argument is greater than zero, the method returns positive infinity.

If the absolute value of the first argument is greater than `1` and the second argument is positive infinity, the method returns positive infinity. If the absolute value of the first argument is greater than `1` and the second argument is negative infinity, the method returns positive zero. If the absolute value of the first argument is less than `1` and the second argument is negative infinity, the method returns positive infinity. If the absolute value of the first argument is less than `1` and the second argument is positive infinity, the method returns positive zero. If the absolute value of the first argument is `1` and the second argument is positive or negative infinity, the method returns NaN.

If the first argument is negative zero and the second argument is greater than zero but not a finite odd integer, the method returns positive zero. If the first argument is negative zero and the second argument is a positive finite odd integer, the method returns negative zero. If the first argument is negative zero and the second argument is less than zero but not a finite odd integer, the method returns positive infinity. If the first argument is negative zero and the second argument is a negative finite odd integer, the method returns negative infinity.

If the first argument is negative infinity and the second argument is less than zero but not a finite odd integer, the method returns positive zero. If the first argument is negative infinity and the second argument is a negative finite odd integer, the method returns negative zero. If the first argument is negative infinity and the second argument is greater than zero but not a finite odd integer, the method returns positive infinity. If the first argument is negative infinity and the second argument is a positive finite odd integer, the method returns negative infinity.

If the first argument is less than zero and the second argument is a finite even integer, the method returns the result of the absolute value of the first argument raised to the power of the second argument. If the first argument is less than zero and the second argument is a finite odd integer, the method returns the negative of the result of the absolute value of the first argument raised to the power of the second argument. If the first argument is finite and less than zero and the second argument is finite and not an integer, the method returns NaN.

If both arguments are integer values, the method returns the first argument raised to the power of the second argument.

random

`public static synchronized double random()`

Returns

A random number between `0.0` and `1.0`.

Description

This method returns a random number greater than or equal to `0.0` and less than `1.0`. The implementation of this method uses the `java.util.Random` class. You may prefer to use the `Random` class directly, in order to gain more control over the distribution, type, and repeatability of the random numbers you are generating.

rint

`public static native double rint(double a)`

Parameters

`a`

A `double` value.

Returns

The value of its argument rounded to the nearest integer.

Description

This method returns its argument rounded to the nearest integer; the result is returned as a `double` value. If the argument is equidistant from two integers (e.g., `1.5`), the method returns the even integer.

If the argument is an infinity value, a zero value, or NaN, the method returns that same value.

round

`public static long round(double a)`

Parameters

`a`

A `double` value.

Returns

The value of its argument rounded to the nearest `long`.

Description

This method returns its `double` argument rounded to the nearest integral value and converted to a `long`. If the argument is equidistant from two integers, the method returns the greater of the two integers.

If the argument is positive infinity or any other value greater than `Long.MAX_VALUE`, the method returns `Long.MAX_VALUE`. If the argument is negative infinity or any other value less than `Long.MIN_VALUE`, the method returns `Long.MIN_VALUE`. If the argument is NaN, the method returns `0`.

public static int round(float a)

Parameters

a

A float value.

Returns

The value of its argument rounded to the nearest int.

Description

This method returns its float argument rounded to the nearest integral value and converted to an int. If the argument is equidistant from two integers, the method returns the greater of the two integers.

If the argument is positive infinity or any other value greater than the Integer.MAX_VALUE, the method returns Integer.MAX_VALUE. If the argument is negative infinity or any other value less than Integer.MIN_VALUE, the method returns Integer.MIN_VALUE. If the argument is NaN, the method returns 0.

sin

public static native double sin(double a)

Parameters

a

A double value that's an angle measured in radians.

Returns

The sine of the given angle.

Description

This method returns the sine of the given angle measured in radians.

If the angle is NaN or an infinity value, the method returns NaN. If the angle is positive zero, the method returns positive zero. If the angle is negative zero, the method returns negative zero.

sqrt

`public static native double sqrt(double a)`

Parameters

`a`

A `double` value.

Returns

The square root of its argument.

Description

This method returns the square root of its argument.

If the argument is negative or NaN, the method returns NaN. If the argument is positive infinity, the method returns positive infinity. If the argument is positive or negative zero, the method returns that same value.

tan

`public static native double tan(double a)`

Parameters

`a`

A `double` value that is an angle measured in radians.

Returns

The tangent of the given angle.

Description

This method returns the tangent of the given angle measured in radians.

If the angle is NaN or an infinity value, the method returns NaN. If the angle is positive zero, the method returns positive zero. If the angle is negative zero, the method returns negative zero.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

Double; Float; Floating-point literals; Floating-point types; Integer; Integer literals; Integer types; Long; Object

SecurityManager

Name

SecurityManager

Synopsis

Class Name:

> `java.lang.SecurityManager`

Superclass:

> `java.lang.Object`

Immediate Subclasses:

> None

Interfaces Implemented:

> None

Availability:

> JDK 1.0 or later

Description

The SecurityManager class provides a way of implementing a comprehensive security policy for a Java program. As of this writing, SecurityManager objects are used primarily by Web browsers to establish security policies for applets. However, the use of a SecurityManager object is appropriate in any situation where a hosting environment wants to limit the actions of hosted programs.

The SecurityManager class contains methods that are called by methods in other classes to ask for permission to do something that can affect the security of the system. These permission methods all have names that begin with check. If a check method does not permit an action, it throws a SecurityException or returns a value that indicates the lack of permission. The SecurityManager class provides default implementations of all of the check methods. These default implementations are the most restrictive possible implementations; they simply deny permission to do anything that can affect the security of the system.

The SecurityManager class is an abstract class. A hosting environment should define a subclass of SecurityManager that implements an appropriate security policy. To give the subclass of SecurityManager control over security, the hosting environment creates an instance of the class and installs it by passing it to the setSecurityManager() method of the System class. Once a SecurityManager object is installed, it cannot be changed. If the setSecurityManager() method is called any additional times, it throws a SecurityException.

The methods in other classes that want to ask the SecurityManager for permission to do something are able to access the SecurityManager object by calling the getSecurityManager() method of the System class. This method returns the SecurityManager object, or null to indicate that there is no SecurityManager installed.

Class Summary

```
public abstract class java.lang.SecurityManager extends java.lang.Object {
    // Constructors
    protected SecurityManager();
    // Variables
    protected boolean inCheck;
    // Instance Methods
    public void checkAccept(String host, int port);
    public void checkAccess(Thread t);
    public void checkAccess(ThreadGroup g);
    public void checkAwtEventQueueAccess();                  // New in 1.1
    public void checkConnect(String host, int port);
    public void checkConnect(String host, int port, Object context);
    public void checkCreateClassLoader();
    public void checkDelete(String file);
    public void checkExec(String cmd);
    public void checkExit(int status);
    public void checkLink(String libname);
    public void checkListen(int port);
    public void checkMemberAccess(Class clazz, int which);   // New in 1.1
    public void checkMulticast(InetAddress maddr);           // New in 1.1
    public void checkMulticast(InetAddress maddr, byte ttl); // New in 1.1
    public void checkPackageAccess();
    public void checkPackageDefinition();
    public void checkPrintJobAccess();                       // New in 1.1
    public void checkPropertiesAccess();
    public void checkPropertyAccess(String key);
    public void checkRead(int fd);
    public void checkRead(String file);
    public void checkRead(String file, Object context);
    public void checkSecurityAccess(String action);          // New in 1.1
    public void checkSetFactory();
    public void checkSystemClipboardAccess();                // New in 1.1
    public boolean checkTopLevelWindow();
    public void checkWrite(int fd);
    public void checkWrite(String file);
    public boolean getInCheck();
    public Object getSecurityContext();
    public ThreadGroup getThreadGroup();                     // New in 1.1
    // Protected Instance Methods
    protected int classDepth(String name);
    protected int classLoaderDepth();
    protected ClassLoader currentClassLoader();
    protected Class currentLoadedClass();                    // New in 1.1
    protected Class[] getClassContext();
    protected boolean inClass(String name);
    protected boolean inClassLoader();
}
```

Variables

inCheck

`protected boolean inCheck = false`

Description

> This variable indicates whether or not a security check is in progress. A subclass of `SecurityManager` should set this variable to `true` while a security check is in progress.
>
> This variable can be useful for security checks that require access to resources that a hosted program may not be permitted to access. For example, a security policy might be based on the contents of a

permissions file. This means that the various `check` methods need to read information from a file to decide what to do. Even though a hosted program may not be allowed to read files, the `check` methods can allow such reads when `inCheck` is `true` to support this style of security policy.

Constructors

SecurityManager

`protected SecurityManager()`

Throws

`SecurityException`

If a `SecurityManager` object already exists. In other words, if `System.getSecurityManager()` returns a value other than `null`.

Description

Creates a new `SecurityManager` object. This constructor cannot be called if there is already a current `SecurityManager` installed for the program.

Public Instance Methods

checkAccept

`public void checkAccept(String host, int port)`

Parameters

`host`

The name of the host machine.

`port`

A port number.

Throws

`SecurityException`

If the caller does not have permission to accept the connection.

Description

This method decides whether or not to allow a connection from the given host on the given port to be accepted. An implementation of the method should throw a `SecurityException` to deny permission to accept the connection. The method is called by the `accept()` method of the `java.net.ServerSocket` class.

The `checkAccept()` method of `SecurityManager` always throws a `SecurityException`.

checkAccess

`public void checkAccess(Thread g)`

Parameters

`g`

A reference to a `Thread` object.

Throws

`SecurityException`

If the current thread does not have permission to modify the specified thread.

Description

This method decides whether or not to allow the current thread to modify the specified `Thread`. An implementation of the method should throw a `SecurityException` to deny permission to modify the thread. Methods of the `Thread` class that call this method include `stop()`, `suspend()`, `resume()`, `setPriority()`, `setName()`, and `setDaemon()`.

The `checkAccess()` method of `SecurityManager` always throws a `SecurityException`.

`public void checkAccess(ThreadGroup g)`

Parameters

`g`

A reference to a `ThreadGroup` object.

Throws

`SecurityException`

If the current thread does not have permission to modify the specified thread group.

Description

This method decides whether or not to allow the current thread to modify the specified `ThreadGroup`. An implementation of the method should throw a `SecurityException` to deny permission to modify the thread group. Methods of the `ThreadGroup` class that call this method include `setDaemon()`, `setMaxPriority()`, `stop()`, `suspend()`, `resume()`, and `destroy()`.

The `checkAccess()` method of `SecurityManager` always throws a `SecurityException`.

checkAwtEventQueueAccess

`public void checkAwtEventQueueAccess()`

Availability

New as of JDK 1.1

Throws

`SecurityException`

If the caller does not have permission to access the AWT event queue.

Description

This method decides whether or not to allow access to the AWT event queue. An implementation of the method should throw a `SecurityException` to deny permission to access the event queue. The method is called by the `getSystemEventQueue()` method of the `java.awt.Toolkit` class.

The `checkAwtEventQueueAccess()` method of `SecurityManager` always throws a `SecurityException`.

checkConnect

`public void checkConnect(String host, int port)`

Parameters

`host`

The name of the host.

`port`

A port number. A value of `-1` indicates an attempt to determine the IP address of given hostname.

Throws

`SecurityException`

If the caller does not have permission to open the socket connection.

Description

This method decides whether or not to allow a socket connection to the given host on the given port to be opened. An implementation of the method should throw a `SecurityException` to deny permission to open the connection. The method is called by the constructors of the `java.net.Socket` class, the `send()` and `receive()` methods of the `java.net.DatagramSocket` class, and the `getByName()` and `getAllByName()` methods of the `java.net.InetAddress` class.

The `checkConnect()` method of `SecurityManager` always throws a `SecurityException`.

`public void checkConnect(String host, int port, Object context)`

Parameters

`host`

The name of the host.

`port`

A port number. A value of `-1` indicates an attempt to determine the IP address of given host name.

`context`

A security context object returned by this object's `getSecurityContext()` method.

Throws

`SecurityException`

If the specified security context does not have permission to open the socket connection.

Description

This method decides whether or not to allow a socket connection to the given host on the given port to be opened for the specified security context. An implementation of the method should throw a `SecurityException` to deny permission to open the connection.

The `checkConnect()` method of `SecurityManager` always throws a `SecurityException`.

checkCreateClassLoader

`public void checkCreateClassLoader()`

Throws

`SecurityException`

If the caller does not have permission to create a `ClassLoader` object.

Description

This method decides whether or not to allow a `ClassLoader` object to be created. An implementation of the method should throw a `SecurityException` to deny permission to create a `ClassLoader`. The method is called by the constructor of the `ClassLoader` class.

The `checkCreateClassLoader()` method of `SecurityManager` always throws a `SecurityException`.

checkDelete

`public void checkDelete(String file)`

Parameters

`file`

The name of a file.

Throws

`SecurityException`

If the caller does not have permission to delete the specified file.

Description

This method decides whether or not to allow a file to be deleted. An implementation of the method should throw a `SecurityException` to deny permission to delete the specified file. The method is called by the `delete()` method of the `java.io.File` class.

The `checkDelete()` method of `SecurityManager` always throws a `SecurityException`.

checkExec

`public void checkExec(String cmd)`

Parameters

`cmd`

The name of an external command.

Throws

`SecurityException`

If the caller does not have permission to execute the specified command.

Description

This method decides whether or not to allow an external command to be executed. An implementation of the method should throw a `SecurityException` to deny permission to execute the specified command. The method is called by the `exec()` methods of the `Runtime` and `System` classes.

The `checkExec()` method of `SecurityManager` always throws a `SecurityException`.

checkExit

`public void checkExit(int status)`

Parameters

`status`

An exit status code.

Throws

`SecurityException`

If the caller does not have permission to exit the Java virtual machine with the given status code.

Description

This method decides whether or not to allow the Java virtual machine to exit with the given status code. An implementation of the method should throw a `SecurityException` to deny permission to exit with the specified status code. The method is called by the `exit()` methods of the `Runtime` and `System` classes.

The `checkExit()` method of `SecurityManager` always throws a `SecurityException`.

checkLink

`public void checkLink(String lib)`

Parameters

`lib`

The name of a library.

Throws

`SecurityException`

If the caller does not have permission to load the specified library.

Description

This method decides whether to allow the specified library to be loaded. An implementation of the method should throw a `SecurityException` to deny permission to load the specified library. The method is called by the `load()` and `loadLibrary()` methods of the `Runtime` and `System` classes.

The `checkLink()` method of `SecurityManager` always throws a `SecurityException`.

checkListen

`public void checkListen(int port)`

Parameters

`port`

A port number.

Throws

`SecurityException`

If the caller does not have permission to listen on the specified port.

Description

This method decides whether or not to allow the caller to listen on the specified port. An implementation of the method should throw a `SecurityException` to deny permission to listen on the specified port. The method is called by the constructors of the `java.net.ServerSocket` class and by the constructor of the `java.net.DatagramSocket` class that takes one argument.

The `checkListen()` method of `SecurityManager` always throws a `SecurityException`.

checkMemberAccess

`public void checkMemberAccess(Class clazz, int which)`

Availability

New as of JDK 1.1

Parameters

clazz

A Class object.

which

The value Member.PUBLIC for the set of all public members including inherited members or the value Member.DECLARED for the set of all declared members of the specified class or interface.

Throws

SecurityException

If the caller does not have permission to access the members of the specified class or interface.

Description

This method decides whether or not to allow access to the members of the specified Class object. An implementation of the method should throw a SecurityException to deny permission to access the members. Methods of the Class class that call this method include getField(), getFields(), getDeclaredField(), getDeclaredFields(), getMethod(), getMethods(), getDeclaredMethod(), getDeclaredMethods(), getConstructor(), getConstructors(), getDeclaredConstructor(), getDeclaredConstructors(), and getDeclaredClasses().

The checkMemberAccess() method of SecurityManager always throws a SecurityException.

checkMulticast

public void checkMulticast(InetAddress maddr)

Availability

New as of JDK 1.1

Parameters

maddr

An InetAddress object that represents a multicast address.

Throws

`SecurityException`

If the current thread does not have permission to use the specified multicast address.

Description

This method decides whether or not to allow the current thread to use the specified multicast `InetAddress`. An implementation of the method should throw a `SecurityException` to deny permission to use the multicast address. The method is called by the `send()` method of `java.net.DatagramSocket` if the packet is being sent to a multicast address. The method is also called by the `joinGroup()` and `leaveGroup()` methods of `java.net.MulticastSocket`.

The `checkMulticast()` method of `SecurityManager` always throws a `SecurityException`.

`public void checkMulticast(InetAddress maddr, byte ttl)`

Availability

New as of JDK 1.1

Parameters

`maddr`

An `InetAddress` object that represents a multicast address.

`ttl`

The time-to-live (TTL) value.

Throws

`SecurityException`

If the current thread does not have permission to use the specified multicast address and TTL value.

Description

This method decides whether or not to allow the current thread to use the specified multicast

InetAddress and TTL value. An implementation of the method should throw a SecurityException to deny permission to use the multicast address. The method is called by the send() method of java.net.MulticastSocket.

The checkMulticast() method of SecurityManager always throws a SecurityException.

checkPackageAccess

public void checkPackageAccess(String pkg)

Parameters

pkg

The name of a package.

Throws

SecurityException

If the caller does not have permission to access the specified package.

Description

This method decides whether or not to allow the specified package to be accessed. An implementation of the method should throw a SecurityException to deny permission to access the specified package. The method is intended to be called by implementations of the loadClass() method in subclasses of the ClassLoader class.

The checkPackageAccess() method of SecurityManager always throws a SecurityException.

checkPackageDefinition

public void checkPackageDefinition(String pkg)

Parameters

pkg

The name of a package.

Throws

`SecurityException`

If the caller does not have permission to define classes in the specified package.

Description

This method decides whether or not to allow the caller to define classes in the specified package. An implementation of the method should throw a `SecurityException` to deny permission to create classes in the specified package. The method is intended to be called by implementations of the `loadClass()` method in subclasses of the `ClassLoader` class.

The `checkPackageDefinition()` method of `SecurityManager` always throws a `SecurityException`.

checkPrintJobAccess

`public void checkPrintJobAccess()`

Availability

New as of JDK 1.1

Throws

`SecurityException`

If the caller does not have permission to initiate a print job request.

Description

This method decides whether or not to allow the caller to initiate a print job request. An implementation of the method should throw a `SecurityException` to deny permission to initiate the request.

The `checkPrintJobAccess()` method of `SecurityManager` always throws a `SecurityException`.

checkPropertiesAccess

`public void checkPropertiesAccess()`

Throws

`SecurityException`

If the caller does not have permission to access the system properties.

Description

This method decides whether or not to allow the caller to access and modify the system properties. An implementation of the method should throw a `SecurityException` to deny permission to access and modify the properties. Methods of the `System` class that call this method include `getProperties()` and `setProperties()`.

The `checkPropertiesAccess()` method of `SecurityManager` always throws a `SecurityException`.

checkPropertyAccess

```
public void checkPropertyAccess(String key)
```

Parameters

`key`

The name of an individual system property.

Throws

`SecurityException`

If the caller does not have permission to access the specified system property.

Description

This method decides whether or not to allow the caller to access the specified system property. An implementation of the method should throw a `SecurityException` to deny permission to access the property. The method is called by the `getProperty()` method of the `System` class.

The `checkPropertyAccess()` method of `SecurityManager` always throws a `SecurityException`.

checkRead

`public void checkRead(FileDescriptor fd)`

Parameters

`fd`

A reference to a `FileDescriptor` object.

Throws

`SecurityException`

If the caller does not have permission to read from the given file descriptor.

Description

This method decides whether or not to allow the caller to read from the specified file descriptor. An implementation of the method should throw a `SecurityException` to deny permission to read from the file descriptor. The method is called by the constructor of the `java.io.FileInputStream` class that takes a `FileDescriptor` argument.

The `checkRead()` method of `SecurityManager` always throws a `SecurityException`.

`public void checkRead(String file)`

Parameters

`file`

The name of a file.

Throws

`SecurityException`

If the caller does not have permission to read from the named file.

Description

This method decides whether or not to allow the caller to read from the named file. An implementation of the method should throw a `SecurityException` to deny permission to read from the file. The method is called by constructors of the `java.io.FileInputStream` and java.io.RandomAccessFile classes, as well as by the canRead(), exists(), isDirectory(), isFile(), lastModified(), length(), and list() methods of the java.io.File class.

The checkRead() method of SecurityManager always throws a SecurityException.

public void checkRead(String file, Object context)

Parameters

file

The name of a file.

context

A security context object returned by this object's getSecurityContext() method.

Throws

SecurityException

If the specified security context does not have permission to read from the named file.

Description

This method decides whether or not to allow the specified security context to read from the named file. An implementation of the method should throw a SecurityException to deny permission to read from the file.

The checkRead() method of SecurityManager always throws a SecurityException.

checkSecurityAccess

public void checkSecurityAccess(String action)

Availability

New as of JDK 1.1

Parameters

action

A string that specifies a security action.

Throws

`SecurityException`

If the caller does not have permission to perform the specified security action.

Description

This method decides whether to allow the caller to perform the specified security action. An implementation of the method should throw a `SecurityException` to deny permission to perform the action. The method is called by many of the methods in the `java.security.Identity` and `java.security.Security` classes.

The `checkSecurityAccess()` method of `SecurityManager` always throws a `SecurityException`.

checkSetFactory

`public void checkSetFactory()`

Throws

`SecurityException`

If the caller does not have permission to set the factory class to be used by another class.

Description

This method decides whether to allow the caller to set the factory class to be used by another class. An implementation of the method should throw a `SecurityException` to deny permission to set the factory class. The method is called by the `setSocketFactory()` method of the `java.net.ServerSocket` class, the `setSocketImplFactory()` method of the `java.net.Socket` class, the `setURLStreamHandlerFactory()` method of the `java.net.URL` class, and the `setContentHandlerFactory()` method of the `java.net.URLConnection` class.

The `checkSetFactory()` method of `SecurityManager` always throws a `SecurityException`.

checkSystemClipboardAccess

`public void checkSystemClipboardAccess()`

Availability

New as of JDK 1.1

Throws

`SecurityException`

If the caller does not have permission to access the system clipboard.

Description

This method decides whether or not to allow the caller to access the system clipboard. An implementation of the method should throw a `SecurityException` to deny permission to access the system clipboard.

The `checkSystemClipboardAccess()` method of `SecurityManager` always throws a `SecurityException`.

checkTopLevelWindow

`public boolean checkTopLevelWindow(Object window)`

Parameters

`window`

A window object.

Returns

`true` if the caller is trusted to put up the specified top-level window; otherwise `false`.

Description

This method decides whether or not to trust the caller to put up the specified top-level window. An implementation of the method should return `false` to indicate that the caller is not trusted. In this case, the hosting environment can still decide to display the window, but the window should include a visual indication that it is not trusted. If the caller is trusted, the method should return `true`, and the window can be displayed without any special indication.

The `checkTopLevelWindow()` method of `SecurityManager` always returns `false`.

checkWrite

`public void checkWrite(FileDescriptor fd)`

Parameters

`fd`

A `FileDescriptor` object.

Throws

`SecurityException`

If the caller does not have permission to write to the given file descriptor.

Description

This method decides whether or not to allow the caller to write to the specified file descriptor. An implementation of the method should throw a `SecurityException` to deny permission to write to the file descriptor. The method is called by the constructor of the `java.io.FileOutputStream` class that takes a `FileDescriptor` argument.

The `checkWrite()` method of `SecurityManager` always throws a `SecurityException`.

`public void checkWrite(String file)`

Parameters

`file`

The name of a file.

Throws

`SecurityException`

If the caller does not have permission to read from the named file.

Description

This method decides whether or not to allow the caller to write to the named file. An implementation of the method should throw a `SecurityException` to deny permission to write to the file. The method is called by constructors of the `java.io.FileOutputStream` and `java.io.RandomAccessFile` classes, as well as by the `canWrite()`, `mkdir()`, and `renameTo()` methods of the `java.io.File` class.

The `checkWrite()` method of `SecurityManager` always throws a `SecurityException`.

getInCheck

`public boolean getInCheck()`

Returns

`true` if a security check is in progress; otherwise `false`.

Description

This method returns the value of the `SecurityManager` object's `inCheck` variable, which is `true` if a security check is in progress and `false` otherwise.

getSecurityContext

`public Object getSecurityContext()`

Returns

An implementation-dependent object that contains enough information about the current execution environment to perform security checks at a later time.

Description

This method is meant to create an object that encapsulates information about the current execution environment. The resulting security context object is used by specific versions of the `checkConnect()` and `checkRead()` methods. The intent is that such a security context object can be used by a trusted method to determine whether or not another, untrusted method can perform a particular operation.

The `getSecurityContext()` method of `SecurityManager` simply returns `null`. This method should be overridden to return an appropriate security context object for the security policy that is being implemented.

getThreadGroup

`public ThreadGroup getThreadGroup()`

Availability

New as of JDK 1.1

Returns

A `ThreadGroup` in which to place any threads that are created when this method is called.

Description

This method returns the appropriate parent `ThreadGroup` for any threads that are created when the method is called. The `getThreadGroup()` method of `SecurityManager` simply returns the `ThreadGroup` of the current thread. This method should be overridden to return an appropriate `ThreadGroup`.

Protected Instance Methods

classDepth

`protected native int classDepth(String name)`

Parameters

`name`

The fully qualified name of a class.

Returns

The number of pending method invocations from the top of the stack to a call to a method of the given class; `-1` if no stack frame in the current thread is associated with a call to a method in the given class.

Description

This method returns the number of pending method invocations between this method invocation and an invocation of a method associated with the named class.

classLoaderDepth

`protected native int classLoaderDepth()`

Returns

> The number of pending method invocations from the top of the stack to a call to a method that is associated with a class that was loaded by a `ClassLoader` object; `-1` if no stack frame in the current thread is associated with a call to such a method.

Description

> This method returns the number of pending method invocations between this method invocation and an invocation of a method associated with a class that was loaded by a `ClassLoader` object.

currentClassLoader

`protected native ClassLoader currentClassLoader()`

Returns

> The most recent `ClassLoader` object executing on the stack.

Description

> This method finds the most recent pending invocation of a method associated with a class that was loaded by a `ClassLoader` object. The method then returns the `ClassLoader` object that loaded that class.

currentLoadedClass

`protected Class currentLoadedClass()`

Availability

> New as of JDK 1.1

Returns

> The most recent `Class` object loaded by a `ClassLoader`.

Description

This method finds the most recent pending invocation of a method associated with a class that was loaded by a `ClassLoader` object. The method then returns the `Class` object for that class.

getClassContext

`protected native Class[] getClassContext()`

Returns

An array of `Class` objects that represents the current execution stack.

Description

This method returns an array of `Class` objects that represents the current execution stack. The length of the array is the number of pending method calls on the current thread's stack, not including the call to `getClassContext()`. Each element of the array references a `Class` object that describes the class associated with the corresponding method call. The first element of the array corresponds to the most recently called method, the second element is that method's caller, and so on.

inClass

`protected boolean inClass(String name)`

Parameters

`name`

The fully qualified name of a class.

Returns

`true` if there is a pending method invocation on the stack for a method of the given class; otherwise `false`.

Description

This method determines whether or not there is a pending method invocation that is associated with the named class.

inClassLoader

`protected boolean inClassLoader()`

Returns

> `true` if there is a pending method invocation on the stack for a method of a class that was loaded by a `ClassLoader` object; otherwise `false`.

Description

> This method determines whether or not there is a pending method invocation that is associated with a class that was loaded by a `ClassLoader` object. The method returns `true` only if the `currentClassLoader()` method does not return `null`.

Inherited Methods

Method	**Inherited From**	**Method**	**Inherited From**
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

Class; ClassLoader; Exceptions; Object; Runtime; System; Thread; ThreadGroup

PREVIOUS HOME NEXT
Runtime BOOK INDEX Short

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

PREVIOUS

NEXT

Compiler

Name

Compiler

Synopsis

Class Name:

```
java.lang.Compiler
```

Superclass:

```
java.lang.Object
```

Immediate Subclasses:

None

Interfaces Implemented:

None

Availability:

JDK 1.0 or later

Description

The `Compiler` class encapsulates a facility for compiling Java classes to native code. As provided by Sun, the methods of this class do not actually do anything. However, if the system property `java.compiler` has been defined and if the method `System.loadLibrary()` is able to load the library named by the property, the methods of this class use the implementations provided in the library.

The `Compiler` class has no `public` constructors, so it cannot be instantiated.

Class Summary

```
public final class java.lang.Compiler extends java.lang.Object {
    // Class Methods
    public static native Object command(Object any);
    public static native boolean compileClass(Class clazz);
    public static native boolean compileClasses(String string);
    public static native void disable();
    public static native void enable();
}
```

Class Methods

command

`public static native Object command(Object any)`

Parameters

`any`

The permissible value and its meaning is determined by the compiler library.

Returns

A value determined by the compiler library, or `null` if no compiler library is loaded.

Description

This method directs the compiler to perform an operation specified by the given argument. The available operations, if any, are determined by the compiler library.

compileClass

`public static native boolean compileClass(Class clazz)`

Parameters

`clazz`

The class to be compiled to native code.

Returns

`true` if the compilation succeeds, or `false` if the compilation fails or no compiler library is loaded.

Description

This method requests the compiler to compile the specified class to native code.

compileClasses

`public static native boolean compileClasses(String string)`

Parameters

`string`

A string that specifies the names of the classes to be compiled.

Returns

`true` if the compilation succeeds or `false` if the compilation fails or no compiler library is loaded.

Description

This method requests the compiler to compile all of the classes named in the string.

disable

```
public static native void disable()
```

Description

> This method disables the compiler if one is loaded.

enable

```
public static native void enable()
```

Description

> This method enables the compiler if one is loaded.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`
`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

Object; System

PREVIOUS
Cloneable

HOME
BOOK INDEX

NEXT
Double

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

ClassLoader

Name

ClassLoader

Synopsis

Class Name:

> `java.lang.ClassLoader`

Superclass:

> `java.lang.Object`

Immediate Subclasses:

> None

Interfaces Implemented:

> None

Availability:

> JDK 1.0 or later

Description

The `ClassLoader` class provides a mechanism for Java to load classes over a network or from any source other than the local filesystem. The default class-loading mechanism loads classes from files found relative to directories specified by the `CLASSPATH` environment variable. This default mechanism does not use an instance of the `ClassLoader` class.

An application can implement another mechanism for loading classes by declaring a subclass of the `abstract ClassLoader` class. A subclass of `ClassLoader` must override the `loadClass()` to define a class-loading policy.

This method implements any sort of security that is necessary for the class-loading mechanism. The other methods of ClassLoader are final, so they cannot be overridden.

A ClassLoader object is typically used by calling its loadClass() method to explicitly load a top-level class, such as a subclass of Applet. The ClassLoader that loads the class becomes associated with the class; it can be obtained by calling the getClassLoader() method of the Class object that represents the class.

Once a class is loaded, it must be resolved before it can be used. Resolving a class means ensuring that all of the other classes it references are loaded. In addition, all of the classes that they reference must be loaded, and so on, until all of the needed classes have been loaded. Classes are resolved using the resolveClass() method of the ClassLoader object that loaded the initial class. This means that when a ClassLoader object is explicitly used to load a class, the same ClassLoader is used to load all of the classes that it references, directly or indirectly.

Classes loaded using a ClassLoader object may attempt to load additional classes without explicitly using a ClassLoader object. They can do this by calling the Class class' forName() method. However, in such a situation, a ClassLoader object is implicitly used. See the description of Class.forName() for more information.

Java identifies a class by a combination of its fully qualified name and the class loader that was used to load the class. If you write a subclass of ClassLoader, it should not attempt to directly load local classes. Instead, it should call findSystemClass(). A local class that is loaded directly by a ClassLoader is considered to be a different class than the same class loaded by findSystemClass(). This can lead to having two copies of the same class loaded, which can cause a number of inconsistencies. For example, the class' equals() method may decide that the same object is not equal to itself.

Class Summary

```
public abstract class java.lang.ClassLoader extends java.lang.Object {
    // Constructors
    protected ClassLoader();
    // Class Methods
    public static final URL
           getSystemResource(String name);                  // New in 1.1
    public static final InputStream
           getSystemResourceAsStream(String name);          // New in 1.1
    // Public Instance Methods
    public URL getResource(String name);                    // New in 1.1
    public InputStream getResourceAsStream(String name);    // New in 1.1
    public Class loadClass(String name);                    // New in 1.1

    // Protected Instance Methods
    protected final Class defineClass(byte data[],
              int offset, int length);               // Deprecated in 1.1
    protected final Class defineClass(String name,
              byte[] data, int offset, int length);         // New in 1.1
    protected final Class findLoadedClass(String name);     // New in 1.1
    protected final Class findSystemClass(String name);
    protected abstract Class loadClass(String name, boolean resolve);
    protected final void resolveClass(Class c);
    protected final void setSigners(Class cl,
              Object[] signers);                            // New in 1.1
}
```

Constructors

ClassLoader

`protected ClassLoader()`

Throws

`SecurityException`

If there is a `SecurityManager` object installed and its `checkCreateClassLoader()` method throws a `SecurityException` when called by this constructor.

Description

Initializes a `ClassLoader` object. Because `ClassLoader` is an `abstract` class, only subclasses of the class can access this constructor.

Class Methods

getSystemResource

`public static final URL getSystemResource(String name)`

Availability

New as of JDK 1.1

Parameters

`name`

A system resource name.

Returns

A `URL` object that is connected to the specified system resource or `null` if the resource cannot be found.

Description

This method finds a system resource with the given name and returns a `URL` object that is connected to the resource. The resource name can be any system resource.

getSystemResourceAsStream

`public static final InputStream getSystemResourceAsStream(String name)`

Availability

New as of JDK 1.1

Parameters

`name`

A system resource name.

Returns

An `InputStream` object that is connected to the specified system resource or `null` if the resource cannot be found.

Description

This method finds a system resource with the given name and returns an `InputStream` object that is connected to the resource. The resource name can be any system resource.

Public Instance Methods

getResource

`public URL getResource(String name)`

Availability

New as of JDK 1.1

Parameters

`name`

A resource name.

Returns

A `URL` object that is connected to the specified resource or `null` if the resource cannot be found.

Description

This method finds a resource with the given name and returns a `URL` object that is connected to the resource.

A resource is a file that contains data (e.g., sound, images, text) and it can be part of a package. The name of a resource is a sequence of identifiers separated by "`/`". For example, a resource might have the name *help/american/logon.html* . System resources are found on the host machine using the conventions of the host implementation. For example, the "/" in the resource name may be treated as a path separator, with the entire resource name treated as a relative path to be found under a directory in `CLASSPATH`.

The implementation of `getResource()` in `ClassLoader` simply returns `null`. A subclass can override this method to provide more useful functionality.

getResourceAsStream

`public InputStream getResourceAsStream(String name)`

Availability

New as of JDK 1.1

Parameters

`name`

A resource name.

Returns

An `InputStream` object that is connected to the specified resource or `null` if the resource cannot be found.

Description

This method finds a resource with the given name and returns an `InputStream` object that is connected to the resource.

A resource is a file that contains data (e.g., sound, images, text) and it can be part of a package. The name of a resource is a sequence of identifiers separated by `/'. For example, a resource might have the name *help/american/logon.html*. System resources are found on the host machine using the conventions of the host implementation. For example, the `/' in the resource name may be treated as a path separator, with the entire resource name treated as a relative path to be found under a directory in `CLASSPATH`.

The implementation of `getResourceAsStream()` in `ClassLoader` simply returns `null`. A subclass can override this method to provide more useful functionality.

loadClass

public Class loadClass(String name) throws ClassNotFoundException

Availability

New as of JDK 1.1

Parameters

`name`

The name of the class to be returned. The class name should be qualified by its package name. The lack of an explicit package name specifies that the class is part of the default package.

Returns

The `Class` object for the specified class.

Throws

`ClassNotFoundException`

If it cannot find a definition for the named class.

Description

This method loads the named class by calling `loadClass(name, true)`.

Protected Instance Methods

defineClass

```
protected final Class defineClass(byte data[], int offset, int length)
```

Availability

Deprecated as of JDK 1.1

Parameters

`data`

An array that contains the byte codes that define a class.

`offset`

The offset in the array of byte codes.

`length`

The number of byte codes in the array.

Returns

The newly created `Class` object.

Throws

`ClassFormatError`

If the `data` array does not constitute a valid class definition.

Description

This method creates a `Class` object from the byte codes that define the class. Before the class can be used, it must be resolved. The method is intended to be called from an implementation of the `loadClass()` method.

Note that this method is deprecated as of Java 1.1. You should use the version of `defineClass()` that takes a `name` parameter and is therefore more secure.

`protected final Class defineClass(String name, byte data[], int offset, int length)`

Availability

New as of JDK 1.1

Parameters

`name`

The expected name of the class to be defined or `null` if it is not known. The class name should be qualified by its package name. The lack of an explicit package name specifies that the class is part of the default package.

`data`

An array that contains the byte codes that define a class.

`offset`

The offset in the array of byte codes.

`length`

The number of byte codes in the array.

Returns

The newly created `Class` object.

Throws

`ClassFormatError`

If the `data` array does not constitute a valid class definition.

Description

This method creates a `Class` object from the byte codes that define the class. Before the class can be used, it must be resolved. The method is intended to be called from an implementation of the `loadClass()` method.

findLoadedClass

`protected final Class findLoadedClass(String name)`

Availability

New as of JDK 1.1

Parameters

`name`

The name of the class to be returned. The class name should be qualified by its package name. The lack of an explicit package name specifies that the class is part of the default package.

Returns

The `Class` object for the specified loaded class or `null` if the class cannot be found.

Description

This method finds the specified class that has already been loaded.

findSystemClass

`protected final Class findSystemClass(String name) throws ClassNotFoundException`

Parameters

`name`

The name of the class to be returned. The class name should be qualified by its package name. The lack of an explicit package name specifies that the class is part of the default package.

Returns

The `Class` object for the specified system class.

Throws

`ClassNotFoundException`

If the default class-loading mechanism cannot find a definition for the class.

`NoClassDefFoundError`

If the default class-loading mechanism cannot find the class.

Description

This method finds and loads a system class if it has not already been loaded. A *system class* is a class that is loaded by the default class-loading mechanism from the local filesystem. An implementation of the `loadClass()` method typically calls this method to attempt to load a class from the locations specified by the `CLASSPATH` environment variable.

loadClass

`protected abstract Class loadClass(String name, boolean resolve) throws ClassNotFoundException`

Parameters

`name`

The name of the class to be returned. The class name should be qualified by its package name. The lack of an explicit package name specifies that the class is part of the default package.

`resolve`

Specifies whether or not the class should be resolved by calling the `resolveClass()` method.

Returns

The `Class` object for the specified class.

Throws

`ClassNotFoundException`

If it cannot find a definition for the named class.

Description

An implementation of this `abstract` method loads the named class and returns its `Class` object. It is permitted and encouraged for an implementation to cache the classes it loads, rather than load one each time the method is called. An implementation of this method should do at least the following:

1. Load the byte codes that comprise the class definition into a `byte[]`.
2. Call the `defineClass()` method to create a `Class` object to represent the class definition.
3. If the `resolve` parameter is `true`, call the `resolveClass()` method to resolve the class.

If an implementation of this method caches the classes that it loads, it is recommended that it use an instance of the `java.util.Hashtable` to implement the cache.

resolveClass

```
protected final void resolveClass(Class c)
```

Parameters

`c`

The `Class` object for the class to be resolved.

Description

This method resolves the given `Class` object. Resolving a class means ensuring that all of the other classes that the `Class` object references are loaded. In addition, all of the classes that they reference must be loaded, and so on, until all of the needed classes have been loaded.

The `resolveClass()` method should be called by an implementation of the `loadClass()` method when the value of the `loadClass()` method's `resolve` parameter is `true`.

setSigners

```
protected final void setSigners(Class cl, Object[] signers)
```

Availability

New as of JDK 1.1

Parameters

`cl`

The `Class` object for the class to be signed.

`signers`

An array of `Object`s that represents the signers of this class.

Description

This method specifies the objects that represent the digital signatures for this class.

Inherited Methods

Method	Inherited From	Method	Inherited From
`clone()`	`Object`	`equals(Object)`	`Object`

`finalize()`	`Object`	`getClass()`	`Object`
`hashCode()`	`Object`	`notify()`	`Object`
`notifyAll()`	`Object`	`toString()`	`Object`
`wait()`	`Object`	`wait(long)`	`Object`
`wait(long, int)`	`Object`		

See Also

Class; Errors; Exceptions; Object; SecurityManager

PREVIOUS HOME NEXT
Class BOOK INDEX Cloneable

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

8.4 Data Transfer with Cut-and-Paste

Java 1.1 adds cut-and-paste capabilities to Java applications through the classes and interfaces of the `java.awt.datatransfer` package. The `DataFlavor` class is perhaps the most central of these classes. It represents the type of data to be transferred. Every data flavor consists of a human-readable name and a data type specification. The data type can be specified in one of two ways: with a Java `Class` object or with a MIME type string. These two different ways of specifying the data type reflect two different ways of transferring the data. When the data type is specified as a class object, objects of that type are transferred using the object serialization mechanism (which is discussed in Chapter 9, *Object Serialization*). In Example 8.1, for example, the `DataFlavor` is specified using the `Class` object for `java.util.Vector`. This means that data is transferred as a serialized `Vector` object. It also means that the `DataFlavor` object has an implicit MIME type of:

```
application/x-java-serialized-object; class=java.util.Vector
```

The data type of a `DataFlavor` can also be specified as a MIME type. In this case, data is transferred through a stream--the recipient of the data receives a `Reader` stream from which it can read textual data. In this case, the recipient usually has to parse the data according to the rules of the specified MIME type.

The `Transferable` interface is another important piece of the AWT data transfer picture. This interface specifies methods that must be implemented by any object that wants to make data available for transfer. One of its methods returns an array of all the `DataFlavor` types it can use to transfer its data. Another method checks whether the `Transferable` object supports a given method. The most important method, `getTransferData()`, actually returns the data in a format appropriate for the requested `DataFlavor`.

While `DataFlavor` and `Transferable` provide the underlying infrastructure for data transfer, it is the `Clipboard` class and `ClipboardOwner` interface that support the cut-and-paste style of data transfer. A typical cut-and-paste scenario works like this:

- When the user issues a command to "copy" or "cut" something, the initiating application first obtains the system `Clipboard` object by calling the `getSystemClipboard()` method of the `Toolkit` object. Next, it creates a `Transferable` object that represents the data to be transferred. Finally, it passes this transferable object to the clipboard by calling the `setContents()` method of the clipboard. The initiating application must also pass an object that implements the `ClipboardOwner` interface to `setContents()`. By doing so, it becomes the "clipboard owner" and must maintain its `Transferable` object until it ceases to be the clipboard owner.

- When the user issues a command to "paste," the receiving application first obtains the system `Clipboard` object in the same way that the initiating application did. Then, it calls the `getContents()` method of the system clipboard to receive the `Transferable` object stored there. Now it can use the methods defined by the `Transferable` interface to choose a `DataFlavor` for the data transfer and actually transfer the data.
- When the user copies or cuts some other piece of data, a new data transfer is initiated, and the new initiating application (it may be the same one) becomes the new clipboard owner. The previous owner is notified that it is no longer the clipboard owner when the system invokes the `lostOwnership()` method of the `ClipboardOwner` object specified in the initiating call to `setContents()`.

Note that untrusted applets are not allowed to work with the system clipboard because there might be sensitive data on it from other applications. This means that applets cannot participate in inter-application cut-and-paste. Instead, an applet must create a private clipboard object that it can use for intra-applet data transfer.

The `cut()`, `copy()`, and `paste()` methods of Example 8.1 implement cut-and-paste functionality for scribbled lines. They rely on the nested `SimpleSelection` class that implements the `Transferable` and `ClipboardOwner` interfaces. Note the definition of a `DataFlavor` object that serves to specify the type of data transfer. [1]

[1] Although the example application uses the system clipboard, scribbles can only be pasted between windows of the same application, not between separate instances of the application running in separate Java interpreters. In Java 1.1.1, inter-application cut-and-paste only works with the pre-defined `DataFlavor.stringFlavor` and `DataFlavor.textFlavor` data flavors. Custom types like the one used in the example do not correctly interface with the system clipboard.

PREVIOUS HOME NEXT
Printing BOOK INDEX New Feature Demo

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

9. Object Serialization

Contents:

Object serialization is one of the important new features of Java 1.1. Despite its importance, however, serialization is done with a very simple API. This chapter demonstrates several uses of serialization.

9.1 Simple Serialization

Objects are serialized with the `ObjectOutputStream` and they are deserialized with the `ObjectInputStream`. Both of these classes are part of the `java.io` package, and they function, in many ways, like `DataOutputStream` and `DataInputStream` because they define the same methods for writing and reading binary representations of Java primitive types to and from streams. What `ObjectOutputStream` and `ObjectInputStream` add, however, is the ability to write and read non-primitive object and array values to and from a stream.

An object is serialized by passing it to the `writeObject()` method of an `ObjectOutputStream`. This writes out the values of all of its fields, including private fields and fields inherited from superclasses. The values of primitive fields are simply written to the stream as they would be with a `DataOutputStream`. When a field in an object refers to another object, an array, or a string, however, the `writeObject()` method is invoked recursively to serialize that object as well. If that object (or an array element) refers to another object, `writeObject()` is again invoked recursively. Thus, a single call to `writeObject()` may result in an entire "object graph" being serialized. When two or more objects each refer to the other, the serialization algorithm is careful to only output each object once--`writeObject()` cannot enter infinite recursion.

Deserializing an object simply follows the reverse of this process. An object is read from a stream of data by calling the `readObject()` method of an `ObjectInputStream`. This re-creates the object in the state it was in when serialized. If the object refers to other objects, they are recursively deserialized as well.

This ability to serialize an entire graph of objects and read those objects back in later is a very powerful feature that hides itself in two simple looking methods. We used object serialization in Example 8.1, but unless you were paying attention, you might have missed those crucial `writeObject()` and `readObject()` calls. Serialization is used in that `Scribble` example to give the program an automatic file format for saving the user's scribbles.

To refresh your memory, Example 9.1 shows the `save()` method of that application. Note the creation of the `ObjectOutputStream` and the use of the `writeObject()` method. The corresponding `load()` method simply reverses the streams to read the scribble back in. You may want to refer to the complete example in Chapter 8, *New AWT Features* to examine the `save()` and `load()` methods in context. Also note the use of a `GZIPOutputStream` (from `java.util.zip`) to compress the serialized object data before writing it to disk.

Example 9.1: Using Serialized Objects as an Application File Format

```
/**
 * Prompt the user for a filename, and save the scribble in that file.
 * Serialize the vector of lines with an ObjectOutputStream.
 * Compress the serialized objects with a GZIPOutputStream.
 * Write the compressed, serialized data to a file with a FileOutputStream.
 * Don't forget to flush and close the stream.
 */
public void save() {
  // Create a file dialog to query the user for a filename.
  FileDialog f = new FileDialog(frame, "Save Scribble", FileDialog.SAVE);
  f.show();                        // Display the dialog and block.
  String filename = f.getFile();   // Get the user's response
  if (filename != null) {          // If user didn't click "Cancel."
    try {
      // Create the necessary output streams to save the scribble.
      FileOutputStream fos = new FileOutputStream(filename);  // Save to file.
      GZIPOutputStream gzos = new GZIPOutputStream(fos);      // Compress.
      ObjectOutputStream out = new ObjectOutputStream(gzos);  // Save objects
      out.writeObject(lines);      // Write the entire Vector of scribbles.
      out.flush();                 // Always flush the output.
      out.close();                 // And close the stream.
    }
    // Print out exceptions.  We should really display them in a dialog...
    catch (IOException e) { System.out.println(e); }
  }
}
```

PREVIOUS | HOME | NEXT
New Feature Demo | BOOK INDEX | Custom Serialization

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

12.2 Invoking a Named Method

Example 12.2 demonstrates another use of the Reflection API. This `UniversalActionListener` object uses reflection to invoke a named method of a specified object, passing another optionally specified object as an argument. It does this in the framework of the `ActionListener` interface, so that it can serve as an action listener within a Java 1.1 GUI. By using the reflection capabilities of the `UniversalActionListener`, a program can avoid having to create a lot of trivial `ActionListener` implementations to handle action events. The `main()` method at the end of this example is a simple test GUI that demonstrates how you could use the `UniversalActionListener` object. Contrast it with the anonymous class event-handling techniques demonstrated in Chapter 7, *Events*.

Java does not allow methods to be passed directly as data values, but the Reflection API makes it possible for methods passed by name to be invoked indirectly. Note that this technique is not particularly efficient. For asynchronous event handling in a GUI, though, it is certainly efficient enough: indirect method invocation through the Reflection API will always be much faster than the response time required by the limits of human perception. Invoking a method by name is not an appropriate technique, however, when repetitive, synchronous calls are required. Thus, you should not use this technique for passing a comparison method to a sorting routine or passing a filename filter to a directory listing method. In cases like these, you should use the standard technique of implementing a class that contains the desired method and passing an instance of the class to the appropriate routine.

Example 12.2: Invoking a Named Method with Reflection

```
import java.awt.event.*;
import java.lang.reflect.*;
import java.awt.*;   // Only used for the test program below.
public class UniversalActionListener implements ActionListener {
  protected Object target;
  protected Object arg;
  protected Method m;
  public UniversalActionListener(Object target, String methodname, Object arg)
       throws NoSuchMethodException, SecurityException
  {
    this.target = target;                            // Save the target object.
    this.arg = arg;                                  // And method argument.
    // Now look up and save the Method to invoke on that target object.
    Class c, parameters[];
    c = target.getClass();                           // The Class object.
    if (arg == null) parameters = new Class[0];      // Method parameter.
    else parameters = new Class[] { arg.getClass() };
    m = c.getMethod(methodname, parameters);         // Find matching method.
  }
  public void actionPerformed(ActionEvent event) {
    Object[] arguments;
    if (arg == null) arguments = new Object[0];      // Set up arguments.
    else arguments = new Object[] { arg };
    try { m.invoke(target, arguments); }             // And invoke the method.
    catch (IllegalAccessException e) {               // Should never happen.
      System.err.println("UniversalActionListener: " + e);
    } catch (InvocationTargetException e) {          // Should never happen.
      System.err.println("UniversalActionListener: " + e);
    }
  }
  // A simple test program for the UniversalActionListener.
  public static void main(String[] args) throws NoSuchMethodException {
    Frame f = new Frame("UniversalActionListener Test");// Create window.
    f.setLayout(new FlowLayout());                      // Set layout manager.
    Button b1 = new Button("tick");                     // Create buttons.
    Button b2 = new Button("tock");
    Button b3 = new Button("Close Window");
    f.add(b1); f.add(b2); f.add(b3);                    // Add them to window.
    // Specify what the buttons do.  Invoke a named method with
    // the UniversalActionListener object.
    b1.addActionListener(new UniversalActionListener(b1, "setLabel", "tock"));
    b1.addActionListener(new UniversalActionListener(b2, "setLabel", "tick"));
    b1.addActionListener(new UniversalActionListener(b3, "hide", null));
    b2.addActionListener(new UniversalActionListener(b1, "setLabel", "tick"));
    b2.addActionListener(new UniversalActionListener(b2, "setLabel", "tock"));
    b2.addActionListener(new UniversalActionListener(b3, "show", null));
    b3.addActionListener(new UniversalActionListener(f, "dispose", null));
    f.pack();                                             // Set window size.
    f.show();                                             // And pop it up.
  }
}
```

32. Class, Method, and Field Index

The following index allows you to look up a class or interface and find out what package it is defined in. It also allows you to look up a method or field and find out what class it is defined in. Use it when you want to look up a class but don't know its package, or want to look up a method but don't know its class.

`addFocusListener()`

java.awt.Component (JDK 1.0)

`addHelpMenu()`

java.awt.peer.MenuBarPeer (JDK 1.0)

`addImage()`

java.awt.MediaTracker (JDK 1.0)

`addImpl()`

java.awt.Container (JDK 1.0), java.awt.ScrollPane (JDK 1.1)

`addInternal()`

java.awt.AWTEventMulticaster (JDK 1.1)

`addItem()`

java.awt.Choice (JDK 1.0), java.awt.peer.ChoicePeer (JDK 1.0), java.awt.List (JDK 1.0), java.awt.peer.ListPeer (JDK 1.0), java.awt.peer.MenuPeer (JDK 1.0)

`addItemListener()`

java.awt.Checkbox (JDK 1.0), java.awt.CheckboxMenuItem (JDK 1.0), java.awt.Choice (JDK 1.0), java.awt.ItemSelectable (JDK 1.1), java.awt.List (JDK 1.0)

`addKeyListener()`

java.awt.Component (JDK 1.0)

`addLayoutComponent()`

java.awt.BorderLayout (JDK 1.0), java.awt.CardLayout (JDK 1.0), java.awt.FlowLayout (JDK 1.0), java.awt.GridBagLayout (JDK 1.0), java.awt.GridLayout (JDK 1.0), java.awt.LayoutManager (JDK 1.0), java.awt.LayoutManager2 (JDK 1.1)

`addMenu()`

java.awt.peer.MenuBarPeer (JDK 1.0)

`addMouseListener()`

java.awt.Component (JDK 1.0)

`addMouseMotionListener()`

java.awt.Component (JDK 1.0)

`addNotify()`

java.awt.Button (JDK 1.0), java.awt.Canvas (JDK 1.0), java.awt.Checkbox (JDK 1.0), java.awt.CheckboxMenuItem (JDK 1.0), java.awt.Choice (JDK 1.0), java.awt.Component (JDK 1.0), java.awt.Container (JDK 1.0), java.awt.Dialog (JDK 1.0), java.awt.FileDialog (JDK 1.0), java.awt.Frame (JDK 1.0), java.awt.Label (JDK 1.0), java.awt.List (JDK 1.0), java.awt.Menu (JDK 1.0), java.awt.MenuBar (JDK 1.0), java.awt.MenuItem (JDK 1.0), java.awt.Panel (JDK 1.0), java.awt.PopupMenu (JDK 1.1), java.awt.Scrollbar (JDK 1.0), java.awt.ScrollPane (JDK 1.1), java.awt.TextArea (JDK 1.0), java.awt.TextField (JDK 1.0), java.awt.Window (JDK 1.0)

`addObserver()`

java.util.Observable (JDK 1.0)

`addPoint()`

java.awt.Polygon (JDK 1.0)

`addPropertyChangeListener()`

java.beans.Customizer (JDK 1.1), java.beans.PropertyChangeSupport (JDK 1.1), java.beans.PropertyEditor (JDK 1.1), java.beans.PropertyEditorSupport (JDK 1.1)

`address`

java.net.SocketImpl (JDK 1.0)

addSeparator()

java.awt.Menu (JDK 1.0), java.awt.peer.MenuPeer (JDK 1.0)

addTextListener()

java.awt.TextComponent (JDK 1.0)

addVetoableChangeListener()

java.beans.VetoableChangeSupport (JDK 1.1)

addWindowListener()

java.awt.Window (JDK 1.0)

Adjustable

The java.awt Package

AdjustForGravity()

java.awt.GridBagLayout (JDK 1.0)

ADJUSTMENT_EVENT_MASK

java.awt.AWTEvent (JDK 1.1)

ADJUSTMENT_FIRST

java.awt.event.AdjustmentEvent (JDK 1.1)

ADJUSTMENT_LAST

java.awt.event.AdjustmentEvent (JDK 1.1)

ADJUSTMENT_VALUE_CHANGED

java.awt.image.ImageConsumer (JDK 1.0)

`Component`

The java.awt Package

`COMPONENT_ADDED`

java.awt.event.ContainerEvent (JDK 1.1)

`COMPONENT_EVENT_MASK`

java.awt.AWTEvent (JDK 1.1)

`COMPONENT_FIRST`

java.awt.event.ComponentEvent (JDK 1.1)

`COMPONENT_HIDDEN`

java.awt.event.ComponentEvent (JDK 1.1)

`COMPONENT_LAST`

java.awt.event.ComponentEvent (JDK 1.1)

`COMPONENT_MOVED`

java.awt.event.ComponentEvent (JDK 1.1)

`COMPONENT_REMOVED`

java.awt.event.ContainerEvent (JDK 1.1)

`COMPONENT_RESIZED`

java.awt.event.ComponentEvent (JDK 1.1)

`COMPONENT_SHOWN`

createCheckboxMenuItem()

java.awt.Toolkit (JDK 1.0)

createChoice()

java.awt.Toolkit (JDK 1.0)

createComponent()

java.awt.Toolkit (JDK 1.0)

createContentHandler()

java.net.ContentHandlerFactory (JDK 1.0)

createDialog()

java.awt.Toolkit (JDK 1.0)

createFileDialog()

java.awt.Toolkit (JDK 1.0)

createFrame()

java.awt.Toolkit (JDK 1.0)

createImage()

java.awt.Component (JDK 1.0), java.awt.peer.ComponentPeer (JDK 1.0), java.awt.Toolkit (JDK 1.0)

createLabel()

java.awt.Toolkit (JDK 1.0)

createList()

`delMenu()`

java.awt.peer.MenuBarPeer (JDK 1.0)

`deselect()`

java.awt.List (JDK 1.0), java.awt.peer.ListPeer (JDK 1.0)

`DESELECTED`

java.awt.event.ItemEvent (JDK 1.1)

`desktop`

java.awt.SystemColor (JDK 1.1)

`DESKTOP`

java.awt.SystemColor (JDK 1.1)

`destHeight`

java.awt.image.ReplicateScaleFilter (JDK 1.1)

`destroy()`

java.applet.Applet (JDK 1.0), java.lang.Process (JDK 1.0), java.lang.Thread (JDK 1.0), java.lang.ThreadGroup (JDK 1.0)

`destWidth`

java.awt.image.ReplicateScaleFilter (JDK 1.1)

`detail`

java.io.WriteAbortedException (JDK 1.1)

`Dialog`

`enableEvents()`

java.awt.Component (JDK 1.0), java.awt.MenuItem (JDK 1.0)

`enableReplaceObject()`

java.io.ObjectOutputStream (JDK 1.1)

`enableResolveObject()`

java.io.ObjectInputStream (JDK 1.1)

`ENCLOSING_MARK`

java.lang.Character (JDK 1.0)

`encode()`

java.net.URLEncoder (JDK 1.0)

`END`

java.awt.Event (JDK 1.0)

`end()`

java.util.zip.Deflater (JDK 1.1), java.util.zip.Inflater (JDK 1.1), java.awt.PrintJob (JDK 1.1)

`END_PUNCTUATION`

java.lang.Character (JDK 1.0)

`endsWith()`

java.lang.String (JDK 1.0)

`endValidate()`

java.awt.peer.ContainerPeer (JDK 1.0)

java.awt.image.ColorModel (JDK 1.0), java.util.zip.Deflater (JDK 1.1), java.io.FileInputStream (JDK 1.0), java.io.FileOutputStream (JDK 1.0), java.awt.Graphics (JDK 1.0), java.util.zip.Inflater (JDK 1.1), java.lang.Object (JDK 1.0), java.awt.PrintJob (JDK 1.1)

`findEditor()`

java.beans.PropertyEditorManager (JDK 1.1)

`findLoadedClass()`

java.lang.ClassLoader (JDK 1.0)

`findSystemClass()`

java.lang.ClassLoader (JDK 1.0)

`finish()`

java.util.zip.Deflater (JDK 1.1), java.util.zip.DeflaterOutputStream (JDK 1.1), java.util.zip.GZIPOutputStream (JDK 1.1), java.util.zip.ZipOutputStream (JDK 1.1)

`finished()`

java.util.zip.Deflater (JDK 1.1), java.util.zip.Inflater (JDK 1.1)

`firePropertyChange()`

java.beans.PropertyChangeSupport (JDK 1.1), java.beans.PropertyEditorSupport (JDK 1.1)

`fireVetoableChange()`

java.beans.VetoableChangeSupport (JDK 1.1)

`first()`

java.text.BreakIterator (JDK 1.1), java.awt.CardLayout (JDK 1.0), java.text.CharacterIterator (JDK 1.1), java.text.StringCharacterIterator (JDK 1.1)

`firstElement()`

`getAdditionalBeanInfo()`

java.beans.BeanInfo (JDK 1.1), java.beans.SimpleBeanInfo (JDK 1.1)

`getAddListenerMethod()`

java.beans.EventSetDescriptor (JDK 1.1)

`getAddress()`

java.net.DatagramPacket (JDK 1.0), java.net.InetAddress (JDK 1.0)

`getAdjustable()`

java.awt.event.AdjustmentEvent (JDK 1.1)

`getAdjustmentType()`

java.awt.event.AdjustmentEvent (JDK 1.1)

`getAdler()`

java.util.zip.Deflater (JDK 1.1), java.util.zip.Inflater (JDK 1.1)

`getAlignment()`

java.awt.FlowLayout (JDK 1.0), java.awt.Label (JDK 1.0)

`getAlignmentX()`

java.awt.Component (JDK 1.0), java.awt.Container (JDK 1.0)

`getAlignmentY()`

java.awt.Component (JDK 1.0), java.awt.Container (JDK 1.0)

`getAllByName()`

java.net.InetAddress (JDK 1.0)

getBeginIndex()

java.text.CharacterIterator (JDK 1.1), java.text.FieldPosition (JDK 1.1), java.text.StringCharacterIterator (JDK 1.1)

getBlockIncrement()

java.awt.Adjustable (JDK 1.1), java.awt.Scrollbar (JDK 1.0)

getBlue()

java.awt.Color (JDK 1.0), java.awt.image.ColorModel (JDK 1.0), java.awt.image.DirectColorModel (JDK 1.0), java.awt.image.IndexColorModel (JDK 1.0)

getBlueMask()

java.awt.image.DirectColorModel (JDK 1.0)

getBlues()

java.awt.image.IndexColorModel (JDK 1.0)

getBoolean()

java.lang.reflect.Array (JDK 1.1), java.lang.Boolean (JDK 1.0), java.lang.reflect.Field (JDK 1.1)

getBoundingBox()

java.awt.Polygon (JDK 1.0)

getBounds()

java.awt.Component (JDK 1.0), java.awt.Polygon (JDK 1.0), java.awt.Rectangle (JDK 1.0), java.awt.Shape (JDK 1.1)

getBuffer()

java.io.StringWriter (JDK 1.1)

`getBundle()`

java.util.ResourceBundle (JDK 1.1)

`getByName()`

java.net.InetAddress (JDK 1.0)

`getByte()`

java.lang.reflect.Array (JDK 1.1), java.lang.reflect.Field (JDK 1.1)

`getBytes()`

java.lang.String (JDK 1.0)

`getCalendar()`

java.text.DateFormat (JDK 1.1)

`getCanonicalPath()`

java.io.File (JDK 1.0)

`getCaretPosition()`

java.awt.TextComponent (JDK 1.0), java.awt.peer.TextComponentPeer (JDK 1.0)

`getChar()`

java.lang.reflect.Array (JDK 1.1), java.lang.reflect.Field (JDK 1.1)

`getCharacterInstance()`

java.text.BreakIterator (JDK 1.1)

`getChars()`

java.lang.String (JDK 1.0), java.lang.StringBuffer (JDK 1.0)

`getClipBounds()`

java.awt.Graphics (JDK 1.0)

`getClipRect()`

java.awt.Graphics (JDK 1.0)

`getCodeBase()`

java.applet.Applet (JDK 1.0), java.applet.AppletStub (JDK 1.0)

`getCollationElementIterator()`

java.text.RuleBasedCollator (JDK 1.1)

`getCollationKey()`

java.text.Collator (JDK 1.1), java.text.RuleBasedCollator (JDK 1.1)

`getColor()`

java.awt.Color (JDK 1.0), java.awt.Graphics (JDK 1.0)

`getColorModel()`

java.awt.Component (JDK 1.0), java.awt.peer.ComponentPeer (JDK 1.0), java.awt.image.PixelGrabber (JDK 1.0), java.awt.Toolkit (JDK 1.0)

`getColumns()`

java.awt.GridLayout (JDK 1.0), java.awt.TextArea (JDK 1.0), java.awt.TextField (JDK 1.0)

`getComment()`

java.util.zip.ZipEntry (JDK 1.1)

`getComponent()`

java.awt.event.ComponentEvent (JDK 1.1), java.awt.Container (JDK 1.0)

`getComponentAt()`

java.awt.Component (JDK 1.0), java.awt.Container (JDK 1.0)

`getComponentCount()`

java.awt.Container (JDK 1.0)

`getComponents()`

java.awt.Container (JDK 1.0)

`getComponentType()`

java.lang.Class (JDK 1.0)

`getCompressedSize()`

java.util.zip.ZipEntry (JDK 1.1)

`getConstraints()`

java.awt.GridBagLayout (JDK 1.0)

`getConstructor()`

java.lang.Class (JDK 1.0)

`getConstructors()`

java.lang.Class (JDK 1.0)

`getContainer()`

java.awt.event.ContainerEvent (JDK 1.1)

`getContent()`

java.net.ContentHandler (JDK 1.0), java.net.URL (JDK 1.0), java.net.URLConnection (JDK 1.0)

getContentEncoding()

java.net.URLConnection (JDK 1.0)

getContentLength()

java.net.URLConnection (JDK 1.0)

getContents()

java.awt.datatransfer.Clipboard (JDK 1.1), java.util.ListResourceBundle (JDK 1.1)

getContentType()

java.net.URLConnection (JDK 1.0)

getContentTypeFor()

java.net.FileNameMap (JDK 1.1)

getCountry()

java.util.Locale (JDK 1.1)

getCrc()

java.util.zip.ZipEntry (JDK 1.1)

getCurrencyInstance()

java.text.NumberFormat (JDK 1.1)

getCurrent()

java.awt.CheckboxGroup (JDK 1.0)

getCursor()

java.awt.Component (JDK 1.0)

getCursorType()

java.awt.Frame (JDK 1.0)

getCustomEditor()

java.beans.PropertyEditor (JDK 1.1), java.beans.PropertyEditorSupport (JDK 1.1)

getCustomizerClass()

java.beans.BeanDescriptor (JDK 1.1)

getData()

java.net.DatagramPacket (JDK 1.0)

getDate()

java.util.Date (JDK 1.0), java.net.URLConnection (JDK 1.0)

getDateFormatSymbols()

java.text.SimpleDateFormat (JDK 1.1)

getDateInstance()

java.text.DateFormat (JDK 1.1)

getDateTimeInstance()

java.text.DateFormat (JDK 1.1)

getDay()

java.util.Date (JDK 1.0)

getDecimalFormatSymbols()

java.text.DecimalFormat (JDK 1.1)

`getDecimalSeparator()`

java.text.DecimalFormatSymbols (JDK 1.1)

`getDeclaredClasses()`

java.lang.Class (JDK 1.0)

`getDeclaredConstructor()`

java.lang.Class (JDK 1.0)

`getDeclaredConstructors()`

java.lang.Class (JDK 1.0)

`getDeclaredField()`

java.lang.Class (JDK 1.0)

`getDeclaredFields()`

java.lang.Class (JDK 1.0)

`getDeclaredMethod()`

java.lang.Class (JDK 1.0)

`getDeclaredMethods()`

java.lang.Class (JDK 1.0)

`getDeclaringClass()`

java.lang.Class (JDK 1.0), java.lang.reflect.Constructor (JDK 1.1), java.lang.reflect.Field (JDK 1.1), java.lang.reflect.Member (JDK 1.1), java.lang.reflect.Method (JDK 1.1) `getDecomposition()`

java.text.Collator (JDK 1.1)

`getDefault()`

java.util.Locale (JDK 1.1), java.util.TimeZone (JDK 1.1)

`getDefaultAllowUserInteraction()`

java.net.URLConnection (JDK 1.0)

`getDefaultCursor()`

java.awt.Cursor (JDK 1.1)

`getDefaultEventIndex()`

java.beans.BeanInfo (JDK 1.1), java.beans.SimpleBeanInfo (JDK 1.1)

`getDefaultPropertyIndex()`

java.beans.BeanInfo (JDK 1.1), java.beans.SimpleBeanInfo (JDK 1.1)

`getDefaultRequestProperty()`

java.net.URLConnection (JDK 1.0)

`getDefaultToolkit()`

java.awt.Toolkit (JDK 1.0)

`getDefaultUseCaches()`

java.net.URLConnection (JDK 1.0)

`getDescent()`

java.awt.FontMetrics (JDK 1.0)

getDigit()

java.text.DecimalFormatSymbols (JDK 1.1)

getDirectory()

java.awt.FileDialog (JDK 1.0)

getDisplayCountry()

java.util.Locale (JDK 1.1)

getDisplayLanguage()

java.util.Locale (JDK 1.1)

getDisplayName()

java.beans.FeatureDescriptor (JDK 1.1), java.util.Locale (JDK 1.1)

getDisplayVariant()

java.util.Locale (JDK 1.1)

getDocumentBase()

java.applet.Applet (JDK 1.0), java.applet.AppletStub (JDK 1.0)

getDoInput()

java.net.URLConnection (JDK 1.0)

getDoOutput()

java.net.URLConnection (JDK 1.0)

getDouble()

java.lang.reflect.Array (JDK 1.1), java.lang.reflect.Field (JDK 1.1) `getEchoChar()`

java.awt.TextField (JDK 1.0)

`getEditorSearchPath()`

java.beans.PropertyEditorManager (JDK 1.1)

`getEncoding()`

java.io.InputStreamReader (JDK 1.1), java.io.OutputStreamWriter (JDK 1.1)

`getEndIndex()`

java.text.CharacterIterator (JDK 1.1), java.text.FieldPosition (JDK 1.1), java.text.StringCharacterIterator (JDK 1.1)

`getEntry()`

java.util.zip.ZipFile (JDK 1.1)

`getenv()`

java.lang.System (JDK 1.0)

`getEras()`

java.text.DateFormatSymbols (JDK 1.1)

`getErrorOffset()`

java.text.ParseException (JDK 1.1)

`getErrorsAny()`

java.awt.MediaTracker (JDK 1.0)

`getErrorsID()`

java.awt.MediaTracker (JDK 1.0)

`getErrorStream()`

java.lang.Process (JDK 1.0)

`getEventSetDescriptors()`

java.beans.BeanInfo (JDK 1.1), java.beans.SimpleBeanInfo (JDK 1.1)

`getException()`

java.lang.ExceptionInInitializerError (JDK 1.1)

`getExceptionTypes()`

java.lang.reflect.Constructor (JDK 1.1), java.lang.reflect.Method (JDK 1.1)

`getExpiration()`

java.net.URLConnection (JDK 1.0)

`getExtra()`

java.util.zip.ZipEntry (JDK 1.1)

`getFamily()`

java.awt.Font (JDK 1.0)

`getFD()`

java.io.FileInputStream (JDK 1.0), java.io.FileOutputStream (JDK 1.0),
java.io.RandomAccessFile (JDK 1.0)

`getField()`

java.lang.Class (JDK 1.0), java.text.FieldPosition (JDK 1.1) `getFields()`

java.lang.Class (JDK 1.0)

`getFile()`

java.awt.FileDialog (JDK 1.0), java.net.URL (JDK 1.0)

`getFileDescriptor()`

java.net.DatagramSocketImpl (JDK 1.1), java.net.SocketImpl (JDK 1.0)

`getFilenameFilter()`

java.awt.FileDialog (JDK 1.0)

`getFilePointer()`

java.io.RandomAccessFile (JDK 1.0)

`getFilterInstance()`

java.awt.image.ImageFilter (JDK 1.0)

`getFirstDayOfWeek()`

java.util.Calendar (JDK 1.1)

`getFloat()`

java.lang.reflect.Array (JDK 1.1), java.lang.reflect.Field (JDK 1.1)

`getFocusOwner()`

java.awt.Window (JDK 1.0)

`getFollowRedirects()`

java.net.HttpURLConnection (JDK 1.1)

java.net.URLConnection (JDK 1.0)

`getHeight()`

java.awt.FontMetrics (JDK 1.0), java.awt.Image (JDK 1.0), java.awt.image.PixelGrabber (JDK 1.0)

`getHelpMenu()`

java.awt.MenuBar (JDK 1.0)

`getHgap()`

java.awt.BorderLayout (JDK 1.0), java.awt.CardLayout (JDK 1.0), java.awt.FlowLayout (JDK 1.0), java.awt.GridLayout (JDK 1.0)

`getHost()`

java.net.URL (JDK 1.0)

`getHostAddress()`

java.net.InetAddress (JDK 1.0)

`getHostName()`

java.net.InetAddress (JDK 1.0)

`getHours()`

java.util.Date (JDK 1.0)

`getHSBColor()`

java.awt.Color (JDK 1.0)

`getHScrollbarHeight()`

java.awt.ScrollPane (JDK 1.1), java.awt.peer.ScrollPanePeer (JDK 1.1)

java.beans.IndexedPropertyDescriptor (JDK 1.1)

`getIndexedWriteMethod()`

java.beans.IndexedPropertyDescriptor (JDK 1.1)

`getInetAddress()`

java.net.ServerSocket (JDK 1.0), java.net.Socket (JDK 1.0), java.net.SocketImpl (JDK 1.0)

`getInfinity()`

java.text.DecimalFormatSymbols (JDK 1.1)

`getInputStream()`

java.lang.Process (JDK 1.0), java.net.Socket (JDK 1.0), java.net.SocketImpl (JDK 1.0), java.net.URLConnection (JDK 1.0), java.util.zip.ZipFile (JDK 1.1)

`getInsets()`

java.awt.Container (JDK 1.0), java.awt.peer.ContainerPeer (JDK 1.0)

`getInstance()`

java.util.Calendar (JDK 1.1), java.text.Collator (JDK 1.1), java.text.DateFormat (JDK 1.1), java.text.NumberFormat (JDK 1.1)

`getInstanceOf()`

java.beans.Beans (JDK 1.1)

`getInt()`

java.lang.reflect.Array (JDK 1.1), java.lang.reflect.Field (JDK 1.1)

`getInteger()`

java.lang.Integer (JDK 1.0)

`getInterface()`

java.net.MulticastSocket (JDK 1.1)

`getInterfaces()`

java.lang.Class (JDK 1.0)

`getISO3Country()`

java.util.Locale (JDK 1.1)

`getISO3Language()`

java.util.Locale (JDK 1.1)

`getItem()`

java.awt.Choice (JDK 1.0), java.awt.event.ItemEvent (JDK 1.1), java.awt.List (JDK 1.0), java.awt.Menu (JDK 1.0)

`getItemCount()`

java.awt.Choice (JDK 1.0), java.awt.List (JDK 1.0), java.awt.Menu (JDK 1.0)

`getItems()`

java.awt.List (JDK 1.0)

`getItemSelectable()`

java.awt.event.ItemEvent (JDK 1.1)

`getJavaInitializationString()`

java.beans.PropertyEditor (JDK 1.1), java.beans.PropertyEditorSupport (JDK 1.1)

`getKey()`

java.awt.MenuShortcut (JDK 1.1), java.util.MissingResourceException (JDK 1.1)

`getKeyChar()`

java.awt.event.KeyEvent (JDK 1.1)

`getKeyCode()`

java.awt.event.KeyEvent (JDK 1.1)

`getKeyModifiersText()`

java.awt.event.KeyEvent (JDK 1.1)

`getKeys()`

java.util.ListResourceBundle (JDK 1.1), java.util.PropertyResourceBundle (JDK 1.1), java.util.ResourceBundle (JDK 1.1)

`getKeyText()`

java.awt.event.KeyEvent (JDK 1.1)

`getLabel()`

java.awt.Button (JDK 1.0), java.awt.Checkbox (JDK 1.0), java.awt.MenuItem (JDK 1.0)

`getLanguage()`

java.util.Locale (JDK 1.1)

`getLastModified()`

java.net.URLConnection (JDK 1.0)

`getLayout()`

java.awt.Container (JDK 1.0)

`getLayoutAlignmentX()`

java.awt.BorderLayout (JDK 1.0), java.awt.CardLayout (JDK 1.0), java.awt.GridBagLayout (JDK 1.0), java.awt.LayoutManager2 (JDK 1.1)

`getLayoutAlignmentY()`

java.awt.BorderLayout (JDK 1.0), java.awt.CardLayout (JDK 1.0), java.awt.GridBagLayout (JDK 1.0), java.awt.LayoutManager2 (JDK 1.1)

`getLayoutDimensions()`

java.awt.GridBagLayout (JDK 1.0)

`GetLayoutInfo()`

java.awt.GridBagLayout (JDK 1.0)

`getLayoutOrigin()`

java.awt.GridBagLayout (JDK 1.0)

`getLayoutWeights()`

java.awt.GridBagLayout (JDK 1.0)

`getLeading()`

java.awt.FontMetrics (JDK 1.0)

`getLeastMaximum()`

java.util.Calendar (JDK 1.1), java.util.GregorianCalendar (JDK 1.1)

`getLength()`

java.lang.reflect.Array (JDK 1.1), java.net.DatagramPacket (JDK 1.0)

`getLimits()`

java.text.ChoiceFormat (JDK 1.1)

`getLineIncrement()`

java.awt.Scrollbar (JDK 1.0)

`getLineInstance()`

java.text.BreakIterator (JDK 1.1)

`getLineNumber()`

java.io.LineNumberInputStream (JDK 1.0; Deprecated.), java.io.LineNumberReader (JDK 1.1)

`getListenerMethodDescriptors()`

java.beans.EventSetDescriptor (JDK 1.1)

`getListenerMethods()`

java.beans.EventSetDescriptor (JDK 1.1)

`getListenerType()`

java.beans.EventSetDescriptor (JDK 1.1)

`getLocalAddress()`

java.net.DatagramSocket (JDK 1.0), java.net.Socket (JDK 1.0)

`getLocale()`

java.applet.Applet (JDK 1.0), java.awt.Component (JDK 1.0), java.text.MessageFormat (JDK 1.1), java.awt.Window (JDK 1.0)

`getLocalHost()`

java.net.InetAddress (JDK 1.0)

`getLocalizedInputStream()`

java.lang.Runtime (JDK 1.0)

`getLocalizedMessage()`

java.lang.Throwable (JDK 1.0)

`getLocalizedOutputStream()`

java.lang.Runtime (JDK 1.0)

`getLocalPatternChars()`

java.text.DateFormatSymbols (JDK 1.1)

`getLocalPort()`

java.net.DatagramSocket (JDK 1.0), java.net.DatagramSocketImpl (JDK 1.1), java.net.ServerSocket (JDK 1.0), java.net.Socket (JDK 1.0), java.net.SocketImpl (JDK 1.0)

`getLocation()`

java.awt.Component (JDK 1.0), java.awt.Point (JDK 1.0), java.awt.Rectangle (JDK 1.0)

`getLocationOnScreen()`

java.awt.Component (JDK 1.0), java.awt.peer.ComponentPeer (JDK 1.0)

`getLong()`

java.lang.reflect.Array (JDK 1.1), java.lang.reflect.Field (JDK 1.1), java.lang.Long (JDK 1.0)

`getLowestSetBit()`

java.math.BigInteger (JDK 1.1)

`getMapSize()`

java.awt.image.IndexColorModel (JDK 1.0)

`getMaxAdvance()`

java.awt.FontMetrics (JDK 1.0)

`getMaxAscent()`

java.awt.FontMetrics (JDK 1.0)

`getMaxDecent()`

java.awt.FontMetrics (JDK 1.0)

`getMaxDescent()`

java.awt.FontMetrics (JDK 1.0)

`getMaximum()`

java.awt.Adjustable (JDK 1.1), java.util.Calendar (JDK 1.1), java.util.GregorianCalendar (JDK 1.1), java.awt.Scrollbar (JDK 1.0)

`getMaximumFractionDigits()`

java.text.NumberFormat (JDK 1.1)

`getMaximumIntegerDigits()`

java.text.NumberFormat (JDK 1.1)

`getMaximumSize()`

java.awt.Component (JDK 1.0), java.awt.Container (JDK 1.0)

`getMaxPriority()`

java.lang.ThreadGroup (JDK 1.0)

`getMenu()`

java.awt.MenuBar (JDK 1.0), java.util.ResourceBundle (JDK 1.1)

`getMenuBar()`

java.awt.Frame (JDK 1.0), java.util.ResourceBundle (JDK 1.1)

`getMenuCount()`

java.awt.MenuBar (JDK 1.0)

`getMenuShortcutKeyMask()`

java.awt.Toolkit (JDK 1.0)

`getMessage()`

java.io.InvalidClassException (JDK 1.1), java.lang.Throwable (JDK 1.0), java.io.WriteAbortedException (JDK 1.1)

`getMethod()`

java.lang.Class (JDK 1.0), java.beans.MethodDescriptor (JDK 1.1), java.util.zip.ZipEntry (JDK 1.1)

`getMethodDescriptors()`

java.beans.BeanInfo (JDK 1.1), java.beans.SimpleBeanInfo (JDK 1.1)

`getMethods()`

java.lang.Class (JDK 1.0)

`getMimeType()`

java.awt.datatransfer.DataFlavor (JDK 1.1)

`getMinimalDaysInFirstWeek()`

java.util.Calendar (JDK 1.1)

`getMinimum()`

java.awt.Adjustable (JDK 1.1), java.util.Calendar (JDK 1.1), java.util.GregorianCalendar (JDK 1.1), java.awt.Scrollbar (JDK 1.0)

`getMinimumFractionDigits()`

java.text.NumberFormat (JDK 1.1)

`getMinimumIntegerDigits()`

java.text.NumberFormat (JDK 1.1)

`getMinimumSize()`

java.awt.Component (JDK 1.0), java.awt.peer.ComponentPeer (JDK 1.0), java.awt.Container (JDK 1.0), java.awt.List (JDK 1.0), java.awt.peer.ListPeer (JDK 1.0), java.awt.TextArea (JDK 1.0), java.awt.peer.TextAreaPeer (JDK 1.0), java.awt.TextField (JDK 1.0), java.awt.peer.TextFieldPeer (JDK 1.0)

`GetMinSize()`

java.awt.GridBagLayout (JDK 1.0)

`getMinusSign()`

java.text.DecimalFormatSymbols (JDK 1.1)

`getMinutes()`

java.util.Date (JDK 1.0)

`getMode()`

java.awt.FileDialog (JDK 1.0)

`getModifiers()`

java.awt.event.ActionEvent (JDK 1.1), java.lang.Class (JDK 1.0), java.lang.reflect.Constructor (JDK 1.1), java.lang.reflect.Field (JDK 1.1), java.awt.event.InputEvent (JDK 1.1), java.lang.reflect.Member (JDK 1.1), java.lang.reflect.Method (JDK 1.1)

getMonth()

java.util.Date (JDK 1.0)

getMonths()

java.text.DateFormatSymbols (JDK 1.1)

getMultiplier()

java.text.DecimalFormat (JDK 1.1)

getName()

java.lang.Class (JDK 1.0), java.awt.datatransfer.Clipboard (JDK 1.1), java.awt.Component (JDK 1.0), java.lang.reflect.Constructor (JDK 1.1), java.beans.FeatureDescriptor (JDK 1.1), java.lang.reflect.Field (JDK 1.1), java.io.File (JDK 1.0), java.awt.Font (JDK 1.0), java.lang.reflect.Member (JDK 1.1), java.awt.MenuComponent (JDK 1.0), java.lang.reflect.Method (JDK 1.1), java.io.ObjectStreamClass (JDK 1.1), java.lang.Thread (JDK 1.0), java.lang.ThreadGroup (JDK 1.0), java.util.zip.ZipEntry (JDK 1.1), java.util.zip.ZipFile (JDK 1.1)

getNaN()

java.text.DecimalFormatSymbols (JDK 1.1)

getNativeContainer()

java.awt.Toolkit (JDK 1.0)

getNegativePrefix()

java.text.DecimalFormat (JDK 1.1)

`getNegativeSuffix()`

java.text.DecimalFormat (JDK 1.1)

`getNewValue()`

java.beans.PropertyChangeEvent (JDK 1.1)

`getNextEntry()`

java.util.zip.ZipInputStream (JDK 1.1)

`getNextEvent()`

java.awt.EventQueue (JDK 1.1)

`getNumberFormat()`

java.text.DateFormat (JDK 1.1)

`getNumberInstance()`

java.text.NumberFormat (JDK 1.1)

`getNumericValue()`

java.lang.Character (JDK 1.0)

`getObject()`

java.util.ResourceBundle (JDK 1.1)

`getOffset()`

java.util.SimpleTimeZone (JDK 1.1), java.util.TimeZone (JDK 1.1)

`getOldValue()`

java.beans.PropertyChangeEvent (JDK 1.1) `getOption()`

java.net.DatagramSocketImpl (JDK 1.1), java.net.SocketImpl (JDK 1.0)

`getOrientation()`

java.awt.Adjustable (JDK 1.1), java.awt.Scrollbar (JDK 1.0)

`getOutputStream()`

java.lang.Process (JDK 1.0), java.net.Socket (JDK 1.0), java.net.SocketImpl (JDK 1.0), java.net.URLConnection (JDK 1.0)

`getPageDimension()`

java.awt.PrintJob (JDK 1.1)

`getPageIncrement()`

java.awt.Scrollbar (JDK 1.0)

`getPageResolution()`

java.awt.PrintJob (JDK 1.1)

`getParameter()`

java.applet.Applet (JDK 1.0), java.applet.AppletStub (JDK 1.0)

`getParameterDescriptors()`

java.beans.MethodDescriptor (JDK 1.1)

`getParameterInfo()`

java.applet.Applet (JDK 1.0)

`getParameterTypes()`

java.lang.reflect.Constructor (JDK 1.1), java.lang.reflect.Method (JDK 1.1)

`getParent()`

java.awt.Component (JDK 1.0), java.io.File (JDK 1.0), java.awt.MenuComponent (JDK 1.0), java.lang.ThreadGroup (JDK 1.0)

`getPath()`

java.io.File (JDK 1.0)

`getPatternSeparator()`

java.text.DecimalFormatSymbols (JDK 1.1)

`getPeer()`

java.awt.Component (JDK 1.0), java.awt.Font (JDK 1.0), java.awt.MenuComponent (JDK 1.0)

`getPercent()`

java.text.DecimalFormatSymbols (JDK 1.1)

`getPercentInstance()`

java.text.NumberFormat (JDK 1.1)

`getPerMill()`

java.text.DecimalFormatSymbols (JDK 1.1)

`getPixels()`

java.awt.image.PixelGrabber (JDK 1.0)

`getPixelSize()`

java.awt.image.ColorModel (JDK 1.0)

`getPoint()`

java.awt.event.MouseEvent (JDK 1.1)

`getPort()`

java.net.DatagramPacket (JDK 1.0), java.net.Socket (JDK 1.0), java.net.SocketImpl (JDK 1.0), java.net.URL (JDK 1.0)

`getPositivePrefix()`

java.text.DecimalFormat (JDK 1.1)

`getPositiveSuffix()`

java.text.DecimalFormat (JDK 1.1)

`getPredefinedCursor()`

java.awt.Cursor (JDK 1.1)

`getPreferredSize()`

java.awt.Component (JDK 1.0), java.awt.peer.ComponentPeer (JDK 1.0), java.awt.Container (JDK 1.0), java.awt.List (JDK 1.0), java.awt.peer.ListPeer (JDK 1.0), java.awt.TextArea (JDK 1.0), java.awt.peer.TextAreaPeer (JDK 1.0), java.awt.TextField (JDK 1.0), java.awt.peer.TextFieldPeer (JDK 1.0)

`getPrintJob()`

java.awt.PrintGraphics (JDK 1.1), java.awt.Toolkit (JDK 1.0)

`getPriority()`

java.lang.Thread (JDK 1.0)

`getPropagationId()`

java.beans.PropertyChangeEvent (JDK 1.1) `getProperties()`

java.lang.System (JDK 1.0)

`getProperty()`

java.awt.Image (JDK 1.0), java.util.Properties (JDK 1.0), java.lang.System (JDK 1.0), java.awt.Toolkit (JDK 1.0)

`getPropertyChangeEvent()`

java.beans.PropertyVetoException (JDK 1.1)

`getPropertyDescriptors()`

java.beans.BeanInfo (JDK 1.1), java.beans.SimpleBeanInfo (JDK 1.1)

`getPropertyEditorClass()`

java.beans.PropertyDescriptor (JDK 1.1)

`getPropertyName()`

java.beans.PropertyChangeEvent (JDK 1.1)

`getPropertyType()`

java.beans.PropertyDescriptor (JDK 1.1)

`getProtocol()`

java.net.URL (JDK 1.0)

`getRawOffset()`

java.util.SimpleTimeZone (JDK 1.1), java.util.TimeZone (JDK 1.1)

`getReadMethod()`

java.beans.PropertyDescriptor (JDK 1.1)

`getRed()`

java.awt.Color (JDK 1.0), java.awt.image.ColorModel (JDK 1.0), java.awt.image.DirectColorModel (JDK 1.0), java.awt.image.IndexColorModel (JDK 1.0)

`getRedMask()`

java.awt.image.DirectColorModel (JDK 1.0)

`getReds()`

java.awt.image.IndexColorModel (JDK 1.0)

`getRef()`

java.net.URL (JDK 1.0)

`getRemaining()`

java.util.zip.Inflater (JDK 1.1)

`getRemoveListenerMethod()`

java.beans.EventSetDescriptor (JDK 1.1)

`getRepresentationClass()`

java.awt.datatransfer.DataFlavor (JDK 1.1)

`getRequestMethod()`

java.net.HttpURLConnection (JDK 1.1)

`getRequestProperty()`

java.net.URLConnection (JDK 1.0)

`getResource()`

java.lang.Class (JDK 1.0), java.lang.ClassLoader (JDK 1.0)

`getResourceAsStream()`

java.lang.Class (JDK 1.0), java.lang.ClassLoader (JDK 1.0)

`getResponseCode()`

java.net.HttpURLConnection (JDK 1.1)

`getResponseMessage()`

java.net.HttpURLConnection (JDK 1.1)

`getReturnType()`

java.lang.reflect.Method (JDK 1.1)

`getRGB()`

java.awt.Color (JDK 1.0), java.awt.image.ColorModel (JDK 1.0), java.awt.image.DirectColorModel (JDK 1.0), java.awt.image.IndexColorModel (JDK 1.0), java.awt.SystemColor (JDK 1.1)

`getRGBdefault()`

java.awt.image.ColorModel (JDK 1.0)

`getRows()`

java.awt.GridLayout (JDK 1.0), java.awt.List (JDK 1.0), java.awt.TextArea (JDK 1.0)

`getRules()`

java.text.RuleBasedCollator (JDK 1.1)

`getRuntime()`

java.lang.Runtime (JDK 1.0)

getScaledInstance()

java.awt.Image (JDK 1.0)

getScreenResolution()

java.awt.Toolkit (JDK 1.0)

getScreenSize()

java.awt.Toolkit (JDK 1.0)

getScrollbarDisplayPolicy()

java.awt.ScrollPane (JDK 1.1)

getScrollbarVisibility()

java.awt.TextArea (JDK 1.0)

getScrollPosition()

java.awt.ScrollPane (JDK 1.1)

getSeconds()

java.util.Date (JDK 1.0)

getSecurityContext()

java.lang.SecurityManager (JDK 1.0)

getSecurityManager()

java.lang.System (JDK 1.0)

getSelectedCheckbox()

java.awt.CheckboxGroup (JDK 1.0)

`getSelectedIndex()`

java.awt.Choice (JDK 1.0), java.awt.List (JDK 1.0)

`getSelectedIndexes()`

java.awt.List (JDK 1.0), java.awt.peer.ListPeer (JDK 1.0)

`getSelectedItem()`

java.awt.Choice (JDK 1.0), java.awt.List (JDK 1.0)

`getSelectedItems()`

java.awt.List (JDK 1.0)

`getSelectedObjects()`

java.awt.Checkbox (JDK 1.0), java.awt.CheckboxMenuItem (JDK 1.0), java.awt.Choice (JDK 1.0), java.awt.ItemSelectable (JDK 1.1), java.awt.List (JDK 1.0)

`getSelectedText()`

java.awt.TextComponent (JDK 1.0)

`getSelectionEnd()`

java.awt.TextComponent (JDK 1.0), java.awt.peer.TextComponentPeer (JDK 1.0)

`getSelectionStart()`

java.awt.TextComponent (JDK 1.0), java.awt.peer.TextComponentPeer (JDK 1.0)

`getSentenceInstance()`

java.text.BreakIterator (JDK 1.1)

`getSerialVersionUID()`

java.io.ObjectStreamClass (JDK 1.1)

`getShort()`

java.lang.reflect.Array (JDK 1.1), java.lang.reflect.Field (JDK 1.1)

`getShortcut()`

java.awt.MenuItem (JDK 1.0)

`getShortcutMenuItem()`

java.awt.MenuBar (JDK 1.0)

`getShortDescription()`

java.beans.FeatureDescriptor (JDK 1.1)

`getShortMonths()`

java.text.DateFormatSymbols (JDK 1.1)

`getShortWeekdays()`

java.text.DateFormatSymbols (JDK 1.1)

`getSigners()`

java.lang.Class (JDK 1.0)

`getSize()`

java.awt.Component (JDK 1.0), java.awt.Dimension (JDK 1.0), java.awt.Font (JDK 1.0), java.awt.Rectangle (JDK 1.0), java.util.zip.ZipEntry (JDK 1.1)

`getSoLinger()`

java.net.Socket (JDK 1.0)

getSoTimeout()

java.net.DatagramSocket (JDK 1.0), java.net.ServerSocket (JDK 1.0), java.net.Socket (JDK 1.0)

getSource()

java.util.EventObject (JDK 1.1), java.awt.Image (JDK 1.0)

getSourceString()

java.text.CollationKey (JDK 1.1)

getState()

java.awt.Checkbox (JDK 1.0), java.awt.CheckboxMenuItem (JDK 1.0)

getStateChange()

java.awt.event.ItemEvent (JDK 1.1)

getStatus()

java.awt.image.PixelGrabber (JDK 1.0)

getStrength()

java.text.Collator (JDK 1.1)

getString()

java.util.ResourceBundle (JDK 1.1)

getStringArray()

java.util.ResourceBundle (JDK 1.1)

getStyle()

java.awt.Font (JDK 1.0)

getSuperclass()

java.lang.Class (JDK 1.0)

getSystemClipboard()

java.awt.Toolkit (JDK 1.0)

getSystemEventQueue()

java.awt.Toolkit (JDK 1.0)

getSystemEventQueueImpl()

java.awt.Toolkit (JDK 1.0)

getSystemResource()

java.lang.ClassLoader (JDK 1.0)

getSystemResourceAsStream()

java.lang.ClassLoader (JDK 1.0)

getTags()

java.beans.PropertyEditor (JDK 1.1), java.beans.PropertyEditorSupport (JDK 1.1)

getTargetException()

java.lang.reflect.InvocationTargetException (JDK 1.1)

getTcpNoDelay()

java.net.Socket (JDK 1.0)

getText()

`getTotalOut()`

java.util.zip.Deflater (JDK 1.1), java.util.zip.Inflater (JDK 1.1)

`getTransferData()`

java.awt.datatransfer.StringSelection (JDK 1.1), java.awt.datatransfer.Transferable (JDK 1.1)

`getTransferDataFlavors()`

java.awt.datatransfer.StringSelection (JDK 1.1), java.awt.datatransfer.Transferable (JDK 1.1)

`getTransparentPixel()`

java.awt.image.IndexColorModel (JDK 1.0)

`getTreeLock()`

java.awt.Component (JDK 1.0)

`getTTL()`

java.net.DatagramSocketImpl (JDK 1.1), java.net.MulticastSocket (JDK 1.1)

`getType()`

java.lang.Character (JDK 1.0), java.awt.Cursor (JDK 1.1), java.lang.reflect.Field (JDK 1.1)

`getUnitIncrement()`

java.awt.Adjustable (JDK 1.1), java.awt.Scrollbar (JDK 1.0)

`getUpdateRect()`

java.awt.event.PaintEvent (JDK 1.1)

`getURL()`

java.net.URLConnection (JDK 1.0)

getUseCaches()

java.net.URLConnection (JDK 1.0)

getVAdjustable()

java.awt.ScrollPane (JDK 1.1)

getValue()

java.awt.Adjustable (JDK 1.1), java.awt.event.AdjustmentEvent (JDK 1.1), java.util.zip.Adler32 (JDK 1.1), java.util.zip.Checksum (JDK 1.1), java.util.zip.CRC32 (JDK 1.1), java.beans.FeatureDescriptor (JDK 1.1), java.beans.PropertyEditor (JDK 1.1), java.beans.PropertyEditorSupport (JDK 1.1), java.awt.Scrollbar (JDK 1.0)

getVariant()

java.util.Locale (JDK 1.1)

getVgap()

java.awt.BorderLayout (JDK 1.0), java.awt.CardLayout (JDK 1.0), java.awt.FlowLayout (JDK 1.0), java.awt.GridLayout (JDK 1.0)

getViewportSize()

java.awt.ScrollPane (JDK 1.1)

getVisible()

java.awt.Scrollbar (JDK 1.0)

getVisibleAmount()

java.awt.Adjustable (JDK 1.1), java.awt.Scrollbar (JDK 1.0)

getVisibleIndex()

hashCode()

java.math.BigDecimal (JDK 1.1), java.math.BigInteger (JDK 1.1), java.util.BitSet (JDK 1.0), java.lang.Boolean (JDK 1.0), java.lang.Byte (JDK 1.1), java.lang.Character (JDK 1.0), java.text.ChoiceFormat (JDK 1.1), java.text.CollationKey (JDK 1.1), java.text.Collator (JDK 1.1), java.awt.Color (JDK 1.0), java.lang.reflect.Constructor (JDK 1.1), java.util.Date (JDK 1.0), java.text.DateFormat (JDK 1.1), java.text.DateFormatSymbols (JDK 1.1), java.text.DecimalFormat (JDK 1.1), java.text.DecimalFormatSymbols (JDK 1.1), java.lang.Double (JDK 1.0), java.lang.reflect.Field (JDK 1.1), java.io.File (JDK 1.0), java.lang.Float (JDK 1.0), java.awt.Font (JDK 1.0), java.util.GregorianCalendar (JDK 1.1), java.net.InetAddress (JDK 1.0), java.lang.Integer (JDK 1.0), java.util.Locale (JDK 1.1), java.lang.Long (JDK 1.0), java.text.MessageFormat (JDK 1.1), java.lang.reflect.Method (JDK 1.1), java.text.NumberFormat (JDK 1.1), java.lang.Object (JDK 1.0), java.awt.Point (JDK 1.0), java.awt.Rectangle (JDK 1.0), java.text.RuleBasedCollator (JDK 1.1), java.lang.Short (JDK 1.1), java.text.SimpleDateFormat (JDK 1.1), java.util.SimpleTimeZone (JDK 1.1), java.lang.String (JDK 1.0), java.text.StringCharacterIterator (JDK 1.1), java.net.URL (JDK 1.0)

Hashtable

The java.util Package

hasMoreElements()

java.util.Enumeration (JDK 1.0), java.util.StringTokenizer (JDK 1.0)

hasMoreTokens()

java.util.StringTokenizer (JDK 1.0)

HEIGHT

java.awt.image.ImageObserver (JDK 1.0)

height

java.awt.Dimension (JDK 1.0), java.awt.Rectangle (JDK 1.0)

hide()

HTTP_ACCEPTED

java.net.HttpURLConnection (JDK 1.1)

HTTP_BAD_GATEWAY

java.net.HttpURLConnection (JDK 1.1)

HTTP_BAD_METHOD

java.net.HttpURLConnection (JDK 1.1)

HTTP_BAD_REQUEST

java.net.HttpURLConnection (JDK 1.1)

HTTP_CLIENT_TIMEOUT

java.net.HttpURLConnection (JDK 1.1)

HTTP_CONFLICT

java.net.HttpURLConnection (JDK 1.1)

HTTP_CREATED

java.net.HttpURLConnection (JDK 1.1)

HTTP_ENTITY_TOO_LARGE

java.net.HttpURLConnection (JDK 1.1)

HTTP_FORBIDDEN

java.net.HttpURLConnection (JDK 1.1)

HTTP_GATEWAY_TIMEOUT

java.net.HttpURLConnection (JDK 1.1)

HTTP_GONE

java.net.HttpURLConnection (JDK 1.1)

HTTP_INTERNAL_ERROR

java.net.HttpURLConnection (JDK 1.1)

HTTP_LENGTH_REQUIRED

java.net.HttpURLConnection (JDK 1.1)

HTTP_MOVED_PERM

java.net.HttpURLConnection (JDK 1.1)

HTTP_MOVED_TEMP

java.net.HttpURLConnection (JDK 1.1)

HTTP_MULT_CHOICE

java.net.HttpURLConnection (JDK 1.1)

HTTP_NO_CONTENT

java.net.HttpURLConnection (JDK 1.1)

HTTP_NOT_ACCEPTABLE

java.net.HttpURLConnection (JDK 1.1)

HTTP_NOT_AUTHORITATIVE

java.net.HttpURLConnection (JDK 1.1)

HTTP_NOT_FOUND

java.net.HttpURLConnection (JDK 1.1)

HTTP_NOT_MODIFIED

java.net.HttpURLConnection (JDK 1.1)

HTTP_OK

java.net.HttpURLConnection (JDK 1.1)

HTTP_PARTIAL

java.net.HttpURLConnection (JDK 1.1)

HTTP_PAYMENT_REQUIRED

java.net.HttpURLConnection (JDK 1.1)

HTTP_PRECON_FAILED

java.net.HttpURLConnection (JDK 1.1)

HTTP_PROXY_AUTH

java.net.HttpURLConnection (JDK 1.1)

HTTP_REQ_TOO_LONG

java.net.HttpURLConnection (JDK 1.1)

HTTP_RESET

java.net.HttpURLConnection (JDK 1.1)

HTTP_SEE_OTHER

java.net.HttpURLConnection (JDK 1.1)

HTTP_SERVER_ERROR

java.net.HttpURLConnection (JDK 1.1)

`isDecimalSeparatorAlwaysShown()`

java.text.DecimalFormat (JDK 1.1)

`isDefined()`

java.lang.Character (JDK 1.0)

`isDesignTime()`

java.beans.Beans (JDK 1.1)

`isDestroyed()`

java.lang.ThreadGroup (JDK 1.0)

`isDigit()`

java.lang.Character (JDK 1.0)

`isDirectory()`

java.io.File (JDK 1.0), java.util.zip.ZipEntry (JDK 1.1)

`isEditable()`

java.awt.TextComponent (JDK 1.0)

`isEmpty()`

java.util.Dictionary (JDK 1.0), java.util.Hashtable (JDK 1.0), java.awt.Rectangle (JDK 1.0), java.util.Vector (JDK 1.0)

`isEnabled()`

java.awt.Component (JDK 1.0), java.awt.MenuItem (JDK 1.0)

`isErrorAny()`

java.awt.Dialog (JDK 1.0)

`isMulticastAddress()`

java.net.InetAddress (JDK 1.0)

`isMultipleMode()`

java.awt.List (JDK 1.0)

`isNaN()`

java.lang.Double (JDK 1.0), java.lang.Float (JDK 1.0)

`isNative()`

java.lang.reflect.Modifier (JDK 1.1)

`isPaintable()`

java.beans.PropertyEditor (JDK 1.1), java.beans.PropertyEditorSupport (JDK 1.1)

`isParseIntegerOnly()`

java.text.NumberFormat (JDK 1.1)

`isPlain()`

java.awt.Font (JDK 1.0)

`isPopupTrigger()`

java.awt.event.MouseEvent (JDK 1.1)

`isPrimitive()`

java.lang.Class (JDK 1.0)

isSpace()

java.lang.Character (JDK 1.0)

isSpaceChar()

java.lang.Character (JDK 1.0)

isStatic()

java.lang.reflect.Modifier (JDK 1.1)

isSynchronized()

java.lang.reflect.Modifier (JDK 1.1)

isTearOff()

java.awt.Menu (JDK 1.0)

isTemporary()

java.awt.event.FocusEvent (JDK 1.1)

isTimeSet

java.util.Calendar (JDK 1.1)

isTitleCase()

java.lang.Character (JDK 1.0)

isTransient()

java.lang.reflect.Modifier (JDK 1.1)

isUnicast()

java.beans.EventSetDescriptor (JDK 1.1)

MOUSE_DOWN

java.awt.Event (JDK 1.0)

MOUSE_DRAG

java.awt.Event (JDK 1.0)

MOUSE_DRAGGED

java.awt.event.MouseEvent (JDK 1.1)

MOUSE_ENTER

java.awt.Event (JDK 1.0)

MOUSE_ENTERED

java.awt.event.MouseEvent (JDK 1.1)

MOUSE_EVENT_MASK

java.awt.AWTEvent (JDK 1.1)

MOUSE_EXIT

java.awt.Event (JDK 1.0)

MOUSE_EXITED

java.awt.event.MouseEvent (JDK 1.1)

MOUSE_FIRST

java.awt.event.MouseEvent (JDK 1.1)

MOUSE_LAST

java.awt.event.MouseEvent (JDK 1.1)

The java.awt.event Package

mouseMove()

java.awt.Component (JDK 1.0)

mouseMoved()

java.awt.AWTEventMulticaster (JDK 1.1), java.awt.event.MouseMotionAdapter (JDK 1.1), java.awt.event.MouseMotionListener (JDK 1.1)

mousePressed()

java.awt.AWTEventMulticaster (JDK 1.1), java.awt.event.MouseAdapter (JDK 1.1), java.awt.event.MouseListener (JDK 1.1)

mouseReleased()

java.awt.AWTEventMulticaster (JDK 1.1), java.awt.event.MouseAdapter (JDK 1.1), java.awt.event.MouseListener (JDK 1.1)

mouseUp()

java.awt.Component (JDK 1.0)

move()

java.awt.Component (JDK 1.0), java.awt.Point (JDK 1.0), java.awt.Rectangle (JDK 1.0)

MOVE_CURSOR

java.awt.Cursor (JDK 1.1), java.awt.Frame (JDK 1.0)

movePointLeft()

java.math.BigDecimal (JDK 1.1)

movePointRight()

`readLine()`

java.io.BufferedReader (JDK 1.1), java.io.DataInput (JDK 1.0), java.io.DataInputStream (JDK 1.0), java.io.LineNumberReader (JDK 1.1), java.io.ObjectInputStream (JDK 1.1), java.io.RandomAccessFile (JDK 1.0)

`readLong()`

java.io.DataInput (JDK 1.0), java.io.DataInputStream (JDK 1.0), java.io.ObjectInputStream (JDK 1.1), java.io.RandomAccessFile (JDK 1.0)

`readObject()`

java.io.ObjectInput (JDK 1.1), java.io.ObjectInputStream (JDK 1.1)

`readShort()`

java.io.DataInput (JDK 1.0), java.io.DataInputStream (JDK 1.0), java.io.ObjectInputStream (JDK 1.1), java.io.RandomAccessFile (JDK 1.0)

`readStreamHeader()`

java.io.ObjectInputStream (JDK 1.1)

`readUnsignedByte()`

java.io.DataInput (JDK 1.0), java.io.DataInputStream (JDK 1.0), java.io.ObjectInputStream (JDK 1.1), java.io.RandomAccessFile (JDK 1.0)

`readUnsignedShort()`

java.io.DataInput (JDK 1.0), java.io.DataInputStream (JDK 1.0), java.io.ObjectInputStream (JDK 1.1), java.io.RandomAccessFile (JDK 1.0)

`readUTF()`

java.io.DataInput (JDK 1.0), java.io.DataInputStream (JDK 1.0), java.io.ObjectInputStream (JDK 1.1), java.io.RandomAccessFile (JDK 1.0)

java.awt.image.FilteredImageSource (JDK 1.0), java.awt.image.ImageProducer (JDK 1.0), java.awt.image.MemoryImageSource (JDK 1.0)

`removeContainerListener()`

java.awt.Container (JDK 1.0)

`removeElement()`

java.util.Vector (JDK 1.0)

`removeElementAt()`

java.util.Vector (JDK 1.0)

`removeFocusListener()`

java.awt.Component (JDK 1.0)

`removeImage()`

java.awt.MediaTracker (JDK 1.0)

`removeInternal()`

java.awt.AWTEventMulticaster (JDK 1.1)

`removeItemListener()`

java.awt.Checkbox (JDK 1.0), java.awt.CheckboxMenuItem (JDK 1.0), java.awt.Choice (JDK 1.0), java.awt.ItemSelectable (JDK 1.1), java.awt.List (JDK 1.0)

`removeKeyListener()`

java.awt.Component (JDK 1.0)

`removeLayoutComponent()`

java.awt.BorderLayout (JDK 1.0), java.awt.CardLayout (JDK 1.0), java.awt.FlowLayout (JDK

1.0), java.awt.GridBagLayout (JDK 1.0), java.awt.GridLayout (JDK 1.0), java.awt.LayoutManager (JDK 1.0)

`removeMouseListener()`

java.awt.Component (JDK 1.0)

`removeMouseMotionListener()`

java.awt.Component (JDK 1.0)

`removeNotify()`

java.awt.Component (JDK 1.0), java.awt.Container (JDK 1.0), java.awt.List (JDK 1.0), java.awt.Menu (JDK 1.0), java.awt.MenuBar (JDK 1.0), java.awt.MenuComponent (JDK 1.0), java.awt.TextComponent (JDK 1.0)

`removePropertyChangeListener()`

java.beans.Customizer (JDK 1.1), java.beans.PropertyChangeSupport (JDK 1.1), java.beans.PropertyEditor (JDK 1.1), java.beans.PropertyEditorSupport (JDK 1.1)

`removeTextListener()`

java.awt.TextComponent (JDK 1.0)

`removeVetoableChangeListener()`

java.beans.VetoableChangeSupport (JDK 1.1)

`removeWindowListener()`

java.awt.Window (JDK 1.0)

`renameTo()`

java.io.File (JDK 1.0)

`repaint()`

RESERVED_ID_MAX

java.awt.AWTEvent (JDK 1.1)

reset()

java.util.zip.Adler32 (JDK 1.1), java.io.BufferedInputStream (JDK 1.0), java.io.BufferedReader (JDK 1.1), java.io.ByteArrayInputStream (JDK 1.0), java.io.ByteArrayOutputStream (JDK 1.0), java.io.CharArrayReader (JDK 1.1), java.io.CharArrayWriter (JDK 1.1), java.util.zip.Checksum (JDK 1.1), java.text.CollationElementIterator (JDK 1.1), java.util.zip.CRC32 (JDK 1.1), java.util.zip.Deflater (JDK 1.1), java.io.FilterInputStream (JDK 1.0), java.io.FilterReader (JDK 1.1), java.util.zip.Inflater (JDK 1.1), java.io.InputStream (JDK 1.0), java.io.LineNumberInputStream (JDK 1.0; Deprecated.), java.io.LineNumberReader (JDK 1.1), java.io.ObjectOutputStream (JDK 1.1), java.io.Reader (JDK 1.1), java.io.StringBufferInputStream (JDK 1.0; Deprecated.), java.io.StringReader (JDK 1.1)

resetSyntax()

java.io.StreamTokenizer (JDK 1.0)

reshape()

java.awt.Component (JDK 1.0), java.awt.peer.ComponentPeer (JDK 1.0), java.awt.Rectangle (JDK 1.0)

resize()

java.applet.Applet (JDK 1.0), java.awt.Component (JDK 1.0), java.awt.Rectangle (JDK 1.0)

resolveClass()

java.lang.ClassLoader (JDK 1.0), java.io.ObjectInputStream (JDK 1.1)

resolveObject()

java.io.ObjectInputStream (JDK 1.1)

ResourceBundle

java.lang.Math (JDK 1.0)

roll()

java.util.Calendar (JDK 1.1), java.util.GregorianCalendar (JDK 1.1)

round()

java.lang.Math (JDK 1.0)

ROUND_CEILING

java.math.BigDecimal (JDK 1.1)

ROUND_DOWN

java.math.BigDecimal (JDK 1.1)

ROUND_FLOOR

java.math.BigDecimal (JDK 1.1)

ROUND_HALF_DOWN

java.math.BigDecimal (JDK 1.1)

ROUND_HALF_EVEN

java.math.BigDecimal (JDK 1.1)

ROUND_HALF_UP

java.math.BigDecimal (JDK 1.1)

ROUND_UNNECESSARY

java.math.BigDecimal (JDK 1.1)

ROUND_UP

setAmPmStrings()

java.text.DateFormatSymbols (JDK 1.1)

setAnimated()

java.awt.image.MemoryImageSource (JDK 1.0)

setAsText()

java.beans.PropertyEditor (JDK 1.1), java.beans.PropertyEditorSupport (JDK 1.1)

setBackground()

java.awt.Component (JDK 1.0), java.awt.peer.ComponentPeer (JDK 1.0)

setBeanInfoSearchPath()

java.beans.Introspector (JDK 1.1)

setBit()

java.math.BigInteger (JDK 1.1)

setBlockIncrement()

java.awt.Adjustable (JDK 1.1), java.awt.Scrollbar (JDK 1.0)

setBoolean()

java.lang.reflect.Array (JDK 1.1), java.lang.reflect.Field (JDK 1.1)

setBound()

java.beans.PropertyDescriptor (JDK 1.1)

setBounds()

java.awt.Component (JDK 1.0), java.awt.peer.ComponentPeer (JDK 1.0), java.awt.Rectangle

(JDK 1.0)

setByte()

java.lang.reflect.Array (JDK 1.1), java.lang.reflect.Field (JDK 1.1)

setCalendar()

java.text.DateFormat (JDK 1.1)

setCaretPosition()

java.awt.TextComponent (JDK 1.0), java.awt.peer.TextComponentPeer (JDK 1.0)

setChanged()

java.util.Observable (JDK 1.0)

setChar()

java.lang.reflect.Array (JDK 1.1), java.lang.reflect.Field (JDK 1.1)

setCharAt()

java.lang.StringBuffer (JDK 1.0)

setCheckboxGroup()

java.awt.Checkbox (JDK 1.0), java.awt.peer.CheckboxPeer (JDK 1.0)

setChoices()

java.text.ChoiceFormat (JDK 1.1)

setClip()

java.awt.Graphics (JDK 1.0)

setColor()

java.awt.Graphics (JDK 1.0)

`setColorModel()`

java.awt.image.ImageConsumer (JDK 1.0), java.awt.image.ImageFilter (JDK 1.0), java.awt.image.PixelGrabber (JDK 1.0), java.awt.image.RGBImageFilter (JDK 1.0)

`setColumns()`

java.awt.GridLayout (JDK 1.0), java.awt.TextArea (JDK 1.0), java.awt.TextField (JDK 1.0)

`setComment()`

java.util.zip.ZipEntry (JDK 1.1), java.util.zip.ZipOutputStream (JDK 1.1)

`setConstrained()`

java.beans.PropertyDescriptor (JDK 1.1)

`setConstraints()`

java.awt.GridBagLayout (JDK 1.0)

`setContentHandlerFactory()`

java.net.URLConnection (JDK 1.0)

`setContents()`

java.awt.datatransfer.Clipboard (JDK 1.1)

`setCrc()`

java.util.zip.ZipEntry (JDK 1.1)

`setCurrent()`

java.awt.CheckboxGroup (JDK 1.0)

java.util.Locale (JDK 1.1), java.util.TimeZone (JDK 1.1)

`setDefaultAllowUserInteraction()`

java.net.URLConnection (JDK 1.0)

`setDefaultRequestProperty()`

java.net.URLConnection (JDK 1.0)

`setDefaultUseCaches()`

java.net.URLConnection (JDK 1.0)

`setDesignTime()`

java.beans.Beans (JDK 1.1)

`setDictionary()`

java.util.zip.Deflater (JDK 1.1), java.util.zip.Inflater (JDK 1.1)

`setDigit()`

java.text.DecimalFormatSymbols (JDK 1.1)

`setDimensions()`

java.awt.image.CropImageFilter (JDK 1.0), java.awt.image.ImageConsumer (JDK 1.0), java.awt.image.ImageFilter (JDK 1.0), java.awt.image.PixelGrabber (JDK 1.0), java.awt.image.ReplicateScaleFilter (JDK 1.1)

`setDirectory()`

java.awt.FileDialog (JDK 1.0), java.awt.peer.FileDialogPeer (JDK 1.0)

`setDisplayName()`

java.beans.FeatureDescriptor (JDK 1.1)

java.util.SimpleTimeZone (JDK 1.1)

setEras()

java.text.DateFormatSymbols (JDK 1.1)

setErr()

java.lang.System (JDK 1.0)

setError()

java.io.PrintStream (JDK 1.0), java.io.PrintWriter (JDK 1.1)

setExpert()

java.beans.FeatureDescriptor (JDK 1.1)

setExtra()

java.util.zip.ZipEntry (JDK 1.1)

setFile()

java.awt.FileDialog (JDK 1.0), java.awt.peer.FileDialogPeer (JDK 1.0)

setFilenameFilter()

java.awt.FileDialog (JDK 1.0), java.awt.peer.FileDialogPeer (JDK 1.0)

setFirstDayOfWeek()

java.util.Calendar (JDK 1.1)

setFloat()

java.lang.reflect.Array (JDK 1.1), java.lang.reflect.Field (JDK 1.1)

setFollowRedirects()

java.net.URLConnection (JDK 1.0)

setIn()

java.lang.System (JDK 1.0)

setInDefaultEventSet()

java.beans.EventSetDescriptor (JDK 1.1)

setIndex()

java.text.CharacterIterator (JDK 1.1), java.text.ParsePosition (JDK 1.1), java.text.StringCharacterIterator (JDK 1.1)

setInfinity()

java.text.DecimalFormatSymbols (JDK 1.1)

setInput()

java.util.zip.Deflater (JDK 1.1), java.util.zip.Inflater (JDK 1.1)

setInt()

java.lang.reflect.Array (JDK 1.1), java.lang.reflect.Field (JDK 1.1)

setInterface()

java.net.MulticastSocket (JDK 1.1)

setKeyChar()

java.awt.event.KeyEvent (JDK 1.1)

setKeyCode()

java.awt.event.KeyEvent (JDK 1.1)

java.awt.List (JDK 1.0), java.awt.peer.ListPeer (JDK 1.0)

setMultiplier()

java.text.DecimalFormat (JDK 1.1)

setName()

java.awt.Component (JDK 1.0), java.beans.FeatureDescriptor (JDK 1.1), java.awt.MenuComponent (JDK 1.0), java.lang.Thread (JDK 1.0)

setNaN()

java.text.DecimalFormatSymbols (JDK 1.1)

setNegativePrefix()

java.text.DecimalFormat (JDK 1.1)

setNegativeSuffix()

java.text.DecimalFormat (JDK 1.1)

setNumberFormat()

java.text.DateFormat (JDK 1.1)

setObject()

java.beans.Customizer (JDK 1.1)

setOption()

java.net.DatagramSocketImpl (JDK 1.1), java.net.SocketImpl (JDK 1.0)

setOrientation()

java.awt.Scrollbar (JDK 1.0)

`setText()`

java.text.BreakIterator (JDK 1.1), java.awt.Label (JDK 1.0), java.awt.peer.LabelPeer (JDK 1.0), java.awt.TextComponent (JDK 1.0), java.awt.peer.TextComponentPeer (JDK 1.0)

`setTime()`

java.util.Calendar (JDK 1.1), java.util.Date (JDK 1.0), java.util.zip.ZipEntry (JDK 1.1)

`setTimeInMillis()`

java.util.Calendar (JDK 1.1)

`setTimeZone()`

java.util.Calendar (JDK 1.1), java.text.DateFormat (JDK 1.1)

`setTitle()`

java.awt.Dialog (JDK 1.0), java.awt.peer.DialogPeer (JDK 1.0), java.awt.Frame (JDK 1.0), java.awt.peer.FramePeer (JDK 1.0)

`setTTL()`

java.net.DatagramSocketImpl (JDK 1.1), java.net.MulticastSocket (JDK 1.1)

`setUnicast()`

java.beans.EventSetDescriptor (JDK 1.1)

`setUnitIncrement()`

java.awt.Adjustable (JDK 1.1), java.awt.Scrollbar (JDK 1.0), java.awt.peer.ScrollPanePeer (JDK 1.1)

`setUpdateRect()`

java.awt.event.PaintEvent (JDK 1.1)

setURL()

java.net.URLStreamHandler (JDK 1.0)

setURLStreamHandlerFactory()

java.net.URL (JDK 1.0)

setUseCaches()

java.net.URLConnection (JDK 1.0)

setValue()

java.awt.Adjustable (JDK 1.1), java.beans.FeatureDescriptor (JDK 1.1), java.beans.PropertyEditor (JDK 1.1), java.beans.PropertyEditorSupport (JDK 1.1), java.awt.Scrollbar (JDK 1.0), java.awt.peer.ScrollPanePeer (JDK 1.1)

setValues()

java.awt.Scrollbar (JDK 1.0), java.awt.peer.ScrollbarPeer (JDK 1.0)

setVgap()

java.awt.BorderLayout (JDK 1.0), java.awt.CardLayout (JDK 1.0), java.awt.FlowLayout (JDK 1.0), java.awt.GridLayout (JDK 1.0)

setVisible()

java.awt.Component (JDK 1.0), java.awt.peer.ComponentPeer (JDK 1.0)

setVisibleAmount()

java.awt.Adjustable (JDK 1.1), java.awt.Scrollbar (JDK 1.0)

setWeekdays()

java.text.DateFormatSymbols (JDK 1.1)

`tertiaryOrder()`

java.text.CollationElementIterator (JDK 1.1)

`testBit()`

java.math.BigInteger (JDK 1.1)

`text`

java.awt.SystemColor (JDK 1.1)

`TEXT`

java.awt.SystemColor (JDK 1.1)

`TEXT_CURSOR`

java.awt.Cursor (JDK 1.1), java.awt.Frame (JDK 1.0)

`TEXT_EVENT_MASK`

java.awt.AWTEvent (JDK 1.1)

`TEXT_FIRST`

java.awt.event.TextEvent (JDK 1.1)

`TEXT_HIGHLIGHT`

java.awt.SystemColor (JDK 1.1)

`TEXT_HIGHLIGHT_TEXT`

java.awt.SystemColor (JDK 1.1)

`TEXT_INACTIVE_TEXT`

java.math.BigInteger (JDK 1.1), java.io.ByteArrayOutputStream (JDK 1.0), java.text.CollationKey (JDK 1.1)

toCharArray()

java.io.CharArrayWriter (JDK 1.1), java.lang.String (JDK 1.0)

toExternalForm()

java.net.URL (JDK 1.0), java.net.URLStreamHandler (JDK 1.0)

toFront()

java.awt.Window (JDK 1.0), java.awt.peer.WindowPeer (JDK 1.0)

toGMTString()

java.util.Date (JDK 1.0)

toHexString()

java.lang.Integer (JDK 1.0), java.lang.Long (JDK 1.0)

toLocaleString()

java.util.Date (JDK 1.0)

toLocalizedPattern()

java.text.DecimalFormat (JDK 1.1), java.text.SimpleDateFormat (JDK 1.1)

toLowerCase()

java.lang.Character (JDK 1.0), java.lang.String (JDK 1.0)

toOctalString()

java.lang.Integer (JDK 1.0), java.lang.Long (JDK 1.0)

TT_NUMBER

java.io.StreamTokenizer (JDK 1.0)

TT_WORD

java.io.StreamTokenizer (JDK 1.0)

ttype

java.io.StreamTokenizer (JDK 1.0)

TUESDAY

java.util.Calendar (JDK 1.1)

TYPE

java.lang.Boolean (JDK 1.0), java.lang.Byte (JDK 1.1), java.lang.Character (JDK 1.0), java.lang.Double (JDK 1.0), java.lang.Float (JDK 1.0), java.lang.Integer (JDK 1.0), java.lang.Long (JDK 1.0), java.lang.Short (JDK 1.1), java.lang.Void (JDK 1.1)

U

UK

java.util.Locale (JDK 1.1)

UNASSIGNED

java.lang.Character (JDK 1.0)

uncaughtException()

java.lang.ThreadGroup (JDK 1.0)

UNDECIMBER

java.util.Calendar (JDK 1.1)

VK_7

java.awt.event.KeyEvent (JDK 1.1)

VK_8

java.awt.event.KeyEvent (JDK 1.1)

VK_9

java.awt.event.KeyEvent (JDK 1.1)

VK_A

java.awt.event.KeyEvent (JDK 1.1)

VK_ACCEPT

java.awt.event.KeyEvent (JDK 1.1)

VK_ADD

java.awt.event.KeyEvent (JDK 1.1)

VK_ALT

java.awt.event.KeyEvent (JDK 1.1)

VK_B

java.awt.event.KeyEvent (JDK 1.1)

VK_BACK_QUOTE

java.awt.event.KeyEvent (JDK 1.1)

VK_BACK_SLASH

java.awt.event.KeyEvent (JDK 1.1)

`VK_BACK_SPACE`

java.awt.event.KeyEvent (JDK 1.1)

`VK_C`

java.awt.event.KeyEvent (JDK 1.1)

`VK_CANCEL`

java.awt.event.KeyEvent (JDK 1.1)

`VK_CAPS_LOCK`

java.awt.event.KeyEvent (JDK 1.1)

`VK_CLEAR`

java.awt.event.KeyEvent (JDK 1.1)

`VK_CLOSE_BRACKET`

java.awt.event.KeyEvent (JDK 1.1)

`VK_COMMA`

java.awt.event.KeyEvent (JDK 1.1)

`VK_CONTROL`

java.awt.event.KeyEvent (JDK 1.1)

`VK_CONVERT`

java.awt.event.KeyEvent (JDK 1.1)

`VK_D`

java.awt.event.KeyEvent (JDK 1.1)

VK_DECIMAL

java.awt.event.KeyEvent (JDK 1.1)

VK_DELETE

java.awt.event.KeyEvent (JDK 1.1)

VK_DIVIDE

java.awt.event.KeyEvent (JDK 1.1)

VK_DOWN

java.awt.event.KeyEvent (JDK 1.1)

VK_E

java.awt.event.KeyEvent (JDK 1.1)

VK_END

java.awt.event.KeyEvent (JDK 1.1)

VK_ENTER

java.awt.event.KeyEvent (JDK 1.1)

VK_EQUALS

java.awt.event.KeyEvent (JDK 1.1)

VK_ESCAPE

java.awt.event.KeyEvent (JDK 1.1)

VK_F

java.awt.event.KeyEvent (JDK 1.1)

VK_F1

java.awt.event.KeyEvent (JDK 1.1)

VK_F10

java.awt.event.KeyEvent (JDK 1.1)

VK_F11

java.awt.event.KeyEvent (JDK 1.1)

VK_F12

java.awt.event.KeyEvent (JDK 1.1)

VK_F2

java.awt.event.KeyEvent (JDK 1.1)

VK_F3

java.awt.event.KeyEvent (JDK 1.1)

VK_F4

java.awt.event.KeyEvent (JDK 1.1)

VK_F5

java.awt.event.KeyEvent (JDK 1.1)

VK_F6

java.awt.event.KeyEvent (JDK 1.1)

VK_F7

java.awt.event.KeyEvent (JDK 1.1)

`VK_F8`

java.awt.event.KeyEvent (JDK 1.1)

`VK_F9`

java.awt.event.KeyEvent (JDK 1.1)

`VK_FINAL`

java.awt.event.KeyEvent (JDK 1.1)

`VK_G`

java.awt.event.KeyEvent (JDK 1.1)

`VK_H`

java.awt.event.KeyEvent (JDK 1.1)

`VK_HELP`

java.awt.event.KeyEvent (JDK 1.1)

`VK_HOME`

java.awt.event.KeyEvent (JDK 1.1)

`VK_I`

java.awt.event.KeyEvent (JDK 1.1)

`VK_INSERT`

java.awt.event.KeyEvent (JDK 1.1)

VK_J

java.awt.event.KeyEvent (JDK 1.1)

VK_K

java.awt.event.KeyEvent (JDK 1.1)

VK_KANA

java.awt.event.KeyEvent (JDK 1.1)

VK_KANJI

java.awt.event.KeyEvent (JDK 1.1)

VK_L

java.awt.event.KeyEvent (JDK 1.1)

VK_LEFT

java.awt.event.KeyEvent (JDK 1.1)

VK_M

java.awt.event.KeyEvent (JDK 1.1)

VK_META

java.awt.event.KeyEvent (JDK 1.1)

VK_MODECHANGE

java.awt.event.KeyEvent (JDK 1.1)

VK_MULTIPLY

java.awt.event.KeyEvent (JDK 1.1)

VK_N

java.awt.event.KeyEvent (JDK 1.1)

VK_NONCONVERT

java.awt.event.KeyEvent (JDK 1.1)

VK_NUM_LOCK

java.awt.event.KeyEvent (JDK 1.1)

VK_NUMPAD0

java.awt.event.KeyEvent (JDK 1.1)

VK_NUMPAD1

java.awt.event.KeyEvent (JDK 1.1)

VK_NUMPAD2

java.awt.event.KeyEvent (JDK 1.1)

VK_NUMPAD3

java.awt.event.KeyEvent (JDK 1.1)

VK_NUMPAD4

java.awt.event.KeyEvent (JDK 1.1)

VK_NUMPAD5

java.awt.event.KeyEvent (JDK 1.1)

VK_NUMPAD6

java.awt.event.KeyEvent (JDK 1.1)

VK_NUMPAD7

java.awt.event.KeyEvent (JDK 1.1)

VK_NUMPAD8

java.awt.event.KeyEvent (JDK 1.1)

VK_NUMPAD9

java.awt.event.KeyEvent (JDK 1.1)

VK_O

java.awt.event.KeyEvent (JDK 1.1)

VK_OPEN_BRACKET

java.awt.event.KeyEvent (JDK 1.1)

VK_P

java.awt.event.KeyEvent (JDK 1.1)

VK_PAGE_DOWN

java.awt.event.KeyEvent (JDK 1.1)

VK_PAGE_UP

java.awt.event.KeyEvent (JDK 1.1)

VK_PAUSE

java.awt.event.KeyEvent (JDK 1.1)

VK_PERIOD

java.awt.event.KeyEvent (JDK 1.1)

`VK_PRINTSCREEN`

java.awt.event.KeyEvent (JDK 1.1)

`VK_Q`

java.awt.event.KeyEvent (JDK 1.1)

`VK_QUOTE`

java.awt.event.KeyEvent (JDK 1.1)

`VK_R`

java.awt.event.KeyEvent (JDK 1.1)

`VK_RIGHT`

java.awt.event.KeyEvent (JDK 1.1)

`VK_S`

java.awt.event.KeyEvent (JDK 1.1)

`VK_SCROLL_LOCK`

java.awt.event.KeyEvent (JDK 1.1)

`VK_SEMICOLON`

java.awt.event.KeyEvent (JDK 1.1)

`VK_SEPARATER`

java.awt.event.KeyEvent (JDK 1.1)

`VK_SHIFT`

java.awt.event.KeyEvent (JDK 1.1)

VK_SLASH

java.awt.event.KeyEvent (JDK 1.1)

VK_SPACE

java.awt.event.KeyEvent (JDK 1.1)

VK_SUBTRACT

java.awt.event.KeyEvent (JDK 1.1)

VK_T

java.awt.event.KeyEvent (JDK 1.1)

VK_TAB

java.awt.event.KeyEvent (JDK 1.1)

VK_U

java.awt.event.KeyEvent (JDK 1.1)

VK_UNDEFINED

java.awt.event.KeyEvent (JDK 1.1)

VK_UP

java.awt.event.KeyEvent (JDK 1.1)

VK_V

java.awt.event.KeyEvent (JDK 1.1)

VK_W

java.awt.SystemColor (JDK 1.1)

WINDOW_CLOSED

java.awt.event.WindowEvent (JDK 1.1)

WINDOW_CLOSING

java.awt.event.WindowEvent (JDK 1.1)

WINDOW_DEACTIVATED

java.awt.event.WindowEvent (JDK 1.1)

WINDOW_DEICONIFIED

java.awt.event.WindowEvent (JDK 1.1)

WINDOW_DEICONIFY

java.awt.Event (JDK 1.0)

WINDOW_DESTROY

java.awt.Event (JDK 1.0)

WINDOW_EVENT_MASK

java.awt.AWTEvent (JDK 1.1)

WINDOW_EXPOSE

java.awt.Event (JDK 1.0)

WINDOW_FIRST

java.awt.event.WindowEvent (JDK 1.1)

WINDOW_ICONIFIED

java.awt.event.WindowEvent (JDK 1.1)

`WINDOW_ICONIFY`

java.awt.Event (JDK 1.0)

`WINDOW_LAST`

java.awt.event.WindowEvent (JDK 1.1)

`WINDOW_MOVED`

java.awt.Event (JDK 1.0)

`WINDOW_OPENED`

java.awt.event.WindowEvent (JDK 1.1)

`WINDOW_TEXT`

java.awt.SystemColor (JDK 1.1)

`windowActivated()`

java.awt.AWTEventMulticaster (JDK 1.1), java.awt.event.WindowAdapter (JDK 1.1), java.awt.event.WindowListener (JDK 1.1)

`WindowAdapter`

The java.awt.event Package

`windowBorder`

java.awt.SystemColor (JDK 1.1)

`windowClosed()`

java.awt.AWTEventMulticaster (JDK 1.1), java.awt.event.WindowAdapter (JDK 1.1), java.awt.event.WindowListener (JDK 1.1)

`writeChar()`

java.io.DataOutput (JDK 1.0), java.io.DataOutputStream (JDK 1.0), java.io.ObjectOutputStream (JDK 1.1), java.io.RandomAccessFile (JDK 1.0)

`writeChars()`

java.io.DataOutput (JDK 1.0), java.io.DataOutputStream (JDK 1.0), java.io.ObjectOutputStream (JDK 1.1), java.io.RandomAccessFile (JDK 1.0)

`writeDouble()`

java.io.DataOutput (JDK 1.0), java.io.DataOutputStream (JDK 1.0), java.io.ObjectOutputStream (JDK 1.1), java.io.RandomAccessFile (JDK 1.0)

`writeExternal()`

java.io.Externalizable (JDK 1.1)

`writeFloat()`

java.io.DataOutput (JDK 1.0), java.io.DataOutputStream (JDK 1.0), java.io.ObjectOutputStream (JDK 1.1), java.io.RandomAccessFile (JDK 1.0)

`writeInt()`

java.io.DataOutput (JDK 1.0), java.io.DataOutputStream (JDK 1.0), java.io.ObjectOutputStream (JDK 1.1), java.io.RandomAccessFile (JDK 1.0)

`writeLong()`

java.io.DataOutput (JDK 1.0), java.io.DataOutputStream (JDK 1.0), java.io.ObjectOutputStream (JDK 1.1), java.io.RandomAccessFile (JDK 1.0)

`writeObject()`

java.io.ObjectOutput (JDK 1.1), java.io.ObjectOutputStream (JDK 1.1)

`Writer`

PREVIOUS HOME

java.util.zip.ZipOutputStream (JDK 1.1) BOOK INDEX

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

31.17 java.util.zip.ZipOutputStream (JDK 1.1)

This class is a subclass of `DeflaterOutputStream` that writes data in ZIP file format to an output stream. Before writing any data to the `ZipOutputStream`, you must begin an entry within the ZIP file with `putNextEntry()`. The `ZipEntry` object passed to this method should specify at least a name for the entry. Once you have begun an entry with `putNextEntry()`, you can write the contents of that entry with the `write()` methods. When you reach the end of an entry, you can begin a new one by calling `putNextEntry()` again, or you can close the current entry with `closeEntry()`, or you can close the stream itself with `close()`.

Before beginning an entry with `putNextEntry()`, you can set the compression method and level with `setMethod()` and `setLevel()`. The constants `DEFLATED` and `STORED` are the two legal values for `setMethod()`. If you use `STORED`, the entry is stored in the ZIP file without any compression. If you use `DEFLATED`, you can also specify the compression speed/strength trade-off by passing a number from 1 to 9 to `setLevel()`, where 9 gives the strongest and slowest level of compression. You can also use the constants `Deflater.BEST_SPEED`, `Deflater.BEST_COMPRESSION`, and `Deflater.DEFAULT_COMPRESSION` with the `setLevel()` method.

Note that if you are simply storing an entry without compression, the ZIP file format requires that you specify, in advance, the entry size and CRC-32 checksum in the `ZipEntry` object for the entry. An exception is thrown if these values are not specified or are incorrectly specified.

```
public class ZipOutputStream extends DeflaterOutputStream {
    // Public Constructor
            public ZipOutputStream(OutputStream out);
    // Constants
            public static final int DEFLATED;
            public static final int STORED;
    // Public Instance Methods
            public void close() throws IOException;  // Overrides 
DeflaterOutputStream
            public void closeEntry() throws IOException;
            public void finish() throws IOException;  // Overrides 
DeflaterOutputStream
            public void putNextEntry(ZipEntry e) throws IOException;
            public void setComment(String comment);
            public void setLevel(int level);
            public void setMethod(int method);
            public synchronized void write(byte[] b, int off, int len) throws
IOException;  // Overrides DeflaterOutputStream
}
```

Hierarchy:

Object->OutputStream->FilterOutputStream->DeflaterOutputStream->ZipOutputStream

PREVIOUS | HOME | NEXT

java.util.zip.ZipInputStream (JDK 1.1) | BOOK INDEX | Class, Method, and Field Index

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

18.3 java.awt.AWTEventMulticaster (JDK 1.1)

`AWTEventMulticaster` is a convenience class used when writing a custom AWT component. It provides an easy way to maintain a list of AWT `EventListener` objects, and to notify the listeners on that list when an event occurs.

`AWTEventMulticaster` implements each of the event listener interfaces defined in the `java.awt.event` package, which means that an `AWTEventMulticaster` object can serve as any desired type of event listener. (It also means, as you can see below, that the class defines quite a few methods.) `AWTEventMulticaster` implements what amounts to a linked list of `EventListener` objects. When you invoke one of the `EventListener` methods of an `AWTEventMulticaster`, it invokes the same method on all of the `EventListener` objects in the linked list.

Rather than instantiate an `AWTEventMulticaster` object directly, you use the static `add()` and `remove()` methods of the class to add and remove `EventListener` objects from the linked list. Doing so returns an `AWTEventMulticaster` with the appropriate `EventListener` object registered. The API for using an `AWTEventMulticaster` is somewhat non-intuitive. Here is some example code that shows its use:

```
public class MyList extends Component {   // a class that sends ItemEvents
  // this will be the head of a linked list of AWTEventMulticaster objects
  protected ItemListener listener = null;
  public void addItemListener(ItemListener l) {      // add a listener
    listener = AWTEventMulticaster.add(listener, l);
  }
  public void removeItemListener(ItemListener l) {   // remove a listener
    listener = AWTEventMulticaster.remove(listener, l);
  }
  protected void fireItemEvent(ItemEvent e) {        // notify all listeners
    if (listener != null) listener.itemStateChanged(e);
  }
  // The rest of the class goes here
}
```

```
public class AWTEvent Multicaster  extends Object
                                   implements ComponentListener, ContainerListener,
FocusListener, KeyListener, MouseListener, MouseMotionListener, WindowListener,
ActionListener,ItemListener, AdjustmentListener, TextListener {
    // Protected Constructor
            protected AWTEventMulticaster(EventListener a, EventListener b);
    // Protected Instance Variables
            protected EventListener a;
            protected EventListener b;
    // Class Methods
```

```
            public static ComponentListener add(ComponentListener a,
ComponentListener b);
            public static ContainerListener add(ContainerListener a,
ContainerListener b);
            public static FocusListener add(FocusListener a, FocusListener b);
            public static KeyListener add(KeyListener a, KeyListener b);
            public static MouseListener add(MouseListener a, MouseListener b);
            public static MouseMotionListener add(MouseMotionListener a,
MouseMotionListener b);
            public static WindowListener add(WindowListener a, WindowListener b);
            public static ActionListener add(ActionListener a, ActionListener b);
            public static ItemListener add(ItemListener a, ItemListener b);
            public static AdjustmentListener add(AdjustmentListener a,
AdjustmentListener b);
            public static TextListener add(TextListener a, TextListener b);
            protected static EventListener addInternal(EventListener a, EventListener
b);
            public static ComponentListener remove(ComponentListener l,
ComponentListener oldl);
            public static ContainerListener remove(ContainerListener l,
ContainerListener oldl);
            public static FocusListener remove(FocusListener l, FocusListener oldl);
            public static KeyListener remove(KeyListener l, KeyListener oldl);
            public static MouseListener remove(MouseListener l, MouseListener oldl);
            public static MouseMotionListener remove(MouseMotionListener l,
MouseMotionListener oldl);
            public static WindowListener remove(WindowListener l, WindowListener
oldl);
            public static ActionListener remove(ActionListener l, ActionListener
oldl);
            public static ItemListener remove(ItemListener l, ItemListener oldl);
            public static AdjustmentListener remove(AdjustmentListener l,
AdjustmentListener oldl);
            public static TextListener remove(TextListener l, TextListener oldl);
            protected static EventListener removeInternal(EventListener l,
EventListener oldl);
            protected static void save(ObjectOutputStream s, String k, EventListener
l) throws IOException;
    // Public Instance Methods
            public void actionPerformed(ActionEvent e);  // From ActionListener
            public void adjustmentValueChanged(AdjustmentEvent e);  // From
AdjustmentListener
            public void componentAdded(ContainerEvent e);  // From ContainerListener
            public void componentHidden(ComponentEvent e);  // From ComponentListener
            public void componentMoved(ComponentEvent e);  // From ComponentListener
            public void componentRemoved(ContainerEvent e);  // From
ContainerListener
            public void componentResized(ComponentEvent e);  // From
ComponentListener
            public void componentShown(ComponentEvent e);  // From ComponentListener
            public void focusGained(FocusEvent e);  // From FocusListener
            public void focusLost(FocusEvent e);  // From FocusListener
            public void itemStateChanged(ItemEvent e);  // From ItemListener
            public void keyPressed(KeyEvent e);  // From KeyListener
            public void keyReleased(KeyEvent e);  // From KeyListener
            public void keyTyped(KeyEvent e);  // From KeyListener
            public void mouseClicked(MouseEvent e);  // From MouseListener
            public void mouseDragged(MouseEvent e);  // From MouseMotionListener
            public void mouseEntered(MouseEvent e);  // From MouseListener
            public void mouseExited(MouseEvent e);  // From MouseListener
            public void mouseMoved(MouseEvent e);  // From MouseMotionListener
            public void mousePressed(MouseEvent e);  // From MouseListener
            public void mouseReleased(MouseEvent e);  // From MouseListener
            public void textValueChanged(TextEvent e);  // From TextListener
            public void windowActivated(WindowEvent e);  // From WindowListener
            public void windowClosed(WindowEvent e);  // From WindowListener
            public void windowClosing(WindowEvent e);  // From WindowListener
            public void windowDeactivated(WindowEvent e);  // From WindowListener
            public void windowDeiconified(WindowEvent e);  // From WindowListener
            public void windowIconified(WindowEvent e);  // From WindowListener
            public void windowOpened(WindowEvent e);  // From WindowListener
    // Protected Instance Methods
            protected EventListener remove(EventListener oldl);
            protected void saveInternal(ObjectOutputStream s, String k) throws
IOException;
}
```

Hierarchy:

```
Object->AWTEventMulticaster(ComponentListener(EventListener),
ContainerListener(EventListener), FocusListener(EventListener),
KeyListener(EventListener), MouseListener(EventListener),
MouseMotionListener(EventListener), WindowListener(EventListener),
ActionListener(EventListener), ItemListener(EventListener),
AdjustmentListener(EventListener), TextListener(EventListener))
```

21.8 java.awt.image.ImageObserver (JDK 1.0)

This interface defines a method and associated constants used by classes that want to receive information asynchronously about the status of an image. Many methods that query information about an image take an `ImageObserver` as an argument. If the specified information is not available when requested, it is passed to the `ImageObserver` when it becomes available. `Component` implements this interface, and components are the most commonly used image observers.

```
public abstract interface ImageObserver {
    // Constants
            public static final int ABORT;
            public static final int ALLBITS;
            public static final int ERROR;
            public static final int FRAMEBITS;
            public static final int HEIGHT;
            public static final int PROPERTIES;
            public static final int SOMEBITS;
            public static final int WIDTH;
    // Public Instance Methods
            public abstract boolean imageUpdate(Image img, int infoflags, int x, int
y, int width, int height);
}
```

Implemented By:

`Component`

Passed To:

`Component.checkImage(), Component.prepareImage(), ComponentPeer.checkImage(), ComponentPeer.prepareImage(), Graphics.drawImage(), Image.getHeight(), Image.getProperty(), Image.getWidth(), Toolkit.checkImage(), Toolkit.prepareImage()`

18.39 java.awt.MediaTracker (JDK 1.0)

This class provides a very convenient way to asynchronously load and keep track of the status of any number of `Image` objects. You can use it to load one or more images and to wait until those images have been completely loaded and are ready to be used.

The `addImage()` method registers an image to be loaded and tracked and assigns it a specified identifier value. `waitForID()` loads all images that have been assigned the specified identifier and returns when they have all finished loading or it has received an error. `isErrorAny()` and `isErrorID()` check whether any errors have occurred while loading images. `statusAll()` and `statusID()` return the status of all images or of all images with the specified identifier. The return value of these two methods is one of the defined constants.

```
public class MediaTracker extends Object implements Serializable {
    // Public Constructor
            public MediaTracker(Component comp);
    // Constants
            public static final int ABORTED;
            public static final int COMPLETE;
            public static final int ERRORED;
            public static final int LOADING;
    // Public Instance Methods
            public void addImage(Image image, int id);
            public synchronized void addImage(Image image, int id, int w, int h);
            public boolean checkAll();
            public boolean checkAll(boolean load);
            public boolean checkID(int id);
            public boolean checkID(int id, boolean load);
            public synchronized Object[] getErrorsAny();
            public synchronized Object[] getErrorsID(int id);
            public synchronized boolean isErrorAny();
            public synchronized boolean isErrorID(int id);
        1.1  public synchronized void removeImage(Image image);
        1.1  public synchronized void removeImage(Image image, int id);
        1.1  public synchronized void removeImage(Image image, int id, int width, int
height);
            public int statusAll(boolean load);
            public int statusID(int id, boolean load);
            public void waitForAll() throws InterruptedException;
            public synchronized boolean waitForAll(long ms) throws
InterruptedException;
            public void waitForID(int id) throws InterruptedException;
            public synchronized boolean waitForID(int id, long ms) throws
InterruptedException;
}
```

PREVIOUS | HOME | NEXT

java.awt.List (JDK 1.0) | BOOK INDEX | java.awt.Menu (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

PREVIOUS

NEXT

21.12 java.awt.image.PixelGrabber (JDK 1.0)

This class is an `ImageConsumer` that extracts a specified rectangular array of pixels (in the default RGB color model) from a specified `Image` or `ImageProducer` and stores them into a specified array (using the specified offset into the array and specified scanline size). Use this class when you want to inspect or manipulate the data of an image or some rectangular portion of an image.

The method `grabPixels()` makes the `PixelGrabber` start grabbing pixels. `status()` returns the status of the pixel-grabbing process. The return value uses the same flag value constants as the `ImageObserver` class does. The remaining methods are the standard `ImageConsumer` methods and should not be called directly.

```
public class PixelGrabber extends Object implements ImageConsumer {
    // Public Constructors
            public PixelGrabber(Image img, int x, int y, int w, int h, int[] pix, int 
off, int scansize);
            public PixelGrabber(ImageProducer ip, int x, int y, int w, int h, int[] 
pix, int off, int scansize);
        1.1  public PixelGrabber(Image img, int x, int y, int w, int h, boolean 
forceRGB);
    // Public Instance Methods
        1.1  public synchronized void abortGrabbing();
        1.1  public synchronized ColorModel getColorModel();
        1.1  public synchronized int getHeight();
        1.1  public synchronized Object getPixels();
        1.1  public synchronized int getStatus();
        1.1  public synchronized int getWidth();
            public boolean grabPixels() throws InterruptedException;
            public synchronized boolean grabPixels(long ms) throws 
InterruptedException;
            public synchronized void imageComplete(int status);  // From 
ImageConsumer
            public void setColorModel(ColorModel model);  // From ImageConsumer
            public void setDimensions(int width, int height);  // From ImageConsumer
            public void setHints(int hints);  // From ImageConsumer
            public void setPixels(int srcX, int srcY, int srcW, int srcH, ColorModel 
model, 
            public void setPixels'u'byte[] pixels, int srcOff, int srcScan);  // From 
ImageConsumer
            public void setPixels(int srcX, int srcY, int srcW, int srcH, ColorModel 
model, 
            public void setPixels'u'int[] pixels, int srcOff, int srcScan);  // From 
ImageConsumer
            public void setProperties(Hashtable props);  // From ImageConsumer
        1.1  public synchronized void startGrabbing();
            public synchronized int status();
}
```

PREVIOUS | HOME | NEXT
java.awt.image.MemoryImageSource (JDK 1.0) | BOOK INDEX | java.awt.image.RGBImageFilter (JDK 1.0)

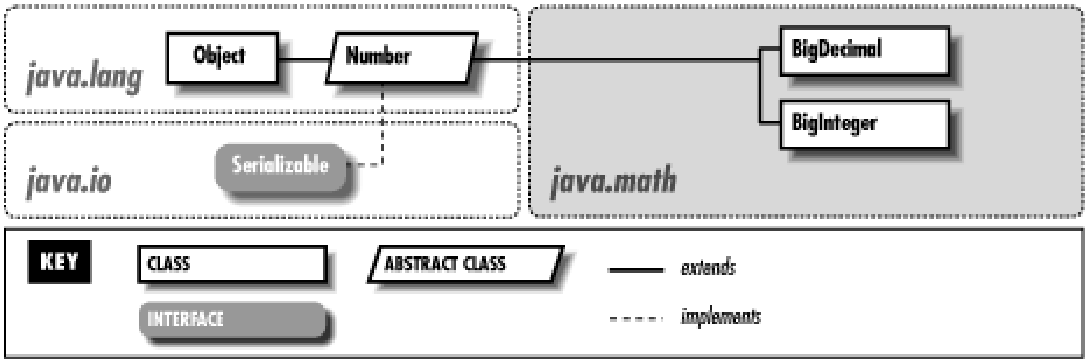
JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

27. The java.math Package

Contents:

The `java.math` package, new in Java 1.1, contains classes for arbitrary-precision integer and floating-point arithmetic. Arbitrary-length integers are required for cryptography, and arbitrary-precision floating-point values are useful for financial applications that need to be careful about rounding errors. The class hierarchy of this extremely small package is shown in Figure 27.1.

Figure 27.1: The java.math package



27.1 java.math.BigDecimal (JDK 1.1)

This subclass of `java.lang.Number` represents a floating-point number of arbitrary size and precision. Its methods duplicate the functionality of the standard Java arithmetic operators. The `compareTo()` method compares the value of two `BigDecimal` objects and returns -1, 0, or 1 to indicate the result of the comparison.

`BigDecimal` objects are represented as an integer of arbitrary size and an integer scale that specifies the number of decimal places in the value. When working with `BigDecimal` values, you can explicitly specify the amount of precision (the number of decimal places) you are interested in. Also, whenever a `BigDecimal` method may discard precision (in a division operation, for example), you are required to specify what sort of rounding should be performed on the digit to the left of the discarded digit or digits. The eight constants defined by this class specify the available rounding modes. Because the `BigDecimal` class provides arbitrary precision and gives you explicit control over rounding and the number of decimal places you are interested in, it can be useful when dealing with quantities that represent money, or in other circumstances where the tolerance for rounding errors is low.

```
public class BigDecimal extends Number {
    // Public Constructors
            public BigDecimal(String val) throws NumberFormatException;
            public BigDecimal(double val) throws NumberFormatException;
            public BigDecimal(BigInteger val);
            public BigDecimal(BigInteger val, int scale) throws
NumberFormatException;
    // Constants
            public static final int ROUND_CEILING;
            public static final int ROUND_DOWN;
            public static final int ROUND_FLOOR;
            public static final int ROUND_HALF_DOWN;
            public static final int ROUND_HALF_EVEN;
            public static final int ROUND_HALF_UP;
            public static final int ROUND_UNNECESSARY;
            public static final int ROUND_UP;
    // Class Methods
            public static BigDecimal valueOf(long val, int scale) throws
NumberFormatException;
            public static BigDecimal valueOf(long val);
    // Public Instance Methods
            public BigDecimal abs();
            public BigDecimal add(BigDecimal val);
            public int compareTo(BigDecimal val);
            public BigDecimal divide(BigDecimal val, int scale, int roundingMode)
throws ArithmeticException, IllegalArgumentException;
            public BigDecimal divide(BigDecimal val, int roundingMode) throws
ArithmeticException, IllegalArgumentException;
            public double doubleValue();  // Defines Number
            public boolean equals(Object x);  // Overrides Object
            public float floatValue();  // Defines Number
            public int hashCode();  // Overrides Object
            public int intValue();  // Defines Number
            public long longValue();  // Defines Number
            public BigDecimal max(BigDecimal val);
            public BigDecimal min(BigDecimal val);
            public BigDecimal movePointLeft(int n);
            public BigDecimal movePointRight(int n);
            public BigDecimal multiply(BigDecimal val);
            public BigDecimal negate();
            public int scale();
            public BigDecimal setScale(int scale, int roundingMode) throws
ArithmeticException, IllegalArgumentException;
            public BigDecimal setScale(int scale) throws ArithmeticException,
IllegalArgumentException;
            public int signum();
            public BigDecimal subtract(BigDecimal val);
            public BigInteger toBigInteger();
            public String toString();  // Overrides Object
}
```

Hierarchy:

Object->Number(Serializable)->BigDecimal

27.2 java.math.BigInteger (JDK 1.1)

This subclass of `java.lang.Number` represents integers that can be arbitrarily large--i.e., integers that are not limited to the 64 bits available with the `long` data type. `BigInteger` defines methods that duplicate the functionality of the standard Java arithmetic and bit-manipulation operators. The `compareTo()` method compares two `BigInteger` objects, and returns -1, 0, or 1 to indicate the result of the comparison.

The `gcd()`, `modPow()`, `modInverse()`, and `isProbablePrime()` methods perform advanced operations and are used primarily in cryptography and related algorithms.

```
public class BigInteger extends Number {
    // Public Constructors
            public BigInteger(byte[] val) throws NumberFormatException;
            public BigInteger(int signum, byte[] magnitude) throws 
NumberFormatException;
            public BigInteger(String val, int radix) throws NumberFormatException;
            public BigInteger(String val) throws NumberFormatException;
            public BigInteger(int numBits, Random rndSrc) throws 
IllegalArgumentException;
            public BigInteger(int bitLength, int certainty, Random rnd);
    // Class Methods
            public static BigInteger valueOf(long val);
    // Public Instance Methods
            public BigInteger abs();
            public BigInteger add(BigInteger val) throws ArithmeticException;
            public BigInteger and(BigInteger val);
            public BigInteger andNot(BigInteger val);
            public int bitCount();
            public int bitLength();
            public BigInteger clearBit(int n) throws ArithmeticException;
            public int compareTo(BigInteger val);
            public BigInteger divide(BigInteger val) throws ArithmeticException;
            public BigInteger[] divideAndRemainder(BigInteger val) throws 
ArithmeticException;
            public double doubleValue();  // Defines Number
            public boolean equals(Object x);  // Overrides Object
            public BigInteger flipBit(int n) throws ArithmeticException;
            public float floatValue();  // Defines Number
            public BigInteger gcd(BigInteger val);
            public int getLowestSetBit();
            public int hashCode();  // Overrides Object
            public int intValue();  // Defines Number
            public boolean isProbablePrime(int certainty);
            public long longValue();  // Defines Number
            public BigInteger max(BigInteger val);
            public BigInteger min(BigInteger val);
            public BigInteger mod(BigInteger m);
            public BigInteger modInverse(BigInteger m) throws ArithmeticException;
            public BigInteger modPow(BigInteger exponent, BigInteger m);
            public BigInteger multiply(BigInteger val);
            public BigInteger negate();
            public BigInteger not();
            public BigInteger or(BigInteger val);
            public BigInteger pow(int exponent) throws ArithmeticException;
            public BigInteger remainder(BigInteger val) throws ArithmeticException;
            public BigInteger setBit(int n) throws ArithmeticException;
            public BigInteger shiftLeft(int n);
            public BigInteger shiftRight(int n);
            public int signum();
            public BigInteger subtract(BigInteger val);
            public boolean testBit(int n) throws ArithmeticException;
            public byte[] toByteArray();
            public String toString(int radix);
            public String toString();  // Overrides Object
            public BigInteger xor(BigInteger val);
}
```

Hierarchy:

Object->Number(Serializable)->BigInteger

Passed To:

BigDecimal()

Returned By:

BigDecimal.toBigInteger()

PREVIOUS HOME NEXT
java.math.BigDecimal (JDK 1.1) BOOK INDEX The java.net Package

25.37 java.lang.Math (JDK 1.0)

This class defines constants for the mathematical values *e* and pi, and defines static methods for floating-point trigonometry, exponentiation, and other operations. It is the equivalent of the C <*math.h*> functions. It also contains methods for computing minimum and maximum values and for generating pseudo-random numbers.

```
public final class Math extends Object {
    // No Constructor
    // Constants
            public static final double E;
            public static final double PI;
    // Class Methods
            public static native double IEEEremainder(double f1, double f2);
            public static int abs(int a);
            public static long abs(long a);
            public static float abs(float a);
            public static double abs(double a);
            public static native double acos(double a);
            public static native double asin(double a);
            public static native double atan(double a);
            public static native double atan2(double a, double b);
            public static native double ceil(double a);
            public static native double cos(double a);
            public static native double exp(double a);
            public static native double floor(double a);
            public static native double log(double a);
            public static int max(int a, int b);
            public static long max(long a, long b);
            public static float max(float a, float b);
            public static double max(double a, double b);
            public static int min(int a, int b);
            public static long min(long a, long b);
            public static float min(float a, float b);
            public static double min(double a, double b);
            public static native double pow(double a, double b);
            public static synchronized double random();
            public static native double rint(double a);
            public static int round(float a);
            public static long round(double a);
            public static native double sin(double a);
            public static native double sqrt(double a);
            public static native double tan(double a);
}
```

PREVIOUS | HOME | NEXT
java.lang.Long (JDK 1.0) | BOOK INDEX | java.lang.NegativeArraySizeException (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

26.7 java.lang.reflect.Modifier (JDK 1.1)

This class defines a number of constants and static methods that are used to interpret the integer values returned by the `getModifiers()` methods of the `Field`, `Method`, and `Constructor` classes. The `isPublic()`, `isAbstract()`, and related methods return `true` if the modifiers value includes the specified modifier, otherwise they return `false`.

The constants defined by this class specify the various bit flags used in the modifiers value. You can use these constants to test for modifiers if you want to perform your own boolean algebra.

```
public class Modifier extends Object {
    // Default Constructor: public Modifier()
    // Constants
            public static final int ABSTRACT;
            public static final int FINAL;
            public static final int INTERFACE;
            public static final int NATIVE;
            public static final int PRIVATE;
            public static final int PROTECTED;
            public static final int PUBLIC;
            public static final int STATIC;
            public static final int SYNCHRONIZED;
            public static final int TRANSIENT;
            public static final int VOLATILE;
    // Class Methods
            public static boolean isAbstract(int mod);
            public static boolean isFinal(int mod);
            public static boolean isInterface(int mod);
            public static boolean isNative(int mod);
            public static boolean isPrivate(int mod);
            public static boolean isProtected(int mod);
            public static boolean isPublic(int mod);
            public static boolean isStatic(int mod);
            public static boolean isSynchronized(int mod);
            public static boolean isTransient(int mod);
            public static boolean isVolatile(int mod);
            public static String toString(int mod);
}
```

PREVIOUS | HOME | NEXT
java.lang.reflect.Method (JDK 1.1) | BOOK INDEX | The java.math Package

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

25. The java.lang Package

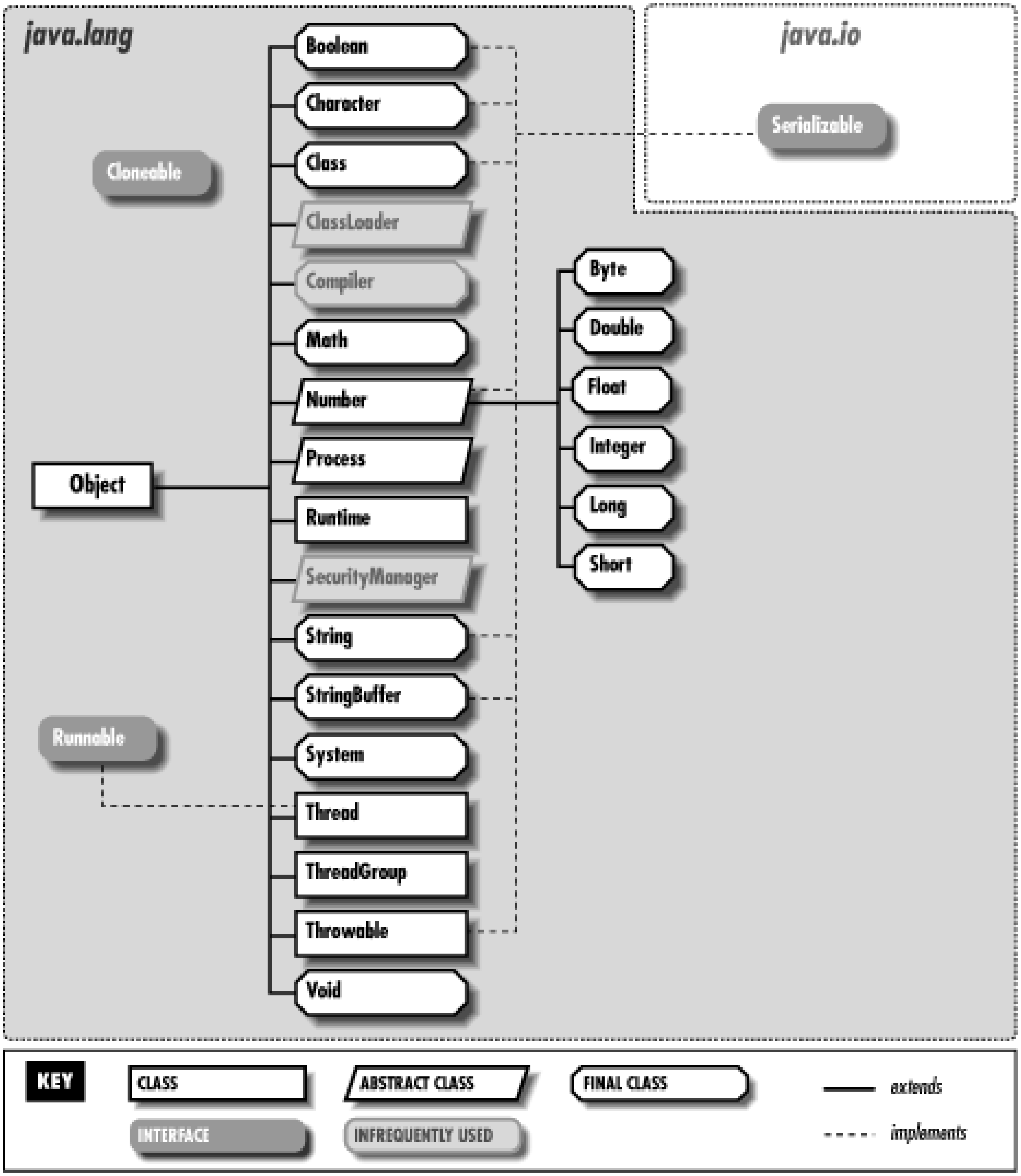
Contents:

java.lang.IncompatibleClassChangeError (JDK 1.0)
java.lang.IndexOutOfBoundsException (JDK 1.0)
java.lang.InstantiationError (JDK 1.0)
java.lang.InstantiationException (JDK 1.0)
java.lang.Integer (JDK 1.0)
java.lang.InternalError (JDK 1.0)
java.lang.InterruptedException (JDK 1.0)
java.lang.LinkageError (JDK 1.0)
java.lang.Long (JDK 1.0)
java.lang.Math (JDK 1.0)
java.lang.NegativeArraySizeException (JDK 1.0)
java.lang.NoClassDefFoundError (JDK 1.0)
java.lang.NoSuchFieldError (JDK 1.0)
java.lang.NoSuchFieldException (JDK 1.1)
java.lang.NoSuchMethodError (JDK 1.0)
java.lang.NoSuchMethodException (JDK 1.0)
java.lang.NullPointerException (JDK 1.0)
java.lang.Number (JDK 1.0)
java.lang.NumberFormatException (JDK 1.0)
java.lang.Object (JDK 1.0)
java.lang.OutOfMemoryError (JDK 1.0)
java.lang.Process (JDK 1.0)
java.lang.Runnable (JDK 1.0)
java.lang.Runtime (JDK 1.0)
java.lang.RuntimeException (JDK 1.0)
java.lang.SecurityException (JDK 1.0)
java.lang.SecurityManager (JDK 1.0)
java.lang.Short (JDK 1.1)
java.lang.StackOverflowError (JDK 1.0)
java.lang.String (JDK 1.0)
java.lang.StringBuffer (JDK 1.0)
java.lang.StringIndexOutOfBoundsException (JDK 1.0)
java.lang.System (JDK 1.0)
java.lang.Thread (JDK 1.0)
java.lang.ThreadDeath (JDK 1.0)
java.lang.ThreadGroup (JDK 1.0)
java.lang.Throwable (JDK 1.0)
java.lang.UnknownError (JDK 1.0)
java.lang.UnsatisfiedLinkError (JDK 1.0)

java.lang.VerifyError (JDK 1.0)
java.lang.VirtualMachineError (JDK 1.0)
java.lang.Void (JDK 1.1)

The `java.lang` package contains the classes that are most central to the Java language. As you can see from Figure 25.1, the class hierarchy is broad rather than deep, which means that the classes are independent of each other.

Figure 25.1: The java.lang package



`Object` is the ultimate superclass of all Java classes and is therefore at the top of all class hierarchies. `Class` is a class that describes a Java class. There is one `Class` object for each class that is loaded into Java.

`Boolean`, `Character`, `Byte`, `Short`, `Integer`, `Long`, `Float`, and `Double` are immutable class wrappers around each of the primitive Java data types. These classes are useful when you need to manipulate primitive types as objects. They also contain useful conversion and utility methods.

`String` and `StringBuffer` are objects that represent strings. `String` is an immutable type; `StringBuffer` may have its string changed in place.

`Runtime` provides a number of low-level methods associated with the Java run-time system. `System` provides similar low-level system methods. All the `System` methods are class methods, and this class may not be instantiated. `Math` is a similar class that supports only class methods--its methods provide floating-point math support.

The `Thread` class provides support for multiple threads of control running within the same Java interpreter. `Process` defines a platform-independent interface to platform-dependent processes running externally to the Java interpreter.

`Throwable` is the root class of the exception and error hierarchy. `Throwable` objects are used with the Java `throw` and `catch` statements. `java.lang` defines quite a few subclasses of `Throwable`. `Exception` and `Error` are the superclasses of all exceptions and errors. Figure 25.2 and Figure 25.3 show the class hierarchies for these core Java exceptions and errors.

Figure 25.2: The exception classes in the java.lang package

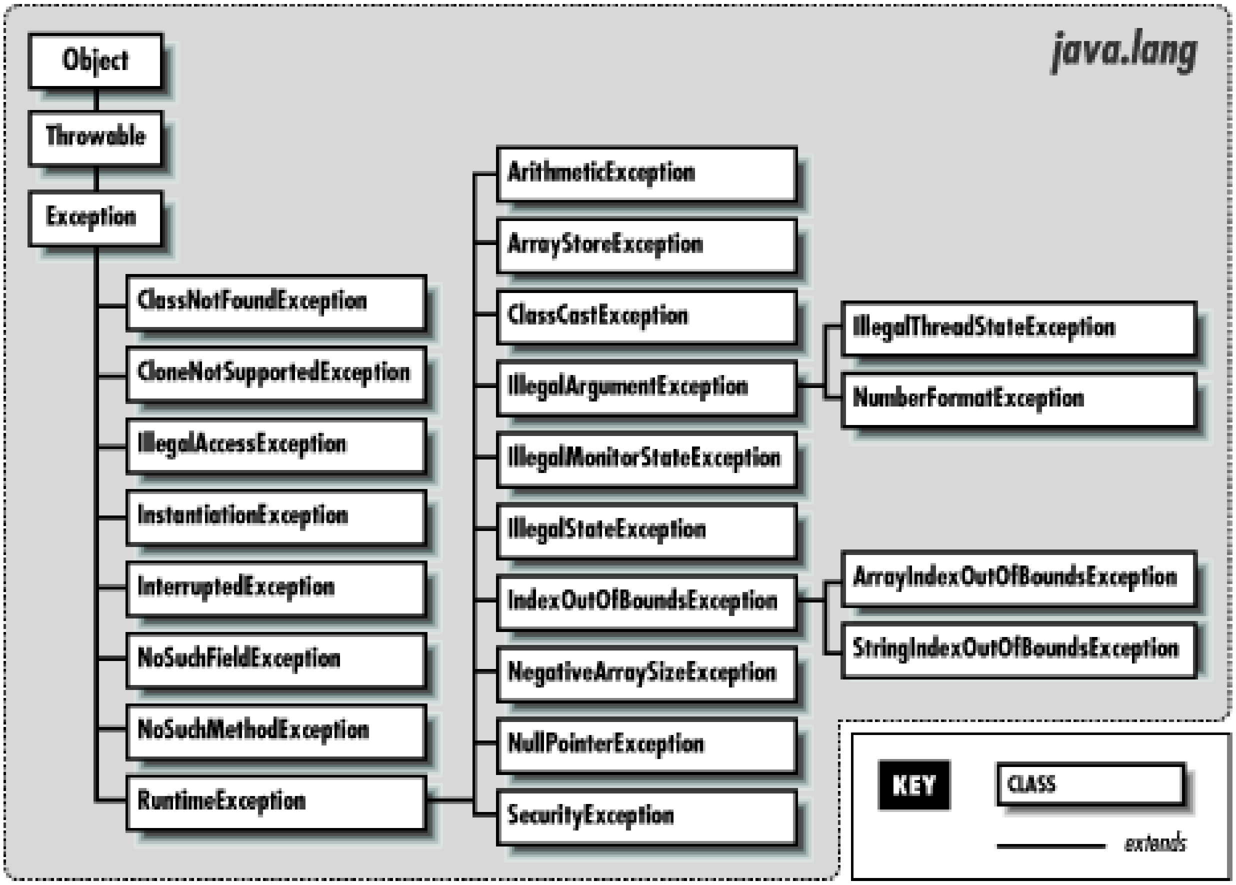
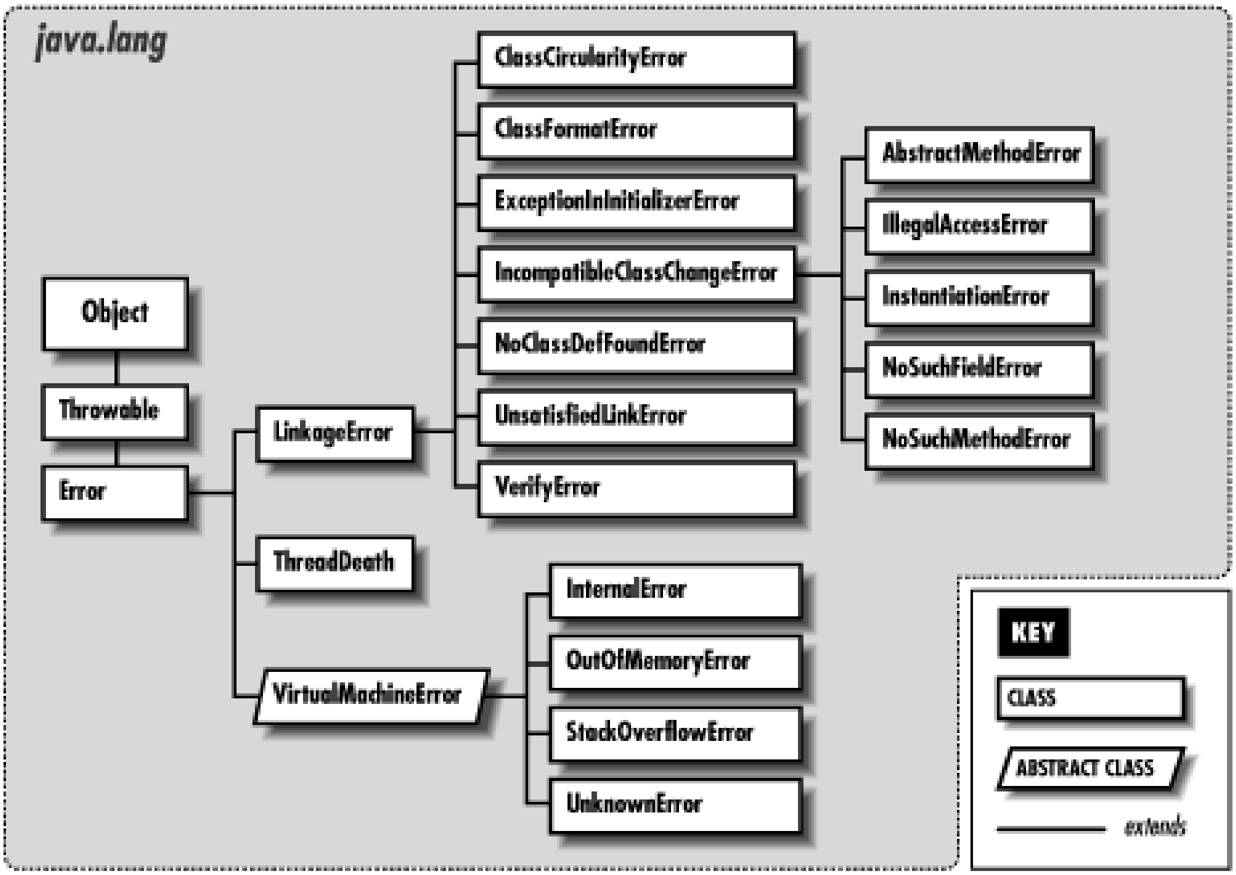


Figure 25.3: The error classes in the java.lang package



25.1 java.lang.AbstractMethodError (JDK 1.0)

Signals an attempt to invoke an abstract method.

```
public class AbstractMethodError extends IncompatibleClassChangeError {
    // Public Constructors
            public AbstractMethodError();
            public AbstractMethodError(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Error->LinkageError->IncompatibleClassChangeError->AbstractMethodError

PREVIOUS java.io.Writer (JDK 1.1)

HOME

BOOK INDEX

NEXT java.lang.ArithmeticException (JDK 1.0)

24.23 java.io.FilenameFilter (JDK 1.0)

This interface defines the `accept()` method that must be implemented by any object that filters filenames (i.e., selects a subset of filenames from a list of filenames). There are no standard `FilenameFilter` classes implemented by Java, but objects that implement this interface are used by the `java.awt.FileDialog` object, and by the `File.list()` method. A typical `FilenameFilter` object might check that the specified `File` represents a file (not a directory), is readable (and possibly writable as well), and that its name ends with some desired extension.

```
public abstract interface FilenameFilter {
    // Public Instance Methods
            public abstract boolean accept(File dir, String name);
}
```

Passed To:

File.list(), FileDialog.setFilenameFilter(), FileDialogPeer.setFilenameFilter()

Returned By:

FileDialog.getFilenameFilter()

28.15 java.net.ServerSocket (JDK 1.0)

This class is used by servers to listen for connection requests from clients. When you create a `ServerSocket`, you specify the port to listen on. The `accept()` method begins listening on that port, and blocks until a client requests a connection on that port. At that point, `accept()` accepts the connection, creating and returning a `Socket` that the server can use to communicate with the client.

```
public class ServerSocket extends Object {
    // Public Constructors
            public ServerSocket(int port) throws IOException;
            public ServerSocket(int port, int backlog) throws IOException;
        1.1public ServerSocket(int port, int backlog, InetAddress bindAddr) throws 
IOException;
    // Class Methods
            public static synchronized void setSocketFactory(SocketImplFactory fac) 
throws IOException;
    // Public Instance Methods
            public Socket accept() throws IOException;
            public void close() throws IOException;
            public InetAddress getInetAddress();
            public int getLocalPort();
        1.1public synchronized int getSoTimeout() throws IOException;
        1.1public synchronized void setSoTimeout(int timeout) throws SocketException;
            public String toString();  // Overrides Object
    // Protected Instance Methods
        1.1  protected final void implAccept(Socket s) throws IOException;
}
```

28.18 java.net.SocketImpl (JDK 1.0)

This abstract class defines the methods necessary to implement communication through sockets. Different subclasses of this class may provide different implementations suitable in different environments (such as behind firewalls). These socket implementations are used by the `Socket` and `ServerSocket` classes.

Normal applications never need to use or subclass this class.

```
public abstract class SocketImpl extends Object {
    // Default Constructor: public SocketImpl()
    // Protected Instance Variables
            protected InetAddress address;
            protected FileDescriptor fd;
            protected int localport;
            protected int port;
    // Public Instance Methods
            public String toString();  // Overrides Object
    // Protected Instance Methods
            protected abstract void accept(SocketImpl s) throws IOException;
            protected abstract int available() throws IOException;
            protected abstract void bind(InetAddress host, int port) throws 
IOException;
            protected abstract void close() throws IOException;
            protected abstract void connect(String host, int port) throws 
IOException;
            protected abstract void connect(InetAddress address, int port) throws 
IOException;
            protected abstract void create(boolean stream) throws IOException;
            protected FileDescriptor getFileDescriptor();
            protected InetAddress getInetAddress();
            protected abstract InputStream getInputStream() throws IOException;
            protected int getLocalPort();
            protected abstract OutputStream getOutputStream() throws IOException;
            protected int getPort();
            protected abstract void listen(int backlog) throws IOException;
}
```

Passed To:

Socket(), SocketImpl.accept()

Returned By:

SocketImplFactory.createSocketImpl()

PREVIOUS | HOME | NEXT

java.net.SocketException (JDK 1.0) | BOOK INDEX | java.net.SocketImplFactory (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

18.15 java.awt.Component (JDK 1.0)

`Component` is the superclass of all GUI components (except menu components) in the `java.awt` package. You may not instantiate a `Component` directly; you must use a subclass.

`Component` defines many methods. Some of these are intended to be implemented by subclasses. Some are used internally by the AWT. Some are to be implemented to handle events. And many are useful utility methods for working with GUI components. `getParent()` returns the `Container` that a `Component` is contained in. `setBackground()`, `setForeground()`, and `setFont()` set the specified display attributes of a component. `hide()`, `show()`, `enable()`, and `disable()` perform the specified actions for a component. `createImage()` creates an `Image` object from a specified `ImageProducer`, or creates an offscreen image that can be draw into and used for double-buffering during animation. `Component` also has quite a few deprecated methods as a result of the Java 1.1 event model and the introduction of the JavaBeans method naming conventions. The class defines quite a few methods for handling many types of events using the 1.0 model and the 1.1 model in both its high-level and low-level forms.

```
public abstract class Component extends Object implements ImageObserver,
MenuContainer, Serializable {
    // Protected Constructor
        1.1  protected Component();
    // Constants
        1.1  public static final float BOTTOM_ALIGNMENT;
        1.1  public static final float CENTER_ALIGNMENT;
        1.1  public static final float LEFT_ALIGNMENT;
        1.1  public static final float RIGHT_ALIGNMENT;
        1.1  public static final float TOP_ALIGNMENT;
    // Public Instance Methods
         #  public boolean action(Event evt, Object what);
        1.1  public synchronized void add(PopupMenu popup);
        1.1  public synchronized void addComponentListener(ComponentListener l);
        1.1  public synchronized void addFocusListener(FocusListener l);
        1.1  public synchronized void addKeyListener(KeyListener l);
        1.1  public synchronized void addMouseListener(MouseListener l);
        1.1  public synchronized void addMouseMotionListener(MouseMotionListener l);
             public void addNotify();
         #  public Rectangle bounds();
             public int checkImage(Image image, ImageObserver observer);
             public int checkImage(Image image, int width, int height, ImageObserver
observer);
        1.1  public boolean contains(int x, int y);
        1.1  public boolean contains(Point p);
             public Image createImage(ImageProducer producer);
             public Image createImage(int width, int height);
         #  public void deliverEvent(Event e);
         #  public void disable();
        1.1  public final void dispatchEvent(AWTEvent e);
        1.1  public void doLayout();
         #  public void enable();
         #  public void enable(boolean b);
        1.1  public float getAlignmentX();
        1.1  public float getAlignmentY();
             public Color getBackground();
        1.1  public Rectangle getBounds();
             public ColorModel getColorModel();
        1.1  public Component getComponentAt(int x, int y);
        1.1  public Component getComponentAt(Point p);
        1.1  public Cursor getCursor();
             public Font getFont();  // From MenuContainer
             public FontMetrics getFontMetrics(Font font);
             public Color getForeground();
             public Graphics getGraphics();
        1.1  public Locale getLocale();
        1.1  public Point getLocation();
        1.1  public Point getLocationOnScreen();
        1.1  public Dimension getMaximumSize();
        1.1  public Dimension getMinimumSize();
        1.1  public String getName();
             public Container getParent();
         #  public ComponentPeer getPeer();
        1.1  public Dimension getPreferredSize();
        1.1  public Dimension getSize();
             public Toolkit getToolkit();
        1.1  public final Object getTreeLock();
         #  public boolean gotFocus(Event evt, Object what);
         #  public boolean handleEvent(Event evt);
         #  public void hide();
            public boolean imageUpdate(Image img, int flags, int x, int y, int w, int
h);  // From ImageObserver
         #  public boolean inside(int x, int y);
            public void invalidate();
            public boolean isEnabled();
        1.1  public boolean isFocusTraversable();
            public boolean isShowing();
            public boolean isValid();
            public boolean isVisible();
         #  public boolean keyDown(Event evt, int key);
         #  public boolean keyUp(Event evt, int key);
         #  public void layout();
            public void list();
            public void list(PrintStream out);
            public void list(PrintStream out, int indent);
        1.1  public void list(PrintWriter out);
        1.1  public void list(PrintWriter out, int indent);
         #  public Component locate(int x, int y);
         #  public Point location();
         #  public boolean lostFocus(Event evt, Object what);
         #  public Dimension minimumSize();
         #  public boolean mouseDown(Event evt, int x, int y);
         #  public boolean mouseDrag(Event evt, int x, int y);
         #  public boolean mouseEnter(Event evt, int x, int y);
         #  public boolean mouseExit(Event evt, int x, int y);
         #  public boolean mouseMove(Event evt, int x, int y);
         #  public boolean mouseUp(Event evt, int x, int y);
         #  public void move(int x, int y);
         #  public void nextFocus();
            public void paint(Graphics g);
            public void paintAll(Graphics g);
         #  public boolean postEvent(Event e);  // From MenuContainer
         #  public Dimension preferredSize();
            public boolean prepareImage(Image image, ImageObserver observer);
            public boolean prepareImage(Image image, int width, int height, 
ImageObserver observer);
            public void print(Graphics g);
            public void printAll(Graphics g);
        1.1  public synchronized void remove(MenuComponent popup);  // From 
MenuContainer
        1.1  public synchronized void removeComponentListener(ComponentListener l);
        1.1  public synchronized void removeFocusListener(FocusListener l);
        1.1  public synchronized void removeKeyListener(KeyListener l);
        1.1  public synchronized void removeMouseListener(MouseListener l);
        1.1  public synchronized void removeMouseMotionListener(MouseMotionListener 
l);
            public void removeNotify();
            public void repaint();
            public void repaint(long tm);
            public void repaint(int x, int y, int width, int height);
            public void repaint(long tm, int x, int y, int width, int height);
            public void requestFocus();
         #  public void reshape(int x, int y, int width, int height);
         #  public void resize(int width, int height);
         #  public void resize(Dimension d);
            public void setBackground(Color c);
        1.1  public void setBounds(int x, int y, int width, int height);
        1.1  public void setBounds(Rectangle r);
        1.1  public synchronized void setCursor(Cursor cursor);
        1.1  public void setEnabled(boolean b);
            public synchronized void setFont(Font f);
            public void setForeground(Color c);
        1.1  public void setLocale(Locale l);
        1.1  public void setLocation(int x, int y);
        1.1  public void setLocation(Point p);
        1.1  public void setName(String name);
        1.1  public void setSize(int width, int height);
        1.1  public void setSize(Dimension d);
        1.1  public void setVisible(boolean b);
        #   public void show();
        #   public void show(boolean b);
        #   public Dimension size();
            public String toString();  // Overrides Object
        1.1  public void transferFocus();
            public void update(Graphics g);
            public void validate();
    // Protected Instance Methods
        1.1  protected final void disableEvents(long eventsToDisable);
        1.1  protected final void enableEvents(long eventsToEnable);
            protected String paramString();
        1.1  protected void processComponentEvent(ComponentEvent e);
        1.1  protected void processEvent(AWTEvent e);
        1.1  protected void processFocusEvent(FocusEvent e);
        1.1  protected void processKeyEvent(KeyEvent e);
        1.1  protected void processMouseEvent(MouseEvent e);
        1.1  protected void processMouseMotionEvent(MouseEvent e);
}
```

Extended By:

Button, Canvas, Checkbox, Choice, Container, Label, List, Scrollbar, TextComponent

Passed To:

BorderLayout.addLayoutComponent(), BorderLayout.removeLayoutComponent(), CardLayout.addLayoutComponent(), CardLayout.removeLayoutComponent(), ComponentEvent(), Container.add(), Container.addImpl(), Container.isAncestorOf(), Container.remove(), ContainerEvent(), FlowLayout.addLayoutComponent(), FlowLayout.removeLayoutComponent(), FocusEvent(), GridBagLayout.addLayoutComponent(), GridBagLayout.getConstraints(), GridBagLayout.lookupConstraints(), GridBagLayout.removeLayoutComponent(), GridBagLayout.setConstraints(), GridLayout.addLayoutComponent(), GridLayout.removeLayoutComponent(), KeyEvent(), LayoutManager.addLayoutComponent(), LayoutManager.removeLayoutComponent(), LayoutManager2.addLayoutComponent(), MediaTracker(), MouseEvent(), PaintEvent(), PopupMenu.show(), ScrollPane.addImpl(), Toolkit.createComponent(), Toolkit.getNativeContainer()

Returned By:

Component.getComponentAt(), Component.locate(), ComponentEvent.getComponent(), Container.add(), Container.getComponent(), Container.getComponentAt(), Container.getComponents(), Container.locate(), ContainerEvent.getChild(), PropertyEditor.getCustomEditor(), PropertyEditorSupport.getCustomEditor(), Window.getFocusOwner()

18.20 java.awt.Event (JDK 1.0)

This class contains public instance variables that describe some kind of GUI event. In Java 1.1, this class has been superseded by `AWTEvent` and the `java.awt.event` package.

The class contains a large number of constants. Some of the constants specify the event type and are values for the `id` variable. Other constants are values for keys, like the function keys, that do not have ASCII (or Latin-1) values, and are set on the `key` field. Other constants are mask values that are ORed into the `modifiers` field to describe the state of the modifier keys on the keyboard. The `target` field is very important--it is the object for which the event occurred. The `when` field specifies when the event occurred. The `x` and `y` fields specify the mouse coordinates at which it occurred. Finally, the `arg` field is a value specific to the type of the event. Not all fields have valid values for all types of events.

```
public class Event extends Object implements Serializable {
    // Public Constructors
            public Event(Object target, long when, int id, int x, int y, int key, int 
modifiers, Object arg);
            public Event(Object target, long when, int id, int x, int y, int key, int 
modifiers);
            public Event(Object target, int id, Object arg);
    // Event Type Constants
            public static final int ACTION_EVENT;
            public static final int GOT_FOCUS, LOST_FOCUS;
            public static final int KEY_ACTION, KEY_ACTION_RELEASE;
            public static final int KEY_PRESS, KEY_RELEASE;
            public static final int LIST_SELECT, LIST_DESELECT;
            public static final int LOAD_FILE, SAVE_FILE;
            public static final int MOUSE_DOWN, MOUSE_UP;
            public static final int MOUSE_DRAG, MOUSE_MOVE;
            public static final int MOUSE_ENTER, MOUSE_EXIT;
            public static final int SCROLL_ABSOLUTE;
        1.1  public static final int SCROLL_BEGIN, SCROLL_END;
            public static final int SCROLL_LINE_DOWN, SCROLL_LINE_UP;
            public static final int SCROLL_PAGE_DOWN, SCROLL_PAGE_UP;
            public static final int WINDOW_EXPOSE;
            public static final int WINDOW_ICONIFY, WINDOW_DEICONIFY;
            public static final int WINDOW_DESTROY;
            public static final int WINDOW_MOVED;
    // Keyboard Modifier Constants
            public static final int ALT_MASK;
            public static final int CTRL_MASK;
            public static final int META_MASK;
            public static final int SHIFT_MASK;
    // Function Key Constants
            public static final int F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11,
F12;
            public static final int LEFT, RIGHT, UP, DOWN;
            public static final int PGUP, PGDN;
            public static final int HOME, END;
        1.1  public static final int INSERT, DELETE;
        1.1  public static final int BACK_SPACE;
        1.1  public static final int ENTER;
        1.1  public static final int ESCAPE;
        1.1  public static final int TAB;
        1.1  public static final int CAPS_LOCK, NUM_LOCK, SCROLL_LOCK;
        1.1  public static final int PAUSE, PRINT_SCREEN;
    // Public Instance Variables
            public Object arg;
            public int clickCount;
            public Event evt;
            public int id;
            public int key;
            public int modifiers;
            public Object target;
            public long when;
            public int x;
            public int y;
    // Public Instance Methods
            public boolean controlDown();
            public boolean metaDown();
            public boolean shiftDown();
            public String toString();  // Overrides Object
            public void translate(int x, int y);
    // Protected Instance Methods
            protected String paramString();
}
```

Passed To:

AWTEvent(), Component.action(), Component.deliverEvent(), Component.gotFocus(), Component.handleEvent(), Component.keyDown(), Component.keyUp(), Component.lostFocus(), Component.mouseDown(), Component.mouseDrag(), Component.mouseEnter(), Component.mouseExit(), Component.mouseMove(), Component.mouseUp(), Component.postEvent(), Container.deliverEvent(), MenuComponent.postEvent(), MenuContainer.postEvent(), PopupMenuPeer.show(), Window.postEvent()

Type Of:

Event.evt

PREVIOUS HOME NEXT

18.2 java.awt.AWTEvent (JDK 1.1)

This abstract class serves as the root event type for all AWT events in Java 1.1 and supersedes the Event class which was used in Java 1.0.

Each AWTEvent has a source object, as all EventObject objects do. You can query the source of an event with the inherited getSource() method. The AWTEvent class adds an event type, or "id," for every AWT event. Use getID() to query the type of the event. Subclasses of AWTEvent define various constants for this type field.

The various _MASK constants defined by this class allow Component.enableEvents() to be called by applets and custom components that want to receive various event types without having to register an EventListener object to receive them.

```
public abstract class AWTEvent extends EventObject {
    // Public Constructors
            public AWTEvent(Event event);
            public AWTEvent(Object source, int id);
    // Constants
            public static final long ACTION_EVENT_MASK;
            public static final long ADJUSTMENT_EVENT_MASK;
            public static final long COMPONENT_EVENT_MASK;
            public static final long CONTAINER_EVENT_MASK;
            public static final long FOCUS_EVENT_MASK;
            public static final long ITEM_EVENT_MASK;
            public static final long KEY_EVENT_MASK;
            public static final long MOUSE_EVENT_MASK;
            public static final long MOUSE_MOTION_EVENT_MASK;
            public static final int RESERVED_ID_MAX;
            public static final long TEXT_EVENT_MASK;
            public static final long WINDOW_EVENT_MASK;
    // Protected Instance Variables
```

```
            protected boolean consumed;
            protected int id;
    // Public Instance Methods
            public int getID();
            public String paramString();
            public String toString();  // Overrides EventObject
    // Protected Instance Methods
            protected void consume();
            protected boolean isConsumed();
}
```

Hierarchy:

Object->EventObject(Serializable)->AWTEvent

Extended By:

ActionEvent, AdjustmentEvent, ComponentEvent, ItemEvent, TextEvent

Passed To:

Button.processEvent(), Checkbox.processEvent(), CheckboxMenuItem.processEvent(), Choice.processEvent(), Component.dispatchEvent(), Component.processEvent(), ComponentPeer.handleEvent(), Container.processEvent(), EventQueue.postEvent(), List.processEvent(), MenuComponent.dispatchEvent(), MenuComponent.processEvent(), MenuItem.processEvent(), Scrollbar.processEvent(), TextComponent.processEvent(), TextField.processEvent(), Window.processEvent()

Returned By:

EventQueue.getNextEvent(), EventQueue.peekEvent()

PREVIOUS | HOME | NEXT
java.awt.AWTError (JDK 1.0) | BOOK INDEX | java.awt.AWTEventMulticaster (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

20. The java.awt.event Package

Contents:

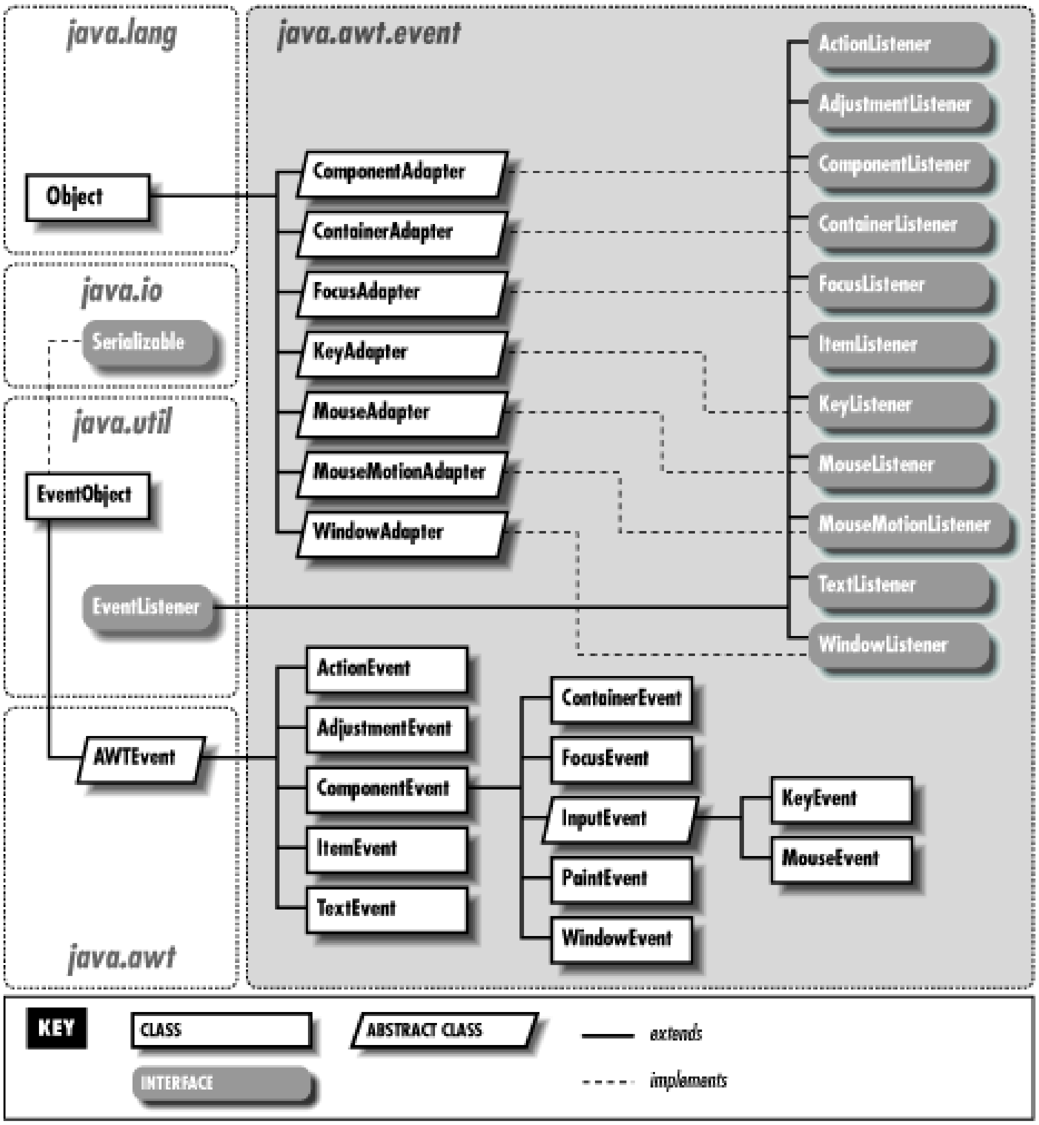
This package defines classes and interfaces used for event handling in the AWT. This package, and all of its classes and interfaces, are new in Java 1.1.

The members of this package fall into three categories:

- Events. The classes with names ending in "Event" represent specific types of events, generated by the AWT or by one of the AWT components.
- Listeners. The interfaces in this package are all "event listeners"; their names end with "Listener." These interfaces define the methods that must be implemented by any object that wants to be notified when a particular event occurs. Note that there is a `Listener` interface for each `Event` class.
- Adapters. Each of the classes with a name ending in "Adapter" provides a no-op implementation for an event listener interface that defines more than one method. When you are only interested in a single method of an event listener interface, it is easier to subclass an `Adapter` class than to implement all of the methods of the corresponding `Listener` interface.

Figure 20.1 shows the class hierarchy for this package. See Chapter 7, *Events*, for more information on events and event handling.

Figure 20.1: The java.awt.event package



20.1 java.awt.event.ActionEvent (JDK 1.1)

An object of this class represents a high-level "action" event generated by an AWT component. Instead of representing a direct user event, such as a mouse or keyboard event, ActionEvent represents some sort of action performed by the user on an AWT component.

The getID() method returns the type of action that has occurred. For AWT-generated action events, this type is always ActionEvent.ACTION_PERFORMED; custom components can generate action events of other types.

The getActionCommand() method returns a String that serves as a kind of name for the action that the event represents. The Button and MenuItem components have a setActionCommand() method that allows the programmer to specify an "action command" string to be included with any action events generated by those components. It is this value that is returned by the getActionCommand() method. When more than one Button or other component notifies the same ActionListener, you can use getActionCommand() to help determine the appropriate response to the event. This is generally a better technique than using the source object returned by getSource(). If no action command string is explicitly set, getActionCommand returns the label of the Button or MenuItem. Note, however, that internationalized programs should not rely on these labels to be constant.

Finally, getModifiers() returns a value that indicates what keyboard modifiers were in effect when the action event was triggered. Use the various _MASK constants, along with the & operator to decode this value.

```
public class ActionEvent extends AWTEvent {
    // Public Constructors
            public ActionEvent(Object source, int id, String command);
            public ActionEvent(Object source, int id, String command, int modifiers);
    // Constants
            public static final int ACTION_FIRST;
            public static final int ACTION_LAST;
            public static final int ACTION_PERFORMED;
            public static final int ALT_MASK;
            public static final int CTRL_MASK;
            public static final int META_MASK;
            public static final int SHIFT_MASK;
    // Public Instance Methods
            public String getActionCommand();
            public int getModifiers();
            public String paramString();  // Overrides AWTEvent
}
```

Hierarchy:

Object->EventObject(Serializable)->AWTEvent->ActionEvent

Passed To:

ActionListener.actionPerformed(), AWTEventMulticaster.actionPerformed(), Button.processActionEvent(), List.processActionEvent(), MenuItem.processActionEvent(), TextField.processActionEvent()

20.2 java.awt.event.ActionListener (JDK 1.1)

This interface defines the method that an object must implement to "listen" for action events on AWT components. When an `ActionEvent` occurs, an AWT component notifies its registered `ActionListener` objects by invoking their `actionPerformed()` methods.

```
public abstract interface ActionListener extends EventListener {
    // Public Instance Methods
            public abstract void actionPerformed(ActionEvent e);
}
```

Implemented By:

`AWTEventMulticaster`

Passed To:

`AWTEventMulticaster.add(), AWTEventMulticaster.remove(), Button.addActionListener(), Button.removeActionListener(), List.addActionListener(), List.removeActionListener(), MenuItem.addActionListener(), MenuItem.removeActionListener(), TextField.addActionListener(), TextField.removeActionListener()`

Returned By:

`AWTEventMulticaster.add(), AWTEventMulticaster.remove()`

java.awt.event.ActionEvent (JDK 1.1) | BOOK INDEX | java.awt.event.AdjustmentEvent (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

18.56 java.awt.SystemColor (JDK 1.1)

Instances of the `SystemColor` class represent colors used in the system desktop. You can use these colors to produce applications and custom components that fit well in the desktop color scheme. On platforms that allow the desktop colors to be modified dynamically, the actual color represented by these symbolic system colors may be dynamically updated.

The `SystemColor` class does not have a constructor, but it defines constant `SystemColor` objects that represent each of the symbolic colors used by the system desktop.

If you need to compare a `SystemColor` object to a regular `Color` object, use the `getRGB()` method of both objects and compare the resulting values.

```
public final class SystemColor extends Color implements Serializable {
    // No Constructor
    // Color Constants
            public static final SystemColor activeCaption, activeCaptionBorder, 
activeCaptionText;
            public static final SystemColor control, controlDkShadow, 
controlHighlight;
            public static final SystemColor controlLtHighlight, controlShadow, 
controlText;
            public static final SystemColor desktop;
            public static final SystemColor inactiveCaption, inactiveCaptionBorder, 
inactiveCaptionText;
            public static final SystemColor info, infoText;
            public static final SystemColor menu, menuText;
            public static final SystemColor scrollbar;
            public static final SystemColor text, textHighlight, textHighlightText;
            public static final SystemColor textInactiveText, textText;
            public static final SystemColor window, windowBorder, windowText;
    // Color Index Constants
            public static final int ACTIVE_CAPTION, ACTIVE_CAPTION_BORDER, 
ACTIVE_CAPTION_TEXT;
            public static final int CONTROL, CONTROL_DK_SHADOW, CONTROL_HIGHLIGHT;
            public static final int CONTROL_LT_HIGHLIGHT, CONTROL_SHADOW, 
CONTROL_TEXT;
            public static final int DESKTOP;
            public static final int INACTIVE_CAPTION, INACTIVE_CAPTION_BORDER, 
INACTIVE_CAPTION_TEXT;
            public static final int INFO, INFO_TEXT;
            public static final int MENU, MENU_TEXT;
            public static final int NUM_COLORS;
            public static final int SCROLLBAR;
            public static final int TEXT, TEXT_HIGHLIGHT, TEXT_HIGHLIGHT_TEXT;
            public static final int TEXT_INACTIVE_TEXT, TEXT_TEXT;
            public static final int WINDOW, WINDOW_BORDER, WINDOW_TEXT;
    // Public Instance Methods
            public int getRGB();  // Overrides Color
            public String toString();  // Overrides Color
}
```

Hierarchy:

`Object->Color(Serializable)->SystemColor(Serializable)`

PREVIOUS	HOME	NEXT
java.awt.Shape (JDK 1.1)	BOOK INDEX	java.awt.TextArea (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

25.61 java.lang.Thread (JDK 1.0)

This class encapsulates all the information about a single thread of control running on the Java interpreter. To create a thread, you must pass a `Runnable` object (i.e., an object that implements the `Runnable` interface by defining a `run()` method) to the `Thread` constructor, or you must subclass `Thread` so that it defines its own `run()` method.

The `run()` method of the `Thread` or of the specified `Runnable` object is the "body" of the thread. It begins executing when the `start()` method of the `Thread` object is called. The thread runs until the `run()` method returns or until the `stop()` method of its `Thread` object is called. The static methods of this class operate on the currently running thread. The instance methods may be called from one thread to operate on a different thread.

`start()` starts a thread running. `stop()` stops it by throwing a `ThreadDeath` error. `suspend()` temporarily halts a thread. `resume()` allows it to resume. `sleep()` makes the current thread stop for a specified amount of time. `yield()` makes the current thread give up control to any other threads of equal priority that are waiting to run. `join()` waits for a thread to die. `interrupt()` wakes up a waiting or sleeping thread (with an `InterruptedException`) or sets an "interrupted" flag on a non-sleeping thread. A thread can test its own "interrupted" flag with `interrupted()` or can test the flag of another thread with `isInterrupted()`. The `Object` `wait()` method makes the current thread block until the object's `notify()` method is called by another thread.

```
public class Thread extends Object implements Runnable {
    // Public Constructors
            public Thread();
            public Thread(Runnable target);
            public Thread(ThreadGroup group, Runnable target);
            public Thread(String name);
            public Thread(ThreadGroup group, String name);
            public Thread(Runnable target, String name);
            public Thread(ThreadGroup group, Runnable target, String name);
    // Constants
            public static final int MAX_PRIORITY;
            public static final int MIN_PRIORITY;
            public static final int NORM_PRIORITY;
    // Class Methods
            public static int activeCount();
            public static native Thread currentThread();
            public static void dumpStack();
            public static int enumerate(Thread[] tarray);
            public static boolean interrupted();
            public static native void sleep(long millis) throws InterruptedException;
            public static void sleep(long millis, int nanos) throws
InterruptedException;
```

```
            public static native void yield();
    // Public Instance Methods
            public void checkAccess();
            public native int countStackFrames();
            public void destroy();
            public final String getName();
            public final int getPriority();
            public final ThreadGroup getThreadGroup();
            public void interrupt();
            public final native boolean isAlive();
            public final boolean isDaemon();
            public boolean isInterrupted();
            public final synchronized void join(long millis) throws 
InterruptedException;
            public final synchronized void join(long millis, int nanos) throws 
InterruptedException;
            public final void join() throws InterruptedException;
            public final void resume();
            public void run();  // From Runnable
            public final void setDaemon(boolean on);
            public final void setName(String name);
            public final void setPriority(int newPriority);
            public synchronized native void start();
            public final void stop();
            public final synchronized void stop(Throwable o);
            public final void suspend();
            public String toString();  // Overrides Object
}
```

Passed To:

SecurityManager.checkAccess(), Thread.enumerate(), ThreadGroup.enumerate(), ThreadGroup.uncaughtException()

Returned By:

Thread.currentThread()

PREVIOUS | HOME | NEXT
java.lang.System (JDK 1.0) | BOOK INDEX | java.lang.ThreadDeath (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

25.63 java.lang.ThreadGroup (JDK 1.0)

This class defines a group of threads and allows operations on the group as a whole. A `ThreadGroup` may contain `Thread` objects, as well as "child" `ThreadGroup` objects. All `ThreadGroup` objects are created as children of some other `ThreadGroup`, and thus there is a parent/child hierarchy of `ThreadGroup` objects.

Some programs may find it convenient to define their own `ThreadGroup`, but generally thread groups are only used by system-level applications.

```
public class ThreadGroup extends Object {
    // Public Constructors
            public ThreadGroup(String name);
            public ThreadGroup(ThreadGroup parent, String name);
    // Public Instance Methods
            public int activeCount();
            public int activeGroupCount();
        1.1public boolean allowThreadSuspension(boolean b);
            public final void checkAccess();
            public final void destroy();
            public int enumerate(Thread[] list);
            public int enumerate(Thread[] list, boolean recurse);
            public int enumerate(ThreadGroup[] list);
            public int enumerate(ThreadGroup[] list, boolean recurse);
            public final int getMaxPriority();
            public final String getName();
            public final ThreadGroup getParent();
            public final boolean isDaemon();
        1.1public synchronized boolean isDestroyed();
            public void list();
            public final boolean parentOf(ThreadGroup g);
            public final void resume();
            public final void setDaemon(boolean daemon);
            public final void setMaxPriority(int pri);
            public final void stop();
            public final void suspend();
            public String toString();  // Overrides Object
            public void uncaughtException(Thread t, Throwable e);
}
```

Passed To:

SecurityManager.checkAccess(), Thread(), ThreadGroup(), ThreadGroup.enumerate(), ThreadGroup.parentOf()

Returned By:

SecurityManager.getThreadGroup(), Thread.getThreadGroup(), ThreadGroup.getParent()

PREVIOUS | HOME | NEXT
java.lang.ThreadDeath (JDK 1.0) | BOOK INDEX | java.lang.Throwable (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

30.9 java.util.GregorianCalendar (JDK 1.1)

This concrete subclass of `Calendar` implements the "standard" solar calendar with years numbered from the birth of Christ, which is used in most locales throughout the world. You do not typically use this class directly, but instead obtain a `Calendar` object suitable for the default locale by calling `Calendar.getInstance()`. See `Calendar` for details on working with `Calendar` objects.

There is a discontinuity in the Gregorian calendar that represents the historical switch from the Julian calendar to the Gregorian calendar. By default `GregorianCalendar` assumes that this switch occurs on October 15, 1582. Most programs need not be concerned with this.

```
public class GregorianCalendar extends Calendar {
    // Public Constructors
            public GregorianCalendar();
            public GregorianCalendar(TimeZone zone);
            public GregorianCalendar(Locale aLocale);
            public GregorianCalendar(TimeZone zone, Locale aLocale);
            public GregorianCalendar(int year, int month, int date);
            public GregorianCalendar(int year, int month, int date, int hour, int
minute);
            public GregorianCalendar(int year, int month, int date, int hour, int
minute, int second);
    // Constants
            public static final int AD;
            public static final int BC;
    // Public Instance Methods
            public void add(int field, int amount);  // Defines Calendar
            public boolean after(Object when);  // Defines Calendar
            public boolean before(Object when);  // Defines Calendar
            public Object clone();  // Overrides Calendar
            public boolean equals(Object obj);  // Defines Calendar
            public int getGreatestMinimum(int field);  // Defines Calendar
            public final Date getGregorianChange();
            public int getLeastMaximum(int field);  // Defines Calendar
            public int getMaximum(int field);  // Defines Calendar
            public int getMinimum(int field);  // Defines Calendar
            public synchronized int hashCode();  // Overrides Object
            public boolean isLeapYear(int year);
            public void roll(int field, boolean up);  // Defines Calendar
            public void setGregorianChange(Date date);
    // Protected Instance Methods
            protected void computeFields();  // Defines Calendar
            protected void computeTime();  // Defines Calendar
}
```

Hierarchy:

Object->Calendar(Serializable, Cloneable)->GregorianCalendar

PREVIOUS | HOME | NEXT
java.util.EventObject (JDK 1.1) | BOOK INDEX | java.util.Hashtable (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

30.2 java.util.Calendar (JDK 1.1)

This abstract class defines methods used to perform date and time arithmetic. It also includes methods that convert dates and times to and from the machine-usable millisecond format used by the `Date` class and units like minutes, hours, days, weeks, months, and years that are more useful to humans.

As an abstract class, `Calendar` cannot be directly instantiated. Instead, it provides static `getInstance()` methods that return instances of a `Calendar` subclass suitable for use in a specified or default locale with a specified or default time zone.

`Calendar` defines a number of useful constants. Some of these are values that represent days of the week and months of the year. Other constants, such as `HOUR` and `DAY_OF_WEEK`, represent various fields of date and time information. These field constants are passed to a number of `Calendar` methods, such as `get()` and `set()`, in order to indicate what particular date or time field is of interest.

`setTime()` and the various `set()` methods set the date represented by a `Calendar` object. The `add()` method adds (or subtracts) values to a calendar field, incrementing the next larger field when the field "rolls over." `roll()` does the same, without modifying any but the specified field. `before()` and `after()` compare two `Calendar` objects.

Many of the methods of the `Calendar` class are replacements for methods of Date that have been deprecated in Java 1.1. While the `Calendar` class is used to convert a time value into its various hour, day, month, and other fields, it is not intended to present those fields into a form suitable for display to the end user. That function is performed by the `DateFormat` class in the `java.text` package, which handles internationalization issues.

See also `Date`, `DateFormat`, and `TimeZone`.

```
public abstract class Calendar extends Object implements Serializable, Cloneable {
    // Protected Constructors
            protected Calendar();
            protected Calendar(TimeZone zone, Locale aLocale);
    // Constants
            public static final int FIELD_COUNT;
    // Date and Time Field Constants
            public static final int ERA;
            public static final int YEAR;
            public static final int MONTH;
            public static final int WEEK_OF_YEAR, WEEK_OF_MONTH;
            public static final int DATE, DAY_OF_MONTH;
            public static final int DAY_OF_YEAR, DAY_OF_WEEK, DAY_OF_WEEK_IN_MONTH;
            public static final int ZONE_OFFSET, DST_OFFSET;
            public static final int AM_PM;
            public static final int HOUR, HOUR_OF_DAY;
            public static final int MINUTE;
            public static final int SECOND;
            public static final int MILLISECOND;
    // Field Value Constants
            public static final int JANUARY, FEBRUARY, MARCH, APRIL;
            public static final int MAY, JUNE, JULY, AUGUST;
            public static final int SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER;
            public static final int UNDECIMBER;
            public static final int SUNDAY, MONDAY, TUESDAY, WEDNESDAY;
            public static final int THURSDAY, FRIDAY, SATURDAY;
            public static final int AM, PM;
    // Protected Instance Variables
            protected boolean areFieldsSet;
            protected int[] fields;
            protected boolean[] isSet;
            protected boolean isTimeSet;
            protected long time;
    // Class Methods
            public static synchronized Locale[] getAvailableLocales();
            public static synchronized Calendar getInstance();
            public static synchronized Calendar getInstance(TimeZone zone);
            public static synchronized Calendar getInstance(Locale aLocale);
            public static synchronized Calendar getInstance(TimeZone zone, Locale 
aLocale);
    // Public Instance Methods
            public abstract void add(int field, int amount);
            public abstract boolean after(Object when);
            public abstract boolean before(Object when);
            public final void clear();
            public final void clear(int field);
            public Object clone();  // Overrides Object
            public abstract boolean equals(Object when);  // Overrides Object
            public final int get(int field);
            public int getFirstDayOfWeek();
            public abstract int getGreatestMinimum(int field);
            public abstract int getLeastMaximum(int field);
            public abstract int getMaximum(int field);
            public int getMinimalDaysInFirstWeek();
            public abstract int getMinimum(int field);
            public final Date getTime();
            public TimeZone getTimeZone();
            public boolean isLenient();
            public final boolean isSet(int field);
            public abstract void roll(int field, boolean up);
            public final void set(int field, int value);
            public final void set(int year, int month, int date);
            public final void set(int year, int month, int date, int hour, int 
minute);
```

```
            public final void set(int year, int month, int date, int hour, int
minute, int second);
            public void setFirstDayOfWeek(int value);
            public void setLenient(boolean lenient);
            public void setMinimalDaysInFirstWeek(int value);
            public final void setTime(Date date);
            public void setTimeZone(TimeZone value);
    // Protected Instance Methods
            protected void complete();
            protected abstract void computeFields();
            protected abstract void computeTime();
            protected long getTimeInMillis();
            protected final int internalGet(int field);
            protected void setTimeInMillis(long millis);
}
```

Extended By:

GregorianCalendar

Passed To:

DateFormat.setCalendar()

Returned By:

Calendar.getInstance(), DateFormat.getCalendar()

Type Of:

DateFormat.calendar

18.13 java.awt.Choice (JDK 1.0)

This class represents an "option menu" or "dropdown list." The `addItem()` method adds an item with the specified label to a `Choice` menu. `getSelectedIndex()` returns the numerical position of the selected item in the menu, and `getSelectedItem()` returns the label of the selected item.

```
public class Choice extends Component implements ItemSelectable {
    // Public Constructor
            public Choice();
    // Public Instance Methods
        1.1 public synchronized void add(String item);
            public synchronized void addItem(String item);
        1.1 public synchronized void addItemListener(ItemListener l);  // From 
ItemSelectable
            public void addNotify();  // Overrides Component
        #   public int countItems();
            public String getItem(int index);
        1.1 public int getItemCount();
            public int getSelectedIndex();
            public synchronized String getSelectedItem();
        1.1 public synchronized Object[] getSelectedObjects();  // From 
ItemSelectable
        1.1 public synchronized void insert(String item, int index);
        1.1 public synchronized void remove(String item);
        1.1 public synchronized void remove(int position);
        1.1 public synchronized void removeAll();
        1.1 public synchronized void removeItemListener(ItemListener l);  // From 
ItemSelectable
            public synchronized void select(int pos);
            public synchronized void select(String str);
    // Protected Instance Methods
            protected String paramString();  // Overrides Component
        1.1 protected void processEvent(AWTEvent e);  // Overrides Component
        1.1 protected void processItemEvent(ItemEvent e);
}
```

Hierarchy:

```
Object->Component(ImageObserver, MenuContainer, Serializable)-
>Choice(ItemSelectable)
```

Passed To:

```
Toolkit.createChoice()
```

PREVIOUS | HOME | NEXT

java.awt.CheckboxMenuItem (JDK 1.0) | BOOK INDEX | java.awt.Color (JDK 1.0)

22.5 java.awt.peer.ChoicePeer (JDK 1.0)

```
public abstract interface ChoicePeer extends ComponentPeer {
    // Public Instance Methods
        1.1  public abstract void add(String item, int index);
             public abstract void addItem(String item, int index);
        1.1  public abstract void remove(int index);
             public abstract void select(int index);
}
```

Returned By:

Toolkit.createChoice()

18.16 java.awt.Container (JDK 1.0)

This class implements a component that can contain other components. You cannot instantiate `Container` directly, but must use one of its subclasses, such as `Panel`, `Frame`, or `Dialog`. Once a `Container` is created, you may set its `LayoutManager` with `setLayout()`, add components to it with `add()`, and remove them with `remove()`. `getComponents()` returns an array of the components contained in a `Container`. `locate()` determines within which contained component a specified point falls. `list()` produces helpful debugging output.

```
public abstract class Container extends Component {
    // Protected Constructor
        1.1  protected Container();
    // Public Instance Methods
             public Component add(Component comp);
             public Component add(String name, Component comp);
             public Component add(Component comp, int index);
        1.1  public void add(Component comp, Object constraints);
        1.1  public void add(Component comp, Object constraints, int index);
        1.1  public void addContainerListener(ContainerListener l);
             public void addNotify();  // Overrides Component
        #    public int countComponents();
        #    public void deliverEvent(Event e);  // Overrides Component
        1.1  public void doLayout();  // Overrides Component
        1.1  public float getAlignmentX();  // Overrides Component
        1.1  public float getAlignmentY();  // Overrides Component
             public Component getComponent(int n);
        1.1  public Component getComponentAt(int x, int y);  // Overrides Component
        1.1  public Component getComponentAt(Point p);  // Overrides Component
        1.1  public int getComponentCount();
             public Component[] getComponents();
        1.1  public Insets getInsets();
             public LayoutManager getLayout();
        1.1  public Dimension getMaximumSize();  // Overrides Component
        1.1  public Dimension getMinimumSize();  // Overrides Component
        1.1  public Dimension getPreferredSize();  // Overrides Component
        #    public Insets insets();
        1.1  public void invalidate();  // Overrides Component
        1.1  public boolean isAncestorOf(Component c);
        #    public void layout();  // Overrides Component
             public void list(PrintStream out, int indent);  // Overrides Component
        1.1  public void list(PrintWriter out, int indent);  // Overrides Component
        #   public Component locate(int x, int y);  // Overrides Component
        #   public Dimension minimumSize();  // Overrides Component
        1.1  public void paint(Graphics g);  // Overrides Component
            public void paintComponents(Graphics g);
        #   public Dimension preferredSize();  // Overrides Component
        1.1  public void print(Graphics g);  // Overrides Component
            public void printComponents(Graphics g);
        1.1  public void remove(int index);
            public void remove(Component comp);
            public void removeAll();
        1.1  public void removeContainerListener(ContainerListener l);
            public void removeNotify();  // Overrides Component
            public void setLayout(LayoutManager mgr);
            public void validate();  // Overrides Component
    // Protected Instance Methods
        1.1  protected void addImpl(Component comp, Object constraints, int index);
            protected String paramString();  // Overrides Component
        1.1  protected void processContainerEvent(ContainerEvent e);
        1.1  protected void processEvent(AWTEvent e);  // Overrides Component
        1.1  protected void validateTree();
}
```

Hierarchy:

Object->Component(ImageObserver, MenuContainer, Serializable)->Container

Extended By:

Panel, ScrollPane, Window

Passed To:

BorderLayout.getLayoutAlignmentX(), BorderLayout.getLayoutAlignmentY(), BorderLayout.invalidateLayout(), BorderLayout.layoutContainer(), BorderLayout.maximumLayoutSize(), BorderLayout.minimumLayoutSize(), BorderLayout.preferredLayoutSize(), CardLayout.first(), CardLayout.getLayoutAlignmentX(), CardLayout.getLayoutAlignmentY(), CardLayout.invalidateLayout(), CardLayout.last(), CardLayout.layoutContainer(), CardLayout.maximumLayoutSize(), CardLayout.minimumLayoutSize(), CardLayout.next(), CardLayout.preferredLayoutSize(), CardLayout.previous(), CardLayout.show(), FlowLayout.layoutContainer(), FlowLayout.minimumLayoutSize(), FlowLayout.preferredLayoutSize(), GridBagLayout.ArrangeGrid(), GridBagLayout.getLayoutAlignmentX(), GridBagLayout.getLayoutAlignmentY(), GridBagLayout.GetLayoutInfo(), GridBagLayout.GetMinSize(), GridBagLayout.invalidateLayout(), GridBagLayout.layoutContainer(), GridBagLayout.maximumLayoutSize(), GridBagLayout.minimumLayoutSize(), GridBagLayout.preferredLayoutSize(), GridLayout.layoutContainer(), GridLayout.minimumLayoutSize(), GridLayout.preferredLayoutSize(), LayoutManager.layoutContainer(), LayoutManager.minimumLayoutSize(), LayoutManager.preferredLayoutSize(), LayoutManager2.getLayoutAlignmentX(), LayoutManager2.getLayoutAlignmentY(), LayoutManager2.invalidateLayout(), LayoutManager2.maximumLayoutSize()

Returned By:

Component.getParent(), ContainerEvent.getContainer(), Toolkit.getNativeContainer()

18.38 java.awt.List (JDK 1.0)

This class is a `Component` that graphically displays a list of strings. The list is scrollable if necessary. The constructor takes optional arguments that specify the number of visible rows in the list and whether selection of more than one item is allowed. The various instance methods allow strings to be added and removed from the `List`, and allow the selected item or items to be queried.

```
public class List extends Component implements ItemSelectable {
    // Public Constructors
            public List();
        1.1  public List(int rows);
            public List(int rows, boolean multipleMode);
    // Public Instance Methods
        1.1  public void add(String item);
        1.1  public synchronized void add(String item, int index);
        1.1  public synchronized void addActionListener(ActionListener l);
            public void addItem(String item);
            public synchronized void addItem(String item, int index);
        1.1  public synchronized void addItemListener(ItemListener l);  // From
ItemSelectable
            public void addNotify();  // Overrides Component
        #   public boolean allowsMultipleSelections();
        #   public synchronized void clear();
        #   public int countItems();
            public synchronized void delItem(int position);
        #   public synchronized void delItems(int start, int end);
            public synchronized void deselect(int index);
            public String getItem(int index);
        1.1  public int getItemCount();
        1.1  public synchronized String[] getItems();
        1.1  public Dimension getMinimumSize(int rows);
        1.1  public Dimension getMinimumSize();  // Overrides Component
        1.1  public Dimension getPreferredSize(int rows);
        1.1  public Dimension getPreferredSize();  // Overrides Component
            public int getRows();
            public synchronized int getSelectedIndex();
            public synchronized int[] getSelectedIndexes();
            public synchronized String getSelectedItem();
            public synchronized String[] getSelectedItems();
        1.1  public Object[] getSelectedObjects();  // From ItemSelectable
            public int getVisibleIndex();
        1.1  public boolean isIndexSelected(int index);
        1.1  public boolean isMultipleMode();
        #   public boolean isSelected(int index);
            public synchronized void makeVisible(int index);
        #   public Dimension minimumSize(int rows);
        #   public Dimension minimumSize();  // Overrides Component
        #   public Dimension preferredSize(int rows);
        #   public Dimension preferredSize();  // Overrides Component
        1.1  public synchronized void remove(String item);
        1.1  public synchronized void remove(int position);
        1.1  public synchronized void removeActionListener(ActionListener l);
        1.1  public synchronized void removeAll();
        1.1  public synchronized void removeItemListener(ItemListener l);  // From
ItemSelectable
            public void removeNotify();  // Overrides Component
            public synchronized void replaceItem(String newValue, int index);
            public synchronized void select(int index);
        1.1  public synchronized void setMultipleMode(boolean b);
        #   public synchronized void setMultipleSelections(boolean b);
    // Protected Instance Methods
            protected String paramString();  // Overrides Component
        1.1  protected void processActionEvent(ActionEvent e);
        1.1  protected void processEvent(AWTEvent e);  // Overrides Component
        1.1  protected void processItemEvent(ItemEvent e);
}
```

Hierarchy:

```
Object->Component(ImageObserver, MenuContainer, Serializable)-
>List(ItemSelectable)
```

Passed To:

```
Toolkit.createList()
```

PREVIOUS | HOME | NEXT
java.awt.LayoutManager2 (JDK 1.1) | BOOK INDEX | java.awt.MediaTracker (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

22.14 java.awt.peer.ListPeer (JDK 1.0)

```
public abstract interface ListPeer extends ComponentPeer {
    // Public Instance Methods
        1.1  public abstract void add(String item, int index);
            public abstract void addItem(String item, int index);
            public abstract void clear();
            public abstract void delItems(int start, int end);
            public abstract void deselect(int index);
        1.1  public abstract Dimension getMinimumSize(int rows);
        1.1  public abstract Dimension getPreferredSize(int rows);
            public abstract int[] getSelectedIndexes();
            public abstract void makeVisible(int index);
            public abstract Dimension minimumSize(int v);
            public abstract Dimension preferredSize(int v);
        1.1  public abstract void removeAll();
            public abstract void select(int index);
        1.1  public abstract void setMultipleMode(boolean b);
            public abstract void setMultipleSelections(boolean v);
}
```

Returned By:

Toolkit.createList()

18.40 java.awt.Menu (JDK 1.0)

This class represents a pulldown menu pane that appears within a `MenuBar`. Each `Menu` has a label that appears in the `MenuBar` and may optionally be a tear-off menu. The `add()` and `addSeparator()` methods add individual items to a `Menu`.

```
public class Menu extends MenuItem implements MenuContainer {
    // Public Constructors
        1.1  public Menu();
            public Menu(String label);
            public Menu(String label, boolean tearOff);
    // Public Instance Methods
            public synchronized MenuItem add(MenuItem mi);
            public void add(String label);
            public void addNotify();  // Overrides MenuItem
            public void addSeparator();
        #   public int countItems();
            public MenuItem getItem(int index);
        1.1  public int getItemCount();
        1.1  public synchronized void insert(MenuItem menuitem, int index);
        1.1  public void insert(String label, int index);
        1.1  public void insertSeparator(int index);
            public boolean isTearOff();
        1.1  public String paramString();  // Overrides MenuItem
            public synchronized void remove(int index);
            public synchronized void remove(MenuComponent item);  // From
MenuContainer
        1.1  public synchronized void removeAll();
            public void removeNotify();  // Overrides MenuComponent
}
```

Hierarchy:

```
Object->MenuComponent(Serializable)->MenuItem->Menu(MenuContainer)
```

Extended By:

`PopupMenu`

Passed To:

`MenuBar.add(), MenuBar.setHelpMenu(), MenuBarPeer.addHelpMenu(), MenuBarPeer.addMenu(), Toolkit.createMenu()`

Returned By:

`MenuBar.add(), MenuBar.getHelpMenu(), MenuBar.getMenu()`

PREVIOUS java.awt.MediaTracker (JDK 1.0) | HOME BOOK INDEX | NEXT java.awt.MenuBar (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

18.41 java.awt.MenuBar (JDK 1.0)

This class represents a menu bar. `add()` adds `Menu` objects to the menu bar, and `setHelpMenu()` adds a **Help** menu in a reserved location of the menu bar. A `MenuBar` object may be displayed within a `Frame` by passing it to `Frame.setMenuBar()`.

```
public class MenuBar extends MenuComponent implements MenuContainer {
    // Public Constructor
            public MenuBar();
    // Public Instance Methods
            public synchronized Menu add(Menu m);
            public void addNotify();
        #   public int countMenus();
        1.1  public void deleteShortcut(MenuShortcut s);
            public Menu getHelpMenu();
            public Menu getMenu(int i);
        1.1  public int getMenuCount();
        1.1  public MenuItem getShortcutMenuItem(MenuShortcut s);
            public synchronized void remove(int index);
            public synchronized void remove(MenuComponent m);  // From MenuContainer
            public void removeNotify();  // Overrides MenuComponent
            public synchronized void setHelpMenu(Menu m);
        1.1  public synchronized Enumeration shortcuts();
}
```

Hierarchy:

`Object->MenuComponent(Serializable)->MenuBar(MenuContainer)`

Passed To:

`Frame.setMenuBar(), FramePeer.setMenuBar(), Toolkit.createMenuBar()`

Returned By:

`Frame.getMenuBar()`

java.awt.Menu (JDK 1.0) BOOK INDEX java.awt.MenuComponent (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

18.52 java.awt.Rectangle (JDK 1.0)

This class defines a rectangle as the X and Y coordinate of its upper-left corner and a width and height. The instance methods perform various tests and transformations on the rectangle. The `x`, `y`, `width`, and `height` variables are public and may thus be manipulated directly. `Rectangle` objects are used in several places in the `java.awt` package to specify clipping rectangles and bounding boxes.

```
public class Rectangle extends Object implements Shape, Serializable {
    // Public Constructors
            public Rectangle();
        1.1  public Rectangle(Rectangle r);
            public Rectangle(int x, int y, int width, int height);
            public Rectangle(int width, int height);
            public Rectangle(Point p, Dimension d);
            public Rectangle(Point p);
            public Rectangle(Dimension d);
    // Public Instance Variables
            public int height;
            public int width;
            public int x;
            public int y;
    // Public Instance Methods
            public void add(int newx, int newy);
            public void add(Point pt);
            public void add(Rectangle r);
        1.1  public boolean contains(Point p);
        1.1  public boolean contains(int x, int y);
            public boolean equals(Object obj);  // Overrides Object
        1.1  public Rectangle getBounds();  // From Shape
        1.1  public Point getLocation();
        1.1  public Dimension getSize();
            public void grow(int h, int v);
            public int hashCode();  // Overrides Object
        #   public boolean inside(int x, int y);
            public Rectangle intersection(Rectangle r);
            public boolean intersects(Rectangle r);
            public boolean isEmpty();
        #   public void move(int x, int y);
        #   public void reshape(int x, int y, int width, int height);
        #   public void resize(int width, int height);
        1.1  public void setBounds(Rectangle r);
        1.1  public void setBounds(int x, int y, int width, int height);
        1.1  public void setLocation(Point p);
        1.1  public void setLocation(int x, int y);
        1.1  public void setSize(Dimension d);
        1.1  public void setSize(int width, int height);
            public String toString();  // Overrides Object
            public void translate(int x, int y);
            public Rectangle union(Rectangle r);
}
```

Passed To:

Component.setBounds(), GridBagLayout.AdjustForGravity(), PaintEvent(), PaintEvent.setUpdateRect(), PropertyEditor.paintValue(), PropertyEditorSupport.paintValue(), Rectangle(), Rectangle.add(), Rectangle.intersection(), Rectangle.intersects(), Rectangle.setBounds(), Rectangle.union()

Returned By:

Component.bounds(), Component.getBounds(), Graphics.getClipBounds(), Graphics.getClipRect(), PaintEvent.getUpdateRect(), Polygon.getBoundingBox(), Polygon.getBounds(), Rectangle.getBounds(), Rectangle.intersection(), Rectangle.union(), Shape.getBounds()

Type Of:

Polygon.bounds

18.7 java.awt.Button (JDK 1.0)

This class represents a GUI pushbutton that displays a specified textual label. Use `setActionCommand()` to specify an identifying string that is included in the `ActionEvent` events generated by the button.

```
public class Button extends Component {
    // Public Constructors
            public Button();
            public Button(String label);
    // Public Instance Methods
        1.1 public synchronized void addActionListener(ActionListener l);
            public void addNotify();  // Overrides Component
        1.1 public String getActionCommand();
            public String getLabel();
        1.1 public synchronized void removeActionListener(ActionListener l);
        1.1 public void setActionCommand(String command);
         public synchronized void setLabel(String label);
    // Protected Instance Methods
            protected String paramString();  // Overrides Component
        1.1 protected void processActionEvent(ActionEvent e);
        1.1 protected void processEvent(AWTEvent e);  // Overrides Component
}
```

Hierarchy:

`Object->Component(ImageObserver, MenuContainer, Serializable)->Button`

Passed To:

`Toolkit.createButton()`

java.awt.BorderLayout (JDK 1.0)

BOOK INDEX

java.awt.Canvas (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

18.44 java.awt.MenuItem (JDK 1.0)

This class encapsulates a menu item with a specified textual label. A `MenuItem` can be added to a menu pane with the `Menu.add()` method. The `disable()` method makes an item non-selectable; you might use it to "gray-out" a menu item when the command it represents is not valid in the current context. The `enable()` method makes an item selectable again. In Java 1.1, use `setActionCommand()` to specify an identifying string that is included in `ActionEvent` events generated by the menu item.

```
public class MenuItem extends MenuComponent {
    // Public Constructors
        1.1  public MenuItem();
            public MenuItem(String label);
        1.1  public MenuItem(String label, MenuShortcut s);
    // Public Instance Methods
        1.1  public synchronized void addActionListener(ActionListener l);
            public void addNotify();
        1.1  public void deleteShortcut();
        #   public synchronized void disable();
        #   public synchronized void enable();
        #   public void enable(boolean b);
        1.1  public String getActionCommand();
            public String getLabel();
        1.1  public MenuShortcut getShortcut();
            public boolean isEnabled();
            public String paramString();  // Overrides MenuComponent
        1.1  public synchronized void removeActionListener(ActionListener l);
        1.1  public void setActionCommand(String command);
        1.1  public synchronized void setEnabled(boolean b);
            public synchronized void setLabel(String label);
        1.1  public void setShortcut(MenuShortcut s);
    // Protected Instance Methods
        1.1  protected final void disableEvents(long eventsToDisable);
        1.1  protected final void enableEvents(long eventsToEnable);
        1.1  protected void processActionEvent(ActionEvent e);
        1.1  protected void processEvent(AWTEvent e);  // Overrides MenuComponent
}
```

Hierarchy:

```
Object->MenuComponent(Serializable)->MenuItem
```

Extended By:

```
CheckboxMenuItem, Menu
```

Passed To:

```
Menu.add(), Menu.insert(), MenuPeer.addItem(), Toolkit.createMenuItem()
```

Returned By:

```
Menu.add(), Menu.getItem(), MenuBar.getShortcutMenuItem()
```

PREVIOUS | HOME | NEXT

java.awt.MenuContainer (JDK 1.0) | BOOK INDEX | java.awt.MenuShortcut (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

18.59 java.awt.TextField (JDK 1.0)

This `Component` displays a line of optionally editable text. Most of its interesting methods are defined by the `TextComponent` superclass. Use `setEchoCharacter()` to specify a character to be echoed when requesting sensitive input such as a password.

See also `TextComponent` and `TextArea`.

```
public class TextField extends TextComponent {
    // Public Constructors
            public TextField();
            public TextField(String text);
            public TextField(int columns);
            public TextField(String text, int columns);
    // Public Instance Methods
        1.1  public synchronized void addActionListener(ActionListener l);
            public void addNotify();  // Overrides Component
            public boolean echoCharIsSet();
            public int getColumns();
            public char getEchoChar();
        1.1  public Dimension getMinimumSize(int columns);
        1.1  public Dimension getMinimumSize();  // Overrides Component
        1.1  public Dimension getPreferredSize(int columns);
        1.1  public Dimension getPreferredSize();  // Overrides Component
        #   public Dimension minimumSize(int columns);
        #   public Dimension minimumSize();  // Overrides Component
        #   public Dimension preferredSize(int columns);
        #   public Dimension preferredSize();  // Overrides Component
        1.1  public synchronized void removeActionListener(ActionListener l);
        1.1  public void setColumns(int columns);
        1.1  public void setEchoChar(char c);
        #   public void setEchoCharacter(char c);
    // Protected Instance Methods
            protected String paramString();  // Overrides TextComponent
        1.1  protected void processActionEvent(ActionEvent e);
        1.1  protected void processEvent(AWTEvent e);  // Overrides TextComponent
}
```

Hierarchy:

```
Object->Component(ImageObserver, MenuContainer, Serializable)->TextComponent-
>TextField
```

Passed To:

```
Toolkit.createTextField()
```

18.5 java.awt.Adjustable (JDK 1.1)

This interface defines the methods that should be implemented by an object that maintains a user-adjustable numeric value. This value has a specified minimum and maximum value, and it may be incremented or decremented either a unit at a time or a block at a time. An `Adjustable` object generates an `AdjustmentEvent` when it is adjusted, and it maintains a list of `AdjustmentListener` objects interested in being notified when such an event occurs.

This interface abstracts the essential functionality of the `Scrollbar` component.

```
public abstract interface Adjustable {
    // Constants
            public static final int HORIZONTAL;
            public static final int VERTICAL;
    // Public Instance Methods
            public abstract void addAdjustmentListener(AdjustmentListener l);
            public abstract int getBlockIncrement();
            public abstract int getMaximum();
            public abstract int getMinimum();
            public abstract int getOrientation();
            public abstract int getUnitIncrement();
            public abstract int getValue();
            public abstract int getVisibleAmount();
            public abstract void removeAdjustmentListener(AdjustmentListener l);
            public abstract void setBlockIncrement(int b);
            public abstract void setMaximum(int max);
            public abstract void setMinimum(int min);
            public abstract void setUnitIncrement(int u);
            public abstract void setValue(int v);
            public abstract void setVisibleAmount(int v);
}
```

Implemented By:

`Scrollbar`

Passed To:

AdjustmentEvent(), ScrollPanePeer.setUnitIncrement(), ScrollPanePeer.setValue()

Returned By:

AdjustmentEvent.getAdjustable(), ScrollPane.getHAdjustable(), ScrollPane.getVAdjustable()

PREVIOUS java.awt.AWTException (JDK 1.0) | HOME BOOK INDEX | NEXT java.awt.BorderLayout (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

18.54 java.awt.Scrollbar (JDK 1.0)

This `Component` represents a graphical scrollbar. `setValue()` sets the displayed value of the scrollbar. `setValues()` sets the displayed value, the page size, and the minimum and maximum values. The constants `HORIZONTAL` and `VERTICAL` are legal values for the scrollbar orientation.

```
public class Scrollbar extends Component implements Adjustable {
    // Public Constructors
            public Scrollbar();
            public Scrollbar(int orientation);
            public Scrollbar(int orientation, int value, int visible, int minimum, 
int maximum);
    // Constants
            public static final int HORIZONTAL;
            public static final int VERTICAL;
    // Public Instance Methods
        1.1  public synchronized void addAdjustmentListener(AdjustmentListener l);  
// From Adjustable
            public void addNotify();  // Overrides Component
        1.1  public int getBlockIncrement();  // From Adjustable
        #   public int getLineIncrement();
            public int getMaximum();  // From Adjustable
            public int getMinimum();  // From Adjustable
            public int getOrientation();  // From Adjustable
        #   public int getPageIncrement();
        1.1  public int getUnitIncrement();  // From Adjustable
            public int getValue();  // From Adjustable
        #   public int getVisible();
        1.1  public int getVisibleAmount();  // From Adjustable
        1.1  public synchronized void removeAdjustmentListener(AdjustmentListener l);  
// From Adjustable
        1.1  public synchronized void setBlockIncrement(int v);  // From Adjustable
        #   public void setLineIncrement(int v);
        1.1  public synchronized void setMaximum(int newMaximum);  // From Adjustable
        1.1  public synchronized void setMinimum(int newMinimum);  // From Adjustable
        1.1  public synchronized void setOrientation(int orientation);
        #   public void setPageIncrement(int v);
        1.1  public synchronized void setUnitIncrement(int v);  // From Adjustable
            public synchronized void setValue(int newValue);  // From Adjustable
            public synchronized void setValues(int value, int visible, int minimum, 
int maximum);
```

```
        1.1  public synchronized void setVisibleAmount(int newAmount);  // From 
Adjustable
    // Protected Instance Methods
            protected String paramString();  // Overrides Component
        1.1  protected void processAdjustmentEvent(AdjustmentEvent e);
        1.1  protected void processEvent(AWTEvent e);  // Overrides Component
}
```

Hierarchy:

`Object->Component(ImageObserver, MenuContainer, Serializable)->Scrollbar(Adjustable)`

Passed To:

`Toolkit.createScrollbar()`

PREVIOUS java.awt.ScrollPane (JDK 1.1) | HOME | BOOK INDEX | NEXT java.awt.Shape (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

21.5 java.awt.image.FilteredImageSource (JDK 1.0)

This class is an ImageProducer that produces image data filtered from some other ImageProducer. A FilteredImageSource is created with a specified ImageProducer and a specified ImageFilter. For example, an applet might use the following code to download and crop an image:

```
Image full_image = getImage(getDocumentBase(), "images/1.gif");
ImageFilter cropper = new CropImageFilter(10, 10, 100, 100);
ImageProducer prod = new FilteredImageSource(full_image.getSource(), cropper);
Image cropped_image = createImage(prod);
```

The methods of this class are the standard ImageProducer methods that you can invoke to add and remove ImageConsumer objects.

```
public class FilteredImageSource extends Object implements ImageProducer {
    // Public Constructor
            public FilteredImageSource(ImageProducer orig, ImageFilter imgf);
    // Public Instance Methods
            public synchronized void addConsumer(ImageConsumer ic);  // From 
ImageProducer
            public synchronized boolean isConsumer(ImageConsumer ic);  // From 
ImageProducer
            public synchronized void removeConsumer(ImageConsumer ic);  // From 
ImageProducer
            public void requestTopDownLeftRightResend(ImageConsumer ic);  // From 
ImageProducer
            public void startProduction(ImageConsumer ic);  // From ImageProducer
}
```

21.9 java.awt.image.ImageProducer (JDK 1.0)

This interface defines the methods that any class that produces image data must define to enable communication with `ImageConsumer` classes. An `ImageConsumer` registers itself as interested in a producer's image by calling the `addConsumer()` method.

Most applications never need to use or implement this interface.

```
public abstract interface ImageProducer {
    // Public Instance Methods
            public abstract void addConsumer(ImageConsumer ic);
            public abstract boolean isConsumer(ImageConsumer ic);
            public abstract void removeConsumer(ImageConsumer ic);
            public abstract void requestTopDownLeftRightResend(ImageConsumer ic);
            public abstract void startProduction(ImageConsumer ic);
}
```

Implemented By:

`FilteredImageSource, MemoryImageSource`

Passed To:

`Component.createImage(), ComponentPeer.createImage(), FilteredImageSource(), ImageFilter.resendTopDownLeftRight(), PixelGrabber(), Toolkit.createImage()`

Returned By:

`Image.getSource()`

21.11 java.awt.image.MemoryImageSource (JDK 1.0)

This class is an `ImageProducer` that produces an image from data stored in memory. The various constructors specify image data, color model, array offset, scan line length, and properties in slightly different ways. The instance methods implement the standard `ImageProducer` interface that allows an `ImageConsumer` object to register interest in the image.

```
public class MemoryImageSource extends Object implements ImageProducer {
    // Public Constructors
            public MemoryImageSource(int w, int h, ColorModel cm, byte[] pix, int 
off, int scan);
            public MemoryImageSource(int w, int h, ColorModel cm, byte[] pix, int 
off, int scan, Hashtable props);
            public MemoryImageSource(int w, int h, ColorModel cm, int[] pix, int off, 
int scan);
            public MemoryImageSource(int w, int h, ColorModel cm, int[] pix, int off, 
int scan, Hashtable props);
            public MemoryImageSource(int w, int h, int[] pix, int off, int scan);
            public MemoryImageSource(int w, int h, int[] pix, int off, int scan, 
Hashtable props);
    // Public Instance Methods
            public synchronized void addConsumer(ImageConsumer ic);  // From 
ImageProducer
            public synchronized boolean isConsumer(ImageConsumer ic);  // From 
ImageProducer
        1.1  public void newPixels();
        1.1  public synchronized void newPixels(int x, int y, int w, int h);
        1.1  public synchronized void newPixels(int x, int y, int w, int h, boolean 
framenotify);
        1.1  public synchronized void newPixels(byte[] newpix, ColorModel newmodel, 
int offset, int scansize);
        1.1  public synchronized void newPixels(int[] newpix, ColorModel newmodel, 
int offset, int scansize);
            public synchronized void removeConsumer(ImageConsumer ic);  // From 
ImageProducer
            public void requestTopDownLeftRightResend(ImageConsumer ic);  // From 
ImageProducer
        1.1  public synchronized void setAnimated(boolean animated);
        1.1  public synchronized void setFullBufferUpdates(boolean fullbuffers);
            public void startProduction(ImageConsumer ic);  // From ImageProducer
}
```

30.26 java.util.Vector (JDK 1.0)

This class implements an array of objects that grows in size as necessary. It is useful when you need to keep track of a number of objects but do not know in advance how many there will be. There are a number of methods for storing objects in and removing objects from the `Vector`. Other methods look up the object at specified positions in the `Vector`, or search for specified objects within the `Vector`. `elements()` returns an `Enumeration` of the objects stored in the `Vector`. `size()` returns the number of objects currently stored in the `Vector`. `capacity()` returns the number of elements that may be stored before the vector's internal storage must be reallocated.

```
public class Vector extends Object implements Cloneable, Serializable {
    // Public Constructors
            public Vector(int initialCapacity, int capacityIncrement);
            public Vector(int initialCapacity);
            public Vector();
    // Protected Instance Variables
            protected int capacityIncrement;
            protected int elementCount;
            protected Object[] elementData;
    // Public Instance Methods
            public final synchronized void addElement(Object obj);
            public final int capacity();
            public synchronized Object clone();  // Overrides Object
            public final boolean contains(Object elem);
            public final synchronized void copyInto(Object[] anArray);
            public final synchronized Object elementAt(int index);
            public final synchronized Enumeration elements();
            public final synchronized void ensureCapacity(int minCapacity);
            public final synchronized Object firstElement();
            public final int indexOf(Object elem);
            public final synchronized int indexOf(Object elem, int index);
            public final synchronized void insertElementAt(Object obj, int index);
            public final boolean isEmpty();
            public final synchronized Object lastElement();
            public final int lastIndexOf(Object elem);
            public final synchronized int lastIndexOf(Object elem, int index);
            public final synchronized void removeAllElements();
            public final synchronized boolean removeElement(Object obj);
            public final synchronized void removeElementAt(int index);
            public final synchronized void setElementAt(Object obj, int index);
            public final synchronized void setSize(int newSize);
            public final int size();
            public final synchronized String toString();  // Overrides Object
            public final synchronized void trimToSize();
}
```

Extended By:

Stack

PREVIOUS | HOME | NEXT

java.util.TooManyListenersException (JDK 1.1) | BOOK INDEX | The java.util.zip Package

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

22.15 java.awt.peer.MenuBarPeer (JDK 1.0)

```
public abstract interface MenuBarPeer extends MenuComponentPeer {
    // Public Instance Methods
            public abstract void addHelpMenu(Menu m);
            public abstract void addMenu(Menu m);
            public abstract void delMenu(int index);
}
```

Returned By:

Toolkit.createMenuBar()

18.53 java.awt.ScrollPane (JDK 1.1)

This `Container` class creates horizontal and vertical scrollbars surrounding a "viewport" and allows a single child component to be displayed and scrolled within this viewport. Typically the child of the `ScrollPane` is larger than the `ScrollPane` itself, so scrollbars allow the user to select the currently visible portion.

When you call the `ScrollPane()` constructor, you may optionally specify a scrollbar display policy, which should be one of the three constants defined by this class. If you do not specify a policy, `ScrollPane` uses the `SCROLLBARS_AS_NEEDED` policy.

A program can programmatically scroll the child within the viewport by calling `setScrollPosition()`. `getHAdjustable()` and `getVAdjustable()` return the horizontal and vertical `Adjustable` objects that control scrolling (typically these are not actually instances of `Scrollbar`). You can use these `Adjustable` objects to specify the unit and block increment values for the scrollbars. You can also directly set the `Adjustable` value, as an alternative to `setScrollPosition()`, but should not set other values of the `Adjustable` objects.

Use `setSize()` to set the size of the `ScrollPane` container. You may want to take the size of the scrollbars into account when computing the overall container size--use `getHScrollbarHeight()` and `getVScrollbarWidth()` to obtain these values.

`ScrollPane` overrides the `printComponents()` method of `Container` so that when a `ScrollPane` is printed, the entire child component is printed, rather than only the currently visible portion.

```
public class ScrollPane extends Container {
    // Public Constructors
            public ScrollPane();
            public ScrollPane(int scrollbarDisplayPolicy);
    // Constants
            public static final int SCROLLBARS_ALWAYS;
            public static final int SCROLLBARS_AS_NEEDED;
            public static final int SCROLLBARS_NEVER;
    // Public Instance Methods
            public void addNotify();  // Overrides Container
            public void doLayout();  // Overrides Container
            public Adjustable getHAdjustable();
            public int getHScrollbarHeight();
            public Point getScrollPosition();
            public int getScrollbarDisplayPolicy();
            public Adjustable getVAdjustable();
            public int getVScrollbarWidth();
            public Dimension getViewportSize();
        #   public void layout();  // Overrides Container
            public String paramString();  // Overrides Container
            public void printComponents(Graphics g);  // Overrides Container
            public final void setLayout(LayoutManager mgr);  // Overrides Container
            public void setScrollPosition(int x, int y);
            public void setScrollPosition(Point p);
    // Protected Instance Methods
            protected final void addImpl(Component comp, Object constraints, int
index);  // Overrides Container
}
```

Hierarchy:

```
Object->Component(ImageObserver, MenuContainer, Serializable)->Container->ScrollPane
```

Passed To:

```
Toolkit.createScrollPane()
```

PREVIOUS | HOME | NEXT
java.awt.Rectangle (JDK 1.0) | BOOK INDEX | java.awt.Scrollbar (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

22.18 java.awt.peer.MenuPeer (JDK 1.0)

```
public abstract interface MenuPeer extends MenuItemPeer {
    // Public Instance Methods
            public abstract void addItem(MenuItem item);
            public abstract void addSeparator();
            public abstract void delItem(int index);
}
```

Hierarchy:

(MenuPeer(MenuItemPeer(MenuComponentPeer)))

Extended By:

PopupMenuPeer

Returned By:

Toolkit.createMenu()

18.10 java.awt.Checkbox (JDK 1.0)

This class represents a GUI checkbox with a textual label. The `Checkbox` maintains a `boolean` state--whether it is checked or not. The checkbox may optionally be part of a `CheckboxGroup` which enforces "radio button" behavior.

```
public class Checkbox extends Component implements ItemSelectable {
    // Public Constructors
            public Checkbox();
            public Checkbox(String label);
        1.1 public Checkbox(String label, boolean state);
        1.1 public Checkbox(String label, boolean state, CheckboxGroup group);
            public Checkbox(String label, CheckboxGroup group, boolean state);
    // Public Instance Methods
        1.1 public synchronized void addItemListener(ItemListener l);  // From 
ItemSelectable
            public void addNotify();  // Overrides Component
            public CheckboxGroup getCheckboxGroup();
            public String getLabel();
        1.1 public Object[] getSelectedObjects();  // From ItemSelectable
            public boolean getState();
        1.1 public synchronized void removeItemListener(ItemListener l);  // From 
ItemSelectable
            public void setCheckboxGroup(CheckboxGroup g);
            public synchronized void setLabel(String label);
            public void setState(boolean state);
    // Protected Instance Methods
            protected String paramString();  // Overrides Component
        1.1 protected void processEvent(AWTEvent e);  // Overrides Component
        1.1 protected void processItemEvent(ItemEvent e);
}
```

Hierarchy:

```
Object->Component(ImageObserver, MenuContainer, Serializable)-
>Checkbox(ItemSelectable)
```

Passed To:

CheckboxGroup.setCurrent(), CheckboxGroup.setSelectedCheckbox(), Toolkit.createCheckbox()

Returned By:

CheckboxGroup.getCurrent(), CheckboxGroup.getSelectedCheckbox()

PREVIOUS HOME NEXT
java.awt.CardLayout (JDK 1.0) BOOK INDEX java.awt.CheckboxGroup (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

18.12 java.awt.CheckboxMenuItem (JDK 1.0)

This class represents a checkbox with a textual label in a GUI menu. It maintains a `boolean` state--whether it is checked or not. See also `MenuItem`.

```
public class CheckboxMenuItem extends MenuItem implements ItemSelectable {
    // Public Constructors
        1.1 public CheckboxMenuItem();
            public CheckboxMenuItem(String label);
        1.1 public CheckboxMenuItem(String label, boolean state);
    // Public Instance Methods
        1.1 public synchronized void addItemListener(ItemListener l);  // From 
ItemSelectable
            public void addNotify();  // Overrides MenuItem
        1.1 public synchronized Object[] getSelectedObjects();  // From 
ItemSelectable
            public boolean getState();
            public String paramString();  // Overrides MenuItem
        1.1 public synchronized void removeItemListener(ItemListener l);  // From 
ItemSelectable
            public synchronized void setState(boolean b);
    // Protected Instance Methods
        1.1 protected void processEvent(AWTEvent e);  // Overrides MenuItem
        1.1  protected void processItemEvent(ItemEvent e);
}
```

Hierarchy:

`Object->MenuComponent(Serializable)->MenuItem->CheckboxMenuItem(ItemSelectable)`

Passed To:

`Toolkit.createCheckboxMenuItem()`

18.34 java.awt.ItemSelectable (JDK 1.1)

This interface abstracts the functionality of an AWT component that presents one or more items to the user and allows the user to select zero or more of them. It is implemented by several components in the AWT.

`getSelectedObjects()` returns an array of selected objects, or `null`, if none are selected. `addItemListener()` and `removeItemListener()` are standard methods for adding and removing `ItemListener` objects to be notified when an item is selected.

```
public abstract interface ItemSelectable {
    // Public Instance Methods
            public abstract void addItemListener(ItemListener l);
            public abstract Object[] getSelectedObjects();
            public abstract void removeItemListener(ItemListener l);
}
```

Implemented By:

`Checkbox, CheckboxMenuItem, Choice, List`

Passed To:

`ItemEvent()`

Returned By:

`ItemEvent.getItemSelectable()`

18.6 java.awt.BorderLayout (JDK 1.0)

This class implements the `LayoutManager` interface to lay out `Component` objects in a `Container`. The `BorderLayout` arranges components that have been added to their `Container` (using the `Container.add()` method) with the names "North," "South," "East," "West," and "Center." These named components are arranged along the edges and in the center of the container. The `hgap` and `vgap` arguments to the `BorderLayout` constructor specify the desired horizontal and vertical spacing between adjacent components.

Note that applications should never call the `LayoutManager` methods of this class directly; the `Container` for which the `BorderLayout` is registered does this.

```
public class BorderLayout extends Object implements LayoutManager2, Serializable {
    // Public Constructors
            public BorderLayout();
            public BorderLayout(int hgap, int vgap);
    // Constants
        1.1 public static final String CENTER;
        1.1 public static final String EAST;
        1.1 public static final String NORTH;
        1.1 public static final String SOUTH;
        1.1 public static final String WEST;
    // Public Instance Methods
        1.1 public void addLayoutComponent(Component comp, Object constraints);  //
From LayoutManager2
        #   public void addLayoutComponent(String name, Component comp);  // From
LayoutManager
        1.1 public int getHgap();
        1.1 public float getLayoutAlignmentX(Container parent);  // From
LayoutManager2
        1.1 public float getLayoutAlignmentY(Container parent);  // From
LayoutManager2
        1.1 public int getVgap();
        1.1 public void invalidateLayout(Container target);  // From LayoutManager2
         public void layoutContainer(Container target);  // From LayoutManager
        1.1 public Dimension maximumLayoutSize(Container target);  // From
LayoutManager2
         public Dimension minimumLayoutSize(Container target);  // From LayoutManager
         public Dimension preferredLayoutSize(Container target);  // From
LayoutManager
         public void removeLayoutComponent(Component comp);  // From LayoutManager
        1.1 public void setHgap(int hgap);
        1.1 public void setVgap(int vgap);
         public String toString();  // Overrides Object
}
```

Hierarchy:

```
Object->BorderLayout(LayoutManager2(LayoutManager), Serializable)
```

PREVIOUS HOME NEXT

java.awt.Adjustable (JDK 1.1) BOOK INDEX java.awt.Button (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

18.9 java.awt.CardLayout (JDK 1.0)

This class is a `LayoutManager` that makes each of the components it manages as large as the container and ensures that only one is visible at a time. The standard `LayoutManager` methods are called by the `Container` object, and should not be called directly by applet or application code. `first()`, `last()`, `next()`, `previous()`, and `show()` make a particular `Component` in the `Container` visible. The names with which the components are added to the container are used only by the `show()` method.

```
public class CardLayout extends Object implements LayoutManager2, Serializable {
    // Public Constructors
            public CardLayout();
            public CardLayout(int hgap, int vgap);
    // Public Instance Methods
        1.1 public void addLayoutComponent(Component comp, Object constraints);  // 
From LayoutManager2
          # public void addLayoutComponent(String name, Component comp);  // From 
LayoutManager
            public void first(Container parent);
        1.1 public int getHgap();
        1.1 public float getLayoutAlignmentX(Container parent);  // From 
LayoutManager2
        1.1 public float getLayoutAlignmentY(Container parent);  // From 
LayoutManager2
        1.1 public int getVgap();
        1.1 public void invalidateLayout(Container target);  // From LayoutManager2
            public void last(Container parent);
            public void layoutContainer(Container parent);  // From LayoutManager
        1.1 public Dimension maximumLayoutSize(Container target);  // From 
LayoutManager2
            public Dimension minimumLayoutSize(Container parent);  // From 
LayoutManager
            public void next(Container parent);
            public Dimension preferredLayoutSize(Container parent);  // From 
LayoutManager
            public void previous(Container parent);
            public void removeLayoutComponent(Component comp);  // From LayoutManager
        1.1 public void setHgap(int hgap);
        1.1 public void setVgap(int vgap);
            public void show(Container parent, String name);
            public String toString();  // Overrides Object
}
```

Hierarchy:

```
Object->CardLayout(LayoutManager2(LayoutManager), Serializable)
```

PREVIOUS | HOME | NEXT
java.awt.Canvas (JDK 1.0) | BOOK INDEX | java.awt.Checkbox (JDK 1.0)

18.23 java.awt.FlowLayout (JDK 1.0)

This class implements the `LayoutManager` interface to lay out `Component` objects in a `Container`. `FlowLayout` arranges components in a container like words on a page: from left to right and top to bottom. It fits as many components as it can in a row before moving on to the next row. The constructor allows you to specify one of three constants as an alignment value for the rows, and also allows you to specify horizontal spacing between components and vertical spacing between rows.

Note that applications should never call the `LayoutManager` methods of this class directly; the `Container` for which the `FlowLayout` is registered does this.

```
public class FlowLayout extends Object implements LayoutManager, Serializable {
    // Public Constructors
            public FlowLayout();
            public FlowLayout(int align);
            public FlowLayout(int align, int hgap, int vgap);
    // Constants
            public static final int CENTER;
            public static final int LEFT;
            public static final int RIGHT;
    // Public Instance Methods
            public void addLayoutComponent(String name, Component comp);  // From 
LayoutManager
        1.1  public int getAlignment();
        1.1  public int getHgap();
        1.1  public int getVgap();
            public void layoutContainer(Container target);  // From LayoutManager
            public Dimension minimumLayoutSize(Container target);  // From 
LayoutManager
            public Dimension preferredLayoutSize(Container target);  // From 
LayoutManager
            public void removeLayoutComponent(Component comp);  // From LayoutManager
        1.1  public void setAlignment(int align);
        1.1  public void setHgap(int hgap);
        1.1  public void setVgap(int vgap);
            public String toString();  // Overrides Object
}
```

18.29 java.awt.GridBagLayout (JDK 1.0)

This class implements the `LayoutManager` interface to lay out `Component` objects in a `Container`. It is the most complicated and most powerful `LayoutManager` in the `java.awt` package. It divides the container into a grid of rows and columns (which need not have the same width and height) and places the components into this grid, adjusting the size of the grid cells as necessary to ensure that components do not overlap. Each component controls how it is positioned within this grid by specifying a number of variables (or "constraints") in a `GridBagConstraints` object. Do not confuse this class with the much simpler `GridLayout` which arranges components in a grid of equally sized cells.

Use `setConstraints()` to specify a `GridBagConstraints` object for each of the components in the container. Or, in Java 1.1, specify the `GridBagConstraints` object when adding the component to the container with `add()`. The variables in this object specify the position of the component in the grid, the number of horizontal and vertical grid cells that the component occupies, and also control other important aspects of component layout. See `GridBagConstraints` for more information on these "constraint" variables. `setConstraints()` makes a copy of the constraints object, so you may reuse a single object in your code.

Note that applications should never call the `LayoutManager` methods of this class directly; the `Container` for which the `GridBagLayout` is registered does this.

```
public class GridBagLayout extends Object implements LayoutManager2, Serializable {
    // Public Constructor
            public GridBagLayout();
    // Constants
            protected static final int MAXGRIDSIZE;
            protected static final int MINSIZE;
            protected static final int PREFERREDSIZE;
    // Public Instance Variables
            public double[] columnWeights;
            public int[] columnWidths;
            public int[] rowHeights;
            public double[] rowWeights;
    // Protected Instance Variables
            protected Hashtable comptable;
            protected GridBagConstraints defaultConstraints;
            protected GridBagLayoutInfo layoutInfo;
    // Public Instance Methods
            public void addLayoutComponent(String name, Component comp);  // From
LayoutManager
        1.1  public void addLayoutComponent(Component comp, Object constraints);  //
From LayoutManager2
            public GridBagConstraints getConstraints(Component comp);
        1.1  public float getLayoutAlignmentX(Container parent);  // From
LayoutManager2
        1.1  public float getLayoutAlignmentY(Container parent);  // From
LayoutManager2
            public int[][] getLayoutDimensions();
            public Point getLayoutOrigin();
            public double[][] getLayoutWeights();
        1.1  public void invalidateLayout(Container target);  // From LayoutManager2
            public void layoutContainer(Container parent);  // From LayoutManager
            public Point location(int x, int y);
        1.1  public Dimension maximumLayoutSize(Container target);  // From
LayoutManager2
            public Dimension minimumLayoutSize(Container parent);  // From
LayoutManager
            public Dimension preferredLayoutSize(Container parent);  // From
LayoutManager
            public void removeLayoutComponent(Component comp);  // From LayoutManager
            public void setConstraints(Component comp, GridBagConstraints
constraints);
            public String toString();  // Overrides Object
    // Protected Instance Methods
            protected void AdjustForGravity(GridBagConstraints constraints, Rectangle
r);
            protected void ArrangeGrid(Container parent);
            protected GridBagLayoutInfo GetLayoutInfo(Container parent, int
sizeflag);
            protected Dimension GetMinSize(Container parent, GridBagLayoutInfo info);
            protected GridBagConstraints lookupConstraints(Component comp);
}
```

Hierarchy:

```
Object->GridBagLayout(LayoutManager2(LayoutManager), Serializable)
```

PREVIOUS | HOME | NEXT
java.awt.GridBagConstraints (JDK 1.0) | BOOK INDEX | java.awt.GridLayout (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

18.30 java.awt.GridLayout (JDK 1.0)

This class implements the `LayoutManager` interface to lay out `Component` objects in a `Container`. It divides the `Container` into a specified number of rows and columns and arranges the components in those rows and columns, left-to-right and top-to-bottom. If either the number of rows or the number of columns is set to zero, its value is computed from the other dimension and the total number of components. Do not confuse this class with the more flexible and complicated `GridBagLayout`.

Note that applications should never call the `LayoutManager` methods of this class directly; the `Container` for which the `GridLayout` is registered does this.

```
public class GridLayout extends Object implements LayoutManager, Serializable {
    // Public Constructors
        1.1  public GridLayout();
            public GridLayout(int rows, int cols);
            public GridLayout(int rows, int cols, int hgap, int vgap);
    // Public Instance Methods
            public void addLayoutComponent(String name, Component comp);  // From 
LayoutManager
        1.1  public int getColumns();
        1.1  public int getHgap();
        1.1  public int getRows();
        1.1  public int getVgap();
            public void layoutContainer(Container parent);  // From LayoutManager
            public Dimension minimumLayoutSize(Container parent);  // From 
LayoutManager
            public Dimension preferredLayoutSize(Container parent);  // From 
LayoutManager
            public void removeLayoutComponent(Component comp);  // From LayoutManager
        1.1  public void setColumns(int cols);
        1.1  public void setHgap(int hgap);
        1.1  public void setRows(int rows);
        1.1  public void setVgap(int vgap);
            public String toString();  // Overrides Object
}
```

18.36 java.awt.LayoutManager (JDK 1.0)

This interface defines the methods necessary for a class to be able to arrange `Component` objects within a `Container` object. Most programs use one of the existing classes that implements this interface: `BorderLayout`, `CardLayout`, `FlowLayout`, `GridBagConstraints`, `GridBagLayout`, or `GridLayout`.

To define a new class that lays out components, you must implement each of the methods defined by this interface. `addLayoutComponent()` is called when a component is added to the container. `removeLayoutComponent()` is called when a component is removed. `layoutContainer()` should perform the actual positioning of components by setting the size and position of each component in the specified container. `minimumLayoutSize()` should return the minimum container width and height that the `LayoutManager` needs in order to lay out its components. `preferredLayoutSize()` should return the optimal container width and height for the `LayoutManager` to lay out its components.

In Java 1.1, layout managers should implement the `LayoutManager2` interface, which is an extension of this one. Note that a Java applet or application never directly calls any of these `LayoutManager` methods--the `Container` object for which the `LayoutManager` is registered does that.

```
public abstract interface LayoutManager {
    // Public Instance Methods
            public abstract void addLayoutComponent(String name, Component comp);
            public abstract void layoutContainer(Container parent);
            public abstract Dimension minimumLayoutSize(Container parent);
            public abstract Dimension preferredLayoutSize(Container parent);
            public abstract void removeLayoutComponent(Component comp);
}
```

Extended By:

`LayoutManager2`

Implemented By:

`FlowLayout, GridLayout`

Passed To:

`Container.setLayout(), Panel(), ScrollPane.setLayout()`

Returned By:

`Container.getLayout()`

PREVIOUS | HOME | NEXT
java.awt.Label (JDK 1.0) | BOOK INDEX | java.awt.LayoutManager2 (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

18.37 java.awt.LayoutManager2 (JDK 1.1)

This interface is an extension of the `LayoutManager` interface. It defines additional layout management methods for layout managers that perform constraint-based layout. `GridBagLayout` is an example of a constraint-based layout manager--each component added to the layout is associated with a `GridBagConstraints` object that specifies the "constraints" on how the component is to be laid out.

Java programs do not directly invoke the methods of this interface--they are used by the `Container` object for which the layout manager is registered.

```
public abstract interface LayoutManager2 extends LayoutManager {
    // Public Instance Methods
            public abstract void addLayoutComponent(Component comp, Object 
constraints);
            public abstract float getLayoutAlignmentX(Container target);
            public abstract float getLayoutAlignmentY(Container target);
            public abstract void invalidateLayout(Container target);
            public abstract Dimension maximumLayoutSize(Container target);
}
```

Implemented By:

BorderLayout, CardLayout, GridBagLayout

18.8 java.awt.Canvas (JDK 1.0)

This class is a `Component` that does no default drawing or event handling on its own. You can subclass it to display any kind of drawing or image, and to handle any kind of user input event. `Canvas` inherits the event handling methods of its superclass. In Java 1.1, you can also subclass `Component` directly to create a "lightweight component," instead of having to subclass `Canvas`.

```
public class Canvas extends Component {
    // Public Constructor
        1.1 public Canvas();
    // Public Instance Methods
            public void addNotify();  // Overrides Component
            public void paint(Graphics g);  // Overrides Component
}
```

Hierarchy:

```
Object->Component(ImageObserver, MenuContainer, Serializable)->Canvas
```

Passed To:

```
Toolkit.createCanvas()
```

18.18 java.awt.Dialog (JDK 1.0)

This class represents a dialog box window. A `Dialog` may be modal so that it blocks user input to all other windows until dismissed, may optionally have a title, and may be resizable. A `Dialog` object is a `Container` and `Component` objects can be added to it in the normal way with the `add()` method. The default `LayoutManager` for `Dialog` is `BorderLayout`. You may specify a different `LayoutManager` object with `setLayout()`. Call the `pack()` method of `Window` to initiate layout management of the dialog and set its initial size appropriately. Call `show()` to pop a dialog up, and `hide()` to pop it down. For modal dialogs, `show()` blocks until the dialog is dismissed. Event handling continues while `show()` is blocked, using a new event dispatcher thread. In Java 1.0, `show()` is inherited from `Window`. Call the `Window.dispose()` method when the `Dialog` is no longer needed so that its window system resources may be reused.

```
public class Dialog extends Window {
    // Public Constructors
        1.1  public Dialog(Frame parent);
            public Dialog(Frame parent, boolean modal);
        1.1  public Dialog(Frame parent, String title);
            public Dialog(Frame parent, String title, boolean modal);
    // Public Instance Methods
            public void addNotify();  // Overrides Window
            public String getTitle();
            public boolean isModal();
            public boolean isResizable();
        1.1  public void setModal(boolean b);
            public synchronized void setResizable(boolean resizable);
            public synchronized void setTitle(String title);
        1.1  public void show();  // Overrides Window
    // Protected Instance Methods
            protected String paramString();  // Overrides Container
}
```

Hierarchy:

```
Object->Component(ImageObserver, MenuContainer, Serializable)->
Container->Window->Dialog
```

Extended By:

```
FileDialog
```

Passed To:

```
Toolkit.createDialog()
```

PREVIOUS | HOME | NEXT
java.awt.Cursor (JDK 1.1) | BOOK INDEX | java.awt.Dimension (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

18.22 java.awt.FileDialog (JDK 1.0)

This class represents a file selection dialog box. The constants `LOAD` and `SAVE` are values of an optional constructor argument that specifies whether the dialog should be an **Open File** dialog or a **Save As** dialog. You may specify a `FilenameFilter` object to control which files are displayed in the dialog.

The inherited `show()` method pops the dialog up. For dialogs of this type, `show()` blocks, not returning until the user has selected a file and dismissed the dialog (which pops down automatically--you don't have to call `hide()`). Once `show()` has returned, use `getFile()` to get the name of the file the user selected.

```
public class FileDialog extends Dialog {
    // Public Constructors
        1.1  public FileDialog(Frame parent);
            public FileDialog(Frame parent, String title);
            public FileDialog(Frame parent, String title, int mode);
    // Constants
            public static final int LOAD;
            public static final int SAVE;
    // Public Instance Methods
            public void addNotify();  // Overrides Dialog
            public String getDirectory();
            public String getFile();
            public FilenameFilter getFilenameFilter();
            public int getMode();
            public synchronized void setDirectory(String dir);
            public synchronized void setFile(String file);
            public synchronized void setFilenameFilter(FilenameFilter filter);
        1.1  public void setMode(int mode);
    // Protected Instance Methods
            protected String paramString();  // Overrides Dialog
}
```

Hierarchy:

```
Object->Component(ImageObserver, MenuContainer, Serializable)-> Container-
>Window->Dialog->FileDialog
```

Passed To:

`Toolkit.createFileDialog()`

18.26 java.awt.Frame (JDK 1.0)

This class represents an optionally resizable top-level application window with a titlebar and other platform-dependent window decorations. `setTitle()` specifies a title, `setMenuBar()` specifies a menu bar, `setCursor()` specifies a cursor, and `setIconImage()` specifies an icon for the window. Call the `pack()` method of `Window` to initiate layout management of the window and set its initial size appropriately. Call the `show()` method of `Window` to make a frame appear on the screen or to bring it to the front of the window stack. Call `hide()` to remove a frame from the screen. Call the `dispose()` method when the `Frame` is no longer needed so that it can release its window system resources for reuse.

The constants defined by this class specify various cursor types. In Java 1.1, these constants and the cursor methods of `Frame` are deprecated in favor of the `Cursor` class and cursor methods of `Component`.

```
public class Frame extends Window implements MenuContainer {
    // Public Constructors
            public Frame();
            public Frame(String title);
    // Constants
            public static final int CROSSHAIR_CURSOR;
            public static final int DEFAULT_CURSOR;
            public static final int E_RESIZE_CURSOR;
            public static final int HAND_CURSOR;
            public static final int MOVE_CURSOR;
            public static final int NE_RESIZE_CURSOR;
            public static final int NW_RESIZE_CURSOR;
            public static final int N_RESIZE_CURSOR;
            public static final int SE_RESIZE_CURSOR;
            public static final int SW_RESIZE_CURSOR;
            public static final int S_RESIZE_CURSOR;
            public static final int TEXT_CURSOR;
            public static final int WAIT_CURSOR;
            public static final int W_RESIZE_CURSOR;
    // Public Instance Methods
            public void addNotify();  // Overrides Window
            public synchronized void dispose();  // Overrides Window
        #   public int getCursorType();
            public Image getIconImage();
            public MenuBar getMenuBar();
            public String getTitle();
            public boolean isResizable();
            public synchronized void remove(MenuComponent m);  // Overrides Component
        #   public synchronized void setCursor(int cursorType);
```

```
            public synchronized void setIconImage(Image image);
            public synchronized void setMenuBar(MenuBar mb);
            public synchronized void setResizable(boolean resizable);
            public synchronized void setTitle(String title);
    // Protected Instance Methods
            protected String paramString();  // Overrides Container
}
```

Hierarchy:

`Object->Component(ImageObserver, MenuContainer, Serializable)->Container->Window->Frame(MenuContainer)`

Passed To:

`Dialog(), FileDialog(), Toolkit.createFrame(), Toolkit.getPrintJob(), Window()`

PREVIOUS
java.awt.FontMetrics (JDK 1.0)

HOME
BOOK INDEX

NEXT
java.awt.Graphics (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

18.35 java.awt.Label (JDK 1.0)

This class is a `Component` that displays a single specified line of read-only text. The constant values specify the text alignment within the component and may be specified to the constructor or to `setAlignment()`.

```
public class Label extends Component {
    // Public Constructors
            public Label();
            public Label(String text);
            public Label(String text, int alignment);
    // Constants
            public static final int CENTER;
            public static final int LEFT;
            public static final int RIGHT;
    // Public Instance Methods
            public void addNotify();  // Overrides Component
            public int getAlignment();
            public String getText();
            public synchronized void setAlignment(int alignment);
            public synchronized void setText(String text);
    // Protected Instance Methods
            protected String paramString();  // Overrides Component
}
```

Hierarchy:

```
Object->Component(ImageObserver, MenuContainer, Serializable)->Label
```

Passed To:

```
Toolkit.createLabel()
```

18.46 java.awt.Panel (JDK 1.0)

This class is a Container that is itself contained within a container. Unlike Frame and Dialog, Panel is a container that does not create a separate window of its own. Panel is suitable for holding portions of a larger interface within a parent Frame or Dialog or within another Panel. (Note that Applet is a subclass of Panel, and thus applets are displayed in a Panel that is contained within a Web browser or applet viewer.) The default LayoutManager for a Panel is FlowLayout.

```
public class Panel extends Container {
    // Public Constructors
            public Panel();
        1.1  public Panel(LayoutManager layout);
    // Public Instance Methods
            public void addNotify();  // Overrides Container
}
```

Hierarchy:

```
Object->Component(ImageObserver, MenuContainer, Serializable)->Container->Panel
```

Extended By:

Applet

Passed To:

Toolkit.createPanel()

PREVIOUS
java.awt.MenuShortcut (JDK 1.1)

HOME
BOOK INDEX

NEXT
java.awt.Point (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

18.49 java.awt.PopupMenu (JDK 1.1)

`PopupMenu` is a simple subclass of `Menu` that represents a popup menu rather than a pulldown menu. You create a `PopupMenu` just as you would create a `Menu` object. The main difference is that a popup menu must be "popped up" in response to a user event by calling its `show()` method.

Another difference is that, unlike a `Menu`, which can only appear within a `MenuBar` or another `Menu`, a `PopupMenu` can be associated with any component in a graphical user interface. A `PopupMenu` is associated with a component by calling the `add()` method of the component.

Popup menus are popped up by the user in different ways on different platforms. In order to hide this platform-dependency, the `MouseEvent` class defines the `isPopupTrigger()` method. If this method returns `true`, the specified `MouseEvent` represents the platform-specific popup menu trigger event, and you should use the `show()` method to pop your `PopupMenu` up. Note that the X and Y coordinates passed to `show()` should be in the coordinate system of the specified `Component`.

```
public class PopupMenu extends Menu {
    // Public Constructors
            public PopupMenu();
            public PopupMenu(String label);
    // Public Instance Methods
            public synchronized void addNotify();  // Overrides Menu
            public void show(Component origin, int x, int y);
}
```

Hierarchy:

`Object->MenuComponent(Serializable)->MenuItem->Menu(MenuContainer)->PopupMenu`

Passed To:

`Component.add(), Toolkit.createPopupMenu()`

PREVIOUS | HOME | NEXT
java.awt.Polygon (JDK 1.0) | BOOK INDEX | java.awt.PrintGraphics (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

18.57 java.awt.TextArea (JDK 1.0)

This class is a GUI component that displays and optionally edits multi-line text. The `appendText()`, `insertText()`, and `replaceText()` provide various techniques for specifying text to appear in the `TextArea`. Many important `TextArea` methods are defined by the `TextComponent` superclass.

See also `TextComponent` and `TextField`.

```
public class TextArea extends TextComponent {
    // Public Constructors
            public TextArea();
            public TextArea(String text);
            public TextArea(int rows, int columns);
            public TextArea(String text, int rows, int columns);
        1.1  public TextArea(String text, int rows, int columns, int scrollbars);
    // Constants
        1.1  public static final int SCROLLBARS_BOTH;
        1.1  public static final int SCROLLBARS_HORIZONTAL_ONLY;
        1.1  public static final int SCROLLBARS_NONE;
        1.1  public static final int SCROLLBARS_VERTICAL_ONLY;
    // Public Instance Methods
            public void addNotify();  // Overrides Component
        1.1  public synchronized void append(String str);
        #   public void appendText(String str);
            public int getColumns();
        1.1  public Dimension getMinimumSize(int rows, int columns);
        1.1  public Dimension getMinimumSize();  // Overrides Component
        1.1  public Dimension getPreferredSize(int rows, int columns);
        1.1  public Dimension getPreferredSize();  // Overrides Component
            public int getRows();
        1.1  public int getScrollbarVisibility();
        1.1  public synchronized void insert(String str, int pos);
        #   public void insertText(String str, int pos);
        #   public Dimension minimumSize(int rows, int columns);
        #   public Dimension minimumSize();  // Overrides Component
        #   public Dimension preferredSize(int rows, int columns);
        #   public Dimension preferredSize();  // Overrides Component
        1.1  public synchronized void replaceRange(String str, int start, int end);
        #   public void replaceText(String str, int start, int end);
        1.1  public void setColumns(int columns);
        1.1  public void setRows(int rows);
    // Protected Instance Methods
            protected String paramString();  // Overrides TextComponent
}
```

Hierarchy:

`Object->Component(ImageObserver, MenuContainer, Serializable)->TextComponent->TextArea`

Passed To:

`Toolkit.createTextArea()`

18.61 java.awt.Window (JDK 1.0)

This class represents a top-level window with no borders or menu bar. `Window` is a `Container` with `BorderLayout` as its default layout manager. `Window` is rarely used directly; its subclasses `Frame` and `Dialog` are more commonly useful.

`show()` (which overrides `Component.show()`) makes a `Window` visible and brings it to the front of other windows. `toFront()` brings a window to the front, and `toBack()` buries a window beneath others. `pack()` is an important method that initiates layout management for the window, and sets the window size to match the preferred size of the components contained within the window. `getToolkit()` returns the `Toolkit()` in use for this window. Call `dispose()` when a `Window` is no longer needed to free its window system resources.

```
public class Window extends Container {
    // Public Constructor
            public Window(Frame parent);
    // Public Instance Methods
            public void addNotify();  // Overrides Container
        1.1  public synchronized void addWindowListener(WindowListener l);
            public void dispose();
        1.1  public Component getFocusOwner();
        1.1  public Locale getLocale();  // Overrides Component
            public Toolkit getToolkit();  // Overrides Component
            public final String getWarningString();
        1.1  public boolean isShowing();  // Overrides Component
            public void pack();
        1.1  public boolean postEvent(Event e);  // Overrides Component
        1.1  public synchronized void removeWindowListener(WindowListener l);
            public void show();  // Overrides Component
            public void toBack();
            public void toFront();
    // Protected Instance Methods
        1.1  protected void processEvent(AWTEvent e);  // Overrides Container
        1.1  protected void processWindowEvent(WindowEvent e);
}
```

Hierarchy:

Object->Component(ImageObserver, MenuContainer, Serializable)->Container->Window

Extended By:

Dialog, Frame

Passed To:

Toolkit.createWindow(), WindowEvent()

Returned By:

WindowEvent.getWindow()

PREVIOUS | HOME | NEXT
java.awt.Toolkit (JDK 1.0) | BOOK INDEX | The java.awt.datatransfer Package

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

30.15 java.util.Observable (JDK 1.0)

This class is the superclass of all "observable" objects to be used in an object-oriented model/view paradigm. The class methods allow you to add and delete `Observer` objects to and from an `Observable`'s list, and to notify all of the `Observer` objects on the list. `Observer` objects are "notified" by invoking their `update()` method. `Observable` also maintains an internal "changed" flag, which may be set and cleared by the `Observable`, and which may be queried with `hasChanged()` by any interested observer.

```
public class Observable extends Object {
    // Public Constructor
        1.1public Observable();
    // Public Instance Methods
            public synchronized void addObserver(Observer o);
            public synchronized int countObservers();
            public synchronized void deleteObserver(Observer o);
            public synchronized void deleteObservers();
            public synchronized boolean hasChanged();
            public void notifyObservers();
            public void notifyObservers(Object arg);
    // Protected Instance Methods
            protected synchronized void clearChanged();
            protected synchronized void setChanged();
}
```

Passed To:

Observer.update()

java.util.NoSuchElementException (JDK 1.0) BOOK INDEX java.util.Observer (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

18.48 java.awt.Polygon (JDK 1.0)

This class defines a polygon as an array of points. The points of the polygon may be specified by the constructor, or with the `addPoint()` method. `getBoundingBox()` returns the smallest `Rectangle` that contains the polygon, and `inside()` tests whether a specified point is within the `Polygon`. Note that the arrays of X and Y points and the number of points in the polygon (not necessarily the same as the array size) are defined as public variables. `Polygon` objects are used when drawing polygons with the `Graphics.drawPolygon()` and `Graphics.fillPolygon()` methods.

```
public class Polygon extends Object implements Shape, Serializable {
    // Public Constructors
            public Polygon();
            public Polygon(int[] xpoints, int[] ypoints, int npoints);
    // Public Instance Variables
            public int npoints;
            public int[] xpoints;
            public int[] ypoints;
    // Protected Instance Variables
        1.1  protected Rectangle bounds;
    // Public Instance Methods
            public void addPoint(int x, int y);
        1.1  public boolean contains(Point p);
        1.1  public boolean contains(int x, int y);
        #   public Rectangle getBoundingBox();
        1.1  public Rectangle getBounds();  // From Shape
        #   public boolean inside(int x, int y);
        1.1  public void translate(int deltaX, int deltaY);
}
```

Passed To:

```
Graphics.drawPolygon(), Graphics.fillPolygon()
```

PREVIOUS | HOME | NEXT
java.awt.Point (JDK 1.0) | BOOK INDEX | java.awt.PopupMenu (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

23.4 java.beans.Customizer (JDK 1.1)

The `Customizer` interface specifies the methods that must be defined by any class designed to customize a Java bean. In addition to implementing this interface, a customizer class must be a subclass of `java.awt.Component`, and it must have a constructor that takes no arguments, so that it can be instantiated by an application builder.

Customizer classes are typically used by a complex Java bean to allow the user to easily configure the bean, and to provide an alternative to a simple list of properties and their values. If a customizer class is defined for a Java bean, it must be associated with the bean through a `BeanDescriptor` object returned by a `BeanInfo` class for the bean. Note that while a `Customizer` class is created by the author of a bean, that class is only instantiated and used by application builders and similar tools.

After a `Customizer` class is instantiated, its `setObject()` method is invoked once to specify the bean object that it is to customize. The `addPropertyChangeListener()` and `removePropertyChangeListener()` methods may be called to register and de-register `PropertyChangeListener` objects. The `Customizer` should send a `PropertyChangeEvent` to all registered listeners any time it changes a property of the bean it is customizing.

```
public abstract interface Customizer {
    // Public Instance Methods
            public abstract void addPropertyChangeListener(PropertyChangeListener 
listener);
            public abstract void removePropertyChangeListener(PropertyChangeListener 
listener);
            public abstract void setObject(Object bean);
}
```

23.14 java.beans.PropertyChangeSupport (JDK 1.1)

The `PropertyChangeSupport` class is a convenience class that maintains a list of registered `PropertyChangeListener` objects and provides the `firePropertyChange()` method for sending a `PropertyChangeEvent` object to all registered listeners. Because there are some tricky thread synchronization issues involved in doing this correctly, it is recommended that all Java beans that support "bound" properties either extend this class, or, more commonly, create an instance of this class to which they can delegate the task of maintaining the list of listeners.

```
public class PropertyChangeSupport extends Object implements Serializable {
    // Public Constructor
            public PropertyChangeSupport(Object sourceBean);
    // Public Instance Methods
            public synchronized void addPropertyChangeListener(PropertyChangeListener 
listener);
            public void firePropertyChange(String propertyName, Object oldValue, 
Object newValue);
            public synchronized void 
removePropertyChangeListener(PropertyChangeListener listener);
}
```

23.16 java.beans.PropertyEditor (JDK 1.1)

The PropertyEditor interface defines the methods that must be implemented by a Java beans property editor intended for use within an application builder or similar tool. PropertyEditor is a complex interface because it defines methods to support several different ways of displaying property values to the user, and it also defines methods to support several different ways of allowing the user to edit the property value.

For a property of type *x*, the author of a Java bean typically implements a property editor of class *x*Editor. While the editor is implemented by the bean author, it is usually only instantiated or used by application builders or similar tools (or by a Customizer class for a Bean).

In addition to implementing the PropertyEditor interface, a property editor must have a constructor that expects no arguments, so that it can be easily instantiated by an application builder. Also, it must accept registration and deregistration of PropertyChangeListener objects, and it must send a PropertyChangeEvent to all registered listeners when it changes the value of the property being edited.

The PropertyEditorSupport class is a trivial implementation of PropertyEditor, suitable for subclassing, or for supporting a list of PropertyChangeListener objects.

```
public abstract interface PropertyEditor {
    // Public Instance Methods
            public abstract void addPropertyChangeListener(PropertyChangeListener
listener);
            public abstract String getAsText();
            public abstract Component getCustomEditor();
            public abstract String getJavaInitializationString();
            public abstract String[] getTags();
            public abstract Object getValue();
            public abstract boolean isPaintable();
            public abstract void paintValue(Graphics gfx, Rectangle box);
            public abstract void removePropertyChangeListener(PropertyChangeListener
listener);
            public abstract void setAsText(String text) throws
IllegalArgumentException;
            public abstract void setValue(Object value);
            public abstract boolean supportsCustomEditor();
}
```

Implemented By:

PropertyEditorSupport

Returned By:

PropertyEditorManager.findEditor()

23.18 java.beans.PropertyEditorSupport (JDK 1.1)

The `PropertyEditorSupport` class is a trivial implementation of the `PropertyEditor` interface. It provides "no-op" default implementations of most methods, so that you can can define simple `PropertyEditor` subclasses that only override a few required methods.

In addition, `PropertyEditorSupport` defines working versions of `addPropertyChangeListener()` and `removePropertyChangeListener()`, along with a `firePropertyChange()` method that sends a `PropertyChangeEvent` to all registered listeners. `PropertyEditor` classes may choose to instantiate a `PropertyEditorSupport` object simply to handle the job of managing the list of listeners. When used in this way, the `PropertyEditorSupport` object should be instantiated with a source object specified, so that that source object can be used in the `PropertyChangeEvent` objects that are sent.

```
public class PropertyEditorSupport extends Object implements PropertyEditor {
    // Protected Constructors
            protected PropertyEditorSupport();
            protected PropertyEditorSupport(Object source);
    // Public Instance Methods
            public synchronized void addPropertyChangeListener(PropertyChangeListener
            public synchronized void addPropertyChangeListener'u'listener);  // From
PropertyEditor
            public void firePropertyChange();
            public String getAsText();  // From PropertyEditor
            public Component getCustomEditor();  // From PropertyEditor
            public String getJavaInitializationString();  // From PropertyEditor
            public String[] getTags();  // From PropertyEditor
            public Object getValue();  // From PropertyEditor
            public boolean isPaintable();  // From PropertyEditor
            public void paintValue(Graphics gfx, Rectangle box);  // From
PropertyEditor
            public synchronized void
removePropertyChangeListener(PropertyChangeListener
            public synchronized void removePropertyChangeListener'u'listener);  //
From PropertyEditor
            public void setAsText(String text) throws IllegalArgumentException;  //
From PropertyEditor
            public void setValue(Object value);  // From PropertyEditor
            public boolean supportsCustomEditor();  // From PropertyEditor
}
```

18.58 java.awt.TextComponent (JDK 1.0)

This class is the superclass of the `TextArea` and `TextField` components. It cannot be instantiated itself, but provides methods that are common to these two component types. `setEditable()` specifies whether the text in the text component is editable or not. `getText()` returns the text in the component, and `setText()` specifies text to be displayed. `getSelectedText()` returns the currently selected text in the text component, and `getSelectionStart()` and `getSelectionEnd()` return the extents of the selected region of text. `select()` and `selectAll()` select some or all of the text displayed in the text component.

See also `TextField` and `TextArea`.

```
public class TextComponent extends Component {
    // No Constructor
    // Protected Instance Variables
        1.1  protected transient TextListener textListener;
    // Public Instance Methods
        1.1  public void addTextListener(TextListener l);
        1.1  public int getCaretPosition();
            public synchronized String getSelectedText();
            public synchronized int getSelectionEnd();
            public synchronized int getSelectionStart();
            public synchronized String getText();
            public boolean isEditable();
            public void removeNotify();  // Overrides Component
        1.1  public void removeTextListener(TextListener l);
            public synchronized void select(int selectionStart, int selectionEnd);
            public synchronized void selectAll();
        1.1  public void setCaretPosition(int position);
            public synchronized void setEditable(boolean b);
        1.1  public synchronized void setSelectionEnd(int selectionEnd);
        1.1  public synchronized void setSelectionStart(int selectionStart);
            public synchronized void setText(String t);
    // Protected Instance Methods
            protected String paramString();  // Overrides Component
        1.1  protected void processEvent(AWTEvent e);  // Overrides Component
        1.1  protected void processTextEvent(TextEvent e);
}
```

Hierarchy:

```
Object->Component(ImageObserver, MenuContainer, Serializable)->TextComponent
```

Extended By:

```
TextArea, TextField
```

PREVIOUS | HOME | NEXT
java.awt.TextArea (JDK 1.0) | BOOK INDEX | java.awt.TextField (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

23.22 java.beans.VetoableChangeSupport (JDK 1.1)

`VetoableChangeSupport` is a convenience class that maintains a list of registered `VetoableChangeListener` objects and provides a `fireVetoableChange()` method for sending a `PropertyChangeEvent` to all registered listeners. If any of the registered listeners veto the proposed change, `fireVetoableChange()` send out another `PropertyChangeEvent` notifying previously notified listeners that the property has changed back to its original value.

Because of the extra complexity of correctly handling vetoable changes, and because of some tricky thread synchronization issues involved in maintaining the list of listeners, it is recommended that all Java beans that support "constrained" events create a `VetoableChangeSupport` object to which they can delegate the tasks of maintaining the list of listeners and of firing events.

```
public class VetoableChangeSupport extends Object implements Serializable {
    // Public Constructor
            public VetoableChangeSupport(Object sourceBean);
    // Public Instance Methods
            public synchronized void addVetoableChangeListener(VetoableChangeListener 
listener);
            public void fireVetoableChange(String propertyName, Object oldValue, 
Object newValue)
            public void fireVetoableChange'u'throws PropertyVetoException;
            public synchronized void 
removeVetoableChangeListener(VetoableChangeListener listener);
}
```

20.3 java.awt.event.AdjustmentEvent (JDK 1.1)

An event of this type indicates that an adjustment has been made to an `Adjustable` object--usually, this means that the user has interacted with a `Scrollbar` component.

The `getValue()` method returns the new value of the `Adjustable` object. This is usually the most important piece of information stored in the event. `getAdjustable()` returns the `Adjustable` object that was the source of the event. It is a convenient alternative to the inherited `getSource()` method.

The `getID()` method returns the type of an `AdjustmentEvent`. The standard AWT components only generate adjustment events of type `AdjustmentEvent.ADJUSTMENT_VALUE_CHANGED`. There are several types of adjustments that can be made to an `Adjustable` object, however, and the `getAdjustmentType()` method returns one of five constants to indicate which type has occurred. `UNIT_INCREMENT` indicates that the `Adjustable` value has been incremented by one unit, as in a scroll-line-down operation. `UNIT_DECREMENT` indicates the opposite: scroll-line-up. `BLOCK_INCREMENT` and `BLOCK_DECREMENT` indicate that the `Adjustable` object has been incremented or decremented by multiple units, as in a scroll-page-down or scroll-page-up operation. Finally, the `TRACK` constant indicates that the `Adjustable` value has been set to an absolute value unrelated to its previous value, as when the user drags a scrollbar to a new position.

```
public class AdjustmentEvent extends AWTEvent {
    // Public Constructor
            public AdjustmentEvent(Adjustable source, int id, int type, int value);
    // Constants
            public static final int ADJUSTMENT_FIRST;
            public static final int ADJUSTMENT_LAST;
            public static final int ADJUSTMENT_VALUE_CHANGED;
            public static final int BLOCK_DECREMENT;
            public static final int BLOCK_INCREMENT;
            public static final int TRACK;
            public static final int UNIT_DECREMENT;
            public static final int UNIT_INCREMENT;
    // Public Instance Methods
            public Adjustable getAdjustable();
            public int getAdjustmentType();
            public int getValue();
            public String paramString();  // Overrides AWTEvent
}
```

Hierarchy:

```
Object->EventObject(Serializable)->AWTEvent->AdjustmentEvent
```

Passed To:

```
AdjustmentListener.adjustmentValueChanged(),
AWTEventMulticaster.adjustmentValueChanged(), Scrollbar.processAdjustmentEvent()
```

PREVIOUS | HOME | NEXT

java.awt.event.ActionListener (JDK 1.1) | BOOK INDEX | java.awt.event.AdjustmentListener (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

20.4 java.awt.event.AdjustmentListener (JDK 1.1)

This interface defines the method that an object must implement to "listen" for adjustment events on AWT components. When an `AdjustmentEvent` occurs, an AWT component notifies its registered `AdjustmentListener` objects by invoking their `adjustmentValueChanged()` methods.

```
public abstract interface AdjustmentListener extends EventListener {
    // Public Instance Methods
            public abstract void adjustmentValueChanged(AdjustmentEvent e);
}
```

Implemented By:

`AWTEventMulticaster`

Passed To:

`Adjustable.addAdjustmentListener(), Adjustable.removeAdjustmentListener(), AWTEventMulticaster.add(), AWTEventMulticaster.remove(), Scrollbar.addAdjustmentListener(), Scrollbar.removeAdjustmentListener()`

Returned By:

`AWTEventMulticaster.add(), AWTEventMulticaster.remove()`

31. The java.util.zip Package

Contents:

The `java.util.zip` package contains classes for data compression and decompression. It is new in Java 1.1. Figure 31.1 shows the class hierarchy of the package.

The `Deflater` and `Inflater` classes perform data compression and decompression. `DeflaterOutputStream` and `InflaterInputStream` apply that functionality to byte streams; the subclasses of these streams implement both the GZIP and ZIP compression formats. The `Adler32` and `CRC32` classes implement the `Checksum` interface and compute the checksums required for data compression.

Figure 31.1: The java.util.zip package

[Graphic: Figure 31-1]

31.1 java.util.zip.Adler32 (JDK 1.1)

This class implements the `Checksum` interface and computes a checksum on a stream of data using the Adler-32 algorithm. This algorithm is significantly faster than the CRC-32 algorithm and is almost as reliable.

The `CheckedInputStream` and `CheckedOutputStream` classes provide a higher-level interface to computing checksums on streams of data.

```
public class Adler32 extends Object implements Checksum {
    // Default Constructor: public Adler32()
    // Public Instance Methods
            public long getValue();  // From Checksum
            public void reset();  // From Checksum
            public void update(int b);  // From Checksum
            public native void update(byte[] b, int off, int len);  // From Checksum
            public void update(byte[] b);
}
```

PREVIOUS | HOME | NEXT
java.util.Vector (JDK 1.0) | BOOK INDEX | java.util.zip.CheckedInputStream (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

30.3 java.util.Date (JDK 1.0)

This class represents dates and times. It lets you work with them in a system-independent way. You can create a `Date` by specifying the number of milliseconds from the epoch (midnight GMT, January 1st, 1970), or by specifying the year, month, date, and optionally, the hour, minute, and second. Years are specified as the number of years since 1900. If you call the `Date` constructor with no arguments, the `Date` is initialized to the current time and date. The instance methods of the class allow you to get and set the various date and time fields, to compare dates and times, and to convert dates to and from string representations.

In Java 1.1, many of the date methods have been deprecated in favor of the methods of the `Calendar` class.

```
public class Date extends Object implements Serializable, Cloneable {
    // Public Constructors
            public Date();
            public Date(long date);
        #   public Date(int year, int month, int date);
        #   public Date(int year, int month, int date, int hrs, int min);
        #   public Date(int year, int month, int date, int hrs, int min, int sec);
        #   public Date(String s);
    // Class Methods
        #   public static long UTC(int year, int month, int date, int hrs, int min, 
int sec);
        #   public static long parse(String s);
    // Public Instance Methods
            public boolean after(Date when);
            public boolean before(Date when);
            public boolean equals(Object obj);  // Overrides Object
        #   public int getDate();
        #   public int getDay();
        #   public int getHours();
        #   public int getMinutes();
        #   public int getMonth();
        #   public int getSeconds();
            public long getTime();
        #   public int getTimezoneOffset();
        #   public int getYear();
            public int hashCode();  // Overrides Object
        #   public void setDate(int date);
        #   public void setHours(int hours);
        #   public void setMinutes(int minutes);
        #   public void setMonth(int month);
        #   public void setSeconds(int seconds);
            public void setTime(long time);
        #   public void setYear(int year);
        #   public String toGMTString();
        #   public String toLocaleString();
            public String toString();  // Overrides Object
}
```

Passed To:

Calendar.setTime(), Date.after(), Date.before(), DateFormat.format(), GregorianCalendar.setGregorianChange(), SimpleDateFormat.format(), SimpleTimeZone.inDaylightTime(), TimeZone.inDaylightTime()

Returned By:

Calendar.getTime(), DateFormat.parse(), GregorianCalendar.getGregorianChange(), SimpleDateFormat.parse()

PREVIOUS | HOME | NEXT
java.util.Calendar (JDK 1.1) | BOOK INDEX | java.util.Dictionary (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

20.14 java.awt.event.InputEvent (JDK 1.1)

This abstract class serves as the superclass for the raw user input event types `MouseEvent` and `KeyEvent`.

Use the inherited `getComponent()` to determine in which component the event occurred. Use `getWhen()` to obtain a timestamp for the event. Use `getModifiers()` to determine which keyboard modifier keys or mouse buttons were down when the event occurred. You can decode the `getModifiers()` return value using the various `_MASK` constants defined by this class. The class also defines four convenience methods for determining the state of keyboard modifiers.

In Java 1.1, input events are delivered to the appropriate listener objects before they are delivered to the AWT components themselves. If a listener calls the `consume()` method of the event, the event is not passed on to the component. For example, if a listener registered on a `Button` "consumes" a mouse click, it prevents the button itself from responding to that event. You can use `isConsumed()` to test whether some other listener object has already consumed the event.

```
public abstract class InputEvent extends ComponentEvent {
    // No Constructor
    // Constants
            public static final int ALT_MASK;
            public static final int BUTTON1_MASK;
            public static final int BUTTON2_MASK;
            public static final int BUTTON3_MASK;
            public static final int CTRL_MASK;
            public static final int META_MASK;
            public static final int SHIFT_MASK;
    // Public Instance Methods
            public void consume();  // Overrides AWTEvent
            public int getModifiers();
            public long getWhen();
            public boolean isAltDown();
            public boolean isConsumed();  // Overrides AWTEvent
            public boolean isControlDown();
            public boolean isMetaDown();
            public boolean isShiftDown();
}
```

Hierarchy:

```
Object->EventObject(Serializable)->AWTEvent->ComponentEvent-
>InputEvent
```

Extended By:

```
KeyEvent, MouseEvent
```

PREVIOUS | HOME | NEXT
java.awt.event.FocusListener (JDK 1.1) | BOOK INDEX | java.awt.event.ItemEvent (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

29.7 java.text.DateFormat (JDK 1.1)

This class formats and parses dates and times in a locale-specific way. As an abstract class, it cannot be instantiated directly, but it provides a number of static methods that return instances of a concrete subclass which you can use to format dates in a variety of ways. The `getDateInstance()` methods return a `DateFormat` object suitable for formatting dates in either the default locale or a specified locale. A formatting style may also optionally be specified--the constants `FULL`, `LONG`, `MEDIUM`, `SHORT`, and `DEFAULT` specify this style. Similarly, the `getTimeInstance()` methods return a `DateFormat` object that formats and parses times, and the `getDateTimeInstance()` methods return a `DateFormat` object that formats both dates and times. These methods also optionally take a format style constant and a `Locale`. Finally, `getInstance()` returns a default `DateFormat` object that formats both dates and times in the `SHORT` format.

Once you have created a `DateFormat` object, you can use the `setCalendar()` and `setTimeZone()` methods if you want to format the date using a calendar or time zone other than the default.

The various `format()` methods convert `java.util.Date` objects to strings, using whatever format is encapsulated in the `DateFormat` object. The `parse()` and `parseObject()` methods perform the reverse operation--they parse a string formatted according to the rules of the `DateFormat` object and convert it into a `Date` object.

The `DEFAULT`, `FULL`, `MEDIUM`, `LONG`, and `SHORT` constants are used to specify how verbose or compact the formatted date or time should be. The remaining constants, which all end with `_FIELD`, specify various fields of formatted dates and times and are used with the `FieldPosition` object that is optionally passed to `format()`.

```
public abstract class DateFormat extends Format implements Cloneable {
    // Protected Constructor
            protected DateFormat();
    // Format Style Constants
            public static final int DEFAULT;
            public static final int FULL;
            public static final int LONG;
            public static final int MEDIUM;
            public static final int SHORT;
    // Date and Time Field Constants
            public static final int ERA_FIELD;
            public static final int YEAR_FIELD;
            public static final int MONTH_FIELD;
            public static final int WEEK_OF_MONTH_FIELD, WEEK_OF_YEAR_FIELD;
            public static final int DATE_FIELD, DAY_OF_YEAR_FIELD;
            public static final int DAY_OF_WEEK_FIELD, DAY_OF_WEEK_IN_MONTH_FIELD;
            public static final int TIMEZONE_FIELD;
            public static final int AM_PM_FIELD;
            public static final int HOUR0_FIELD, HOUR1_FIELD;
            public static final int HOUR_OF_DAY0_FIELD, HOUR_OF_DAY1_FIELD;
            public static final int MINUTE_FIELD;
            public static final int SECOND_FIELD;
            public static final int MILLISECOND_FIELD;
    // Protected Instance Variables
            protected Calendar calendar;
            protected NumberFormat numberFormat;
    // Class Methods
            public static Locale[] getAvailableLocales();
            public static final DateFormat getDateInstance();
            public static final DateFormat getDateInstance(int style);
            public static final DateFormat getDateInstance(int style, Locale
aLocale);
            public static final DateFormat getDateTimeInstance();
            public static final DateFormat getDateTimeInstance(int dateStyle, int
timeStyle);
            public static final DateFormat getDateTimeInstance(int dateStyle, int
timeStyle, Locale aLocale);
            public static final DateFormat getInstance();
            public static final DateFormat getTimeInstance();
            public static final DateFormat getTimeInstance(int style);
            public static final DateFormat getTimeInstance(int style, Locale
aLocale);
    // Public Instance Methods
            public Object clone();  // Overrides Format
            public boolean equals(Object obj);  // Overrides Object
            public final StringBuffer format(Object obj, StringBuffer toAppendTo,
FieldPosition fieldPosition);  // Defines Format
            public abstract StringBuffer format(Date date, StringBuffer toAppendTo,
FieldPosition fieldPosition);
            public final String format(Date date);
            public Calendar getCalendar();
            public NumberFormat getNumberFormat();
            public TimeZone getTimeZone();
            public int hashCode();  // Overrides Object
            public boolean isLenient();
            public Date parse(String text) throws ParseException;
            public abstract Date parse(String text, ParsePosition pos);
            public Object parseObject(String source, ParsePosition pos);  // Defines
Format
            public void setCalendar(Calendar newCalendar);
            public void setLenient(boolean lenient);
            public void setNumberFormat(NumberFormat newNumberFormat);
            public void setTimeZone(TimeZone zone);
}
```

Hierarchy:

Object->Format(Serializable, Cloneable)->DateFormat(Cloneable)

Extended By:

SimpleDateFormat

Returned By:

DateFormat.getDateInstance(), DateFormat.getDateTimeInstance(), DateFormat.getInstance(), DateFormat.getTimeInstance()

18.28 java.awt.GridBagConstraints (JDK 1.0)

This class encapsulates the instance variables that tell a `GridBagLayout` how to position a given `Component` within its `Container`.

`gridx`, `gridy`

> These fields specify the grid position of the component. The `RELATIVE` constant specifies a position to the right or below the previous component.

`gridwidth`, `gridheight`

> These fields specify the height and width of the component in grid cells. The constant `REMAINDER` specifies that the component is the last one and should get all remaining cells.

`fill`

> This field specifies which dimensions of a component should grow when the space available for it is larger than its default size. Legal values are the constants `NONE`, `BOTH`, `HORIZONTAL`, and `VERTICAL`.

`ipadx`, `ipady`

> These fields specify internal padding to add on each side of the component in each dimension. They increase the size of the component beyond its default minimum size.

`insets`

> This `Insets` object specifies margins to appear on all sides of the component.

`anchor`

> This field specifies how the component should be displayed within its grid cells when it is smaller than those cells. The `CENTER` constant and the compass-point constants are legal values.

`weightx`, `weighty`

> These fields specify how extra space in the container should be distributed among its components in the X and Y dimensions. Larger weights specify that a component should receive a proportionally larger amount of extra space. A

zero weight specifies that the component should not receive any extra space. These weights specify the resizing behavior of the component and its container.

See also `GridBagLayout`.

```
public class GridBagConstraints extends Object implements Cloneable, Serializable {
    // Public Constructor
            public GridBagConstraints();
    // Constants
            public static final int BOTH;
            public static final int CENTER;
            public static final int EAST;
            public static final int HORIZONTAL;
            public static final int NONE;
            public static final int NORTH;
            public static final int NORTHEAST;
            public static final int NORTHWEST;
            public static final int RELATIVE;
            public static final int REMAINDER;
            public static final int SOUTH;
            public static final int SOUTHEAST;
            public static final int SOUTHWEST;
            public static final int VERTICAL;
            public static final int WEST;
    // Public Instance Variables
            public int anchor;
            public int fill;
            public int gridheight;
            public int gridwidth;
            public int gridx;
            public int gridy;
            public Insets insets;
            public int ipadx;
            public int ipady;
            public double weightx;
            public double weighty;
    // Public Instance Methods
            public Object clone();  // Overrides Object
}
```

Passed To:

`GridBagLayout.AdjustForGravity(), GridBagLayout.setConstraints()`

Returned By:

`GridBagLayout.getConstraints(), GridBagLayout.lookupConstraints()`

Type Of:

```
GridBagLayout.defaultConstraints
```

PREVIOUS | HOME | NEXT
java.awt.Graphics (JDK 1.0) | BOOK INDEX | java.awt.GridBagLayout (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

30. The java.util Package

Contents:

The `java.util` package defines a number of useful classes. This package should not be considered a "utility" package separate from the rest of the language; in fact, Java depends directly on several of the classes in this package. Figure 30.1 shows the class hierarchy of this package.

Figure 30.1: The java.util package

[Graphic: Figure 30-1]

The `Hashtable` class is one of the most useful in the package--it implements a hashtable or associative array. It allows arbitrary objects to be stored and retrieved by arbitrary keys. The `Properties` subclass of `Hashtable` is used to store the Java system properties list.

`Vector` is another extremely useful class. It implements an array of objects that grows as needed when objects are added.

The `Enumeration` interface provides a simple and consistent way to loop through all the elements contained within some kind of object or data structure.

The Date class represents a date and time, using a millisecond representation. In Java 1.1, the Calendar class manipulates dates using more familiar units such as months, days, hours, and minutes. The TimeZone class is also used in conjunction with dates.

In Java 1.1, ResourceBundle and its subclasses, ListResourceBundle and PropertyResourceBundle, represent a "bundle" of localized resources that are read in by an internationalized program at runtime.

The remaining classes are also useful. BitSet implements an arbitrary-size array of bits. Random generates and returns pseudo-random numbers in a variety of forms. StringTokenizer parses a string into tokens. Stack implements a last-in-first-out stack on which objects may be pushed, and from which they may be popped. And the Observer interface and Observable class provide infrastructure for implementing the object-oriented model-view paradigm in Java.

30.1 java.util.BitSet (JDK 1.0)

This class defines an arbitrarily large set of bits. Instance methods allow you to set, clear, and query individual bits in the set, and also to perform bitwise boolean arithmetic on the bits in BitSet objects. This class can be used as an extremely compact array of boolean values, although reading and writing those values is slower than normal array access.

```
public final class BitSet extends Object implements Cloneable, Serializable {
    // Public Constructors
            public BitSet();
            public BitSet(int nbits);
    // Public Instance Methods
            public void and(BitSet set);
            public void clear(int bit);
            public Object clone();  // Overrides Object
            public boolean equals(Object obj);  // Overrides Object
            public boolean get(int bit);
            public int hashCode();  // Overrides Object
            public void or(BitSet set);
            public void set(int bit);
            public int size();
            public String toString();  // Overrides Object
            public void xor(BitSet set);
}
```

Passed To:

BitSet.and(), BitSet.or(), BitSet.xor()

PREVIOUS HOME NEXT

java.text.StringCharacterIterator (JDK 1.1) BOOK INDEX java.util.Calendar (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

24.42 java.io.ObjectOutputStream (JDK 1.1)

The `ObjectOutputStream` is used to serialize objects, arrays, and other values to a stream. The `writeObject()` method serializes an object or array, and various other methods are used to write primitive data values to the stream. Note that only objects that implement the `Serializable` interface or the `Externalizable` interface can be serialized.

The `defaultWriteObject()` may only be called from the `writeObject()` method of a `Serializable` object. It allows an object to perform additional processing before or after serializing itself.

The remaining methods of `ObjectOutputStream` are miscellaneous stream manipulation methods and protected methods for use by subclasses that want to customize its serialization behavior.

```
public class ObjectOutputStream extends OutputStream implements ObjectOutput {
    // Public Constructor
            public ObjectOutputStream(OutputStream out) throws IOException;
    // Public Instance Methods
            public void close() throws IOException;  // Overrides OutputStream
            public final void defaultWriteObject() throws IOException;
            public void flush() throws IOException;  // Overrides OutputStream
            public void reset() throws IOException;
            public void write(int data) throws IOException;  // Defines OutputStream
            public void write(byte[] b) throws IOException;  // Overrides 
OutputStream
            public void write(byte[] b, int off, int len) throws IOException;  // 
Overrides OutputStream
            public void writeBoolean(boolean data) throws IOException;  // From 
DataOutput
            public void writeByte(int data) throws IOException;  // From DataOutput
            public void writeBytes(String data) throws IOException;  // From 
DataOutput
            public void writeChar(int data) throws IOException;  // From DataOutput
            public void writeChars(String data) throws IOException;  // From 
DataOutput
            public void writeDouble(double data) throws IOException;  // From 
DataOutput
            public void writeFloat(float data) throws IOException;  // From 
DataOutput
            public void writeInt(int data) throws IOException;  // From DataOutput
            public void writeLong(long data) throws IOException;  // From DataOutput
            public final void writeObject(Object obj) throws IOException;  // From 
ObjectOutput
```

```
            public void writeShort(int data) throws IOException;  // From DataOutput
            public void writeUTF(String data) throws IOException;  // From DataOutput
    // Protected Instance Methods
            protected void annotateClass(Class cl) throws IOException;
            protected void drain() throws IOException;
            protected final boolean enableReplaceObject(boolean enable) throws
SecurityException;
            protected Object replaceObject(Object obj) throws IOException;
            protected void writeStreamHeader() throws IOException;
}
```

Hierarchy:

Object->OutputStream->ObjectOutputStream(ObjectOutput(DataOutput))

Passed To:

AWTEventMulticaster.save(), AWTEventMulticaster.saveInternal()

25.58 java.lang.StringBuffer (JDK 1.0)

This class represents a string of characters. It differs from the `String` class in that its contents may be modified. A `StringBuffer` object grows in length as necessary. The string stored in a `StringBuffer` object may be modified in place with the `setCharAt()`, `append()`, and `insert()` methods.

After a string is processed in a `StringBuffer` object, it may be efficiently converted to a `String` object for subsequent use. The `StringBuffer.toString()` method does not copy the internal array of characters; instead it shares that array with the new `String` object, and makes a new copy for itself only when further modifications are made to the `StringBuffer` object.

```
public final class StringBuffer extends Object implements Serializable {
    // Public Constructors
            public StringBuffer();
            public StringBuffer(int length);
            public StringBuffer(String str);
    // Public Instance Methods
            public synchronized StringBuffer append(Object obj);
            public synchronized StringBuffer append(String str);
            public synchronized StringBuffer append(char[] str);
            public synchronized StringBuffer append(char[] str, int offset, int len);
            public StringBuffer append(boolean b);
            public synchronized StringBuffer append(char c);
            public StringBuffer append(int i);
            public StringBuffer append(long l);
            public StringBuffer append(float f);
            public StringBuffer append(double d);
            public int capacity();
            public synchronized char charAt(int index);
            public synchronized void ensureCapacity(int minimumCapacity);
            public synchronized void getChars(int srcBegin, int srcEnd, char[] dst, 
int dstBegin);
            public synchronized StringBuffer insert(int offset, Object obj);
            public synchronized StringBuffer insert(int offset, String str);
            public synchronized StringBuffer insert(int offset, char[] str);
            public StringBuffer insert(int offset, boolean b);
            public synchronized StringBuffer insert(int offset, char c);
            public StringBuffer insert(int offset, int i);
            public StringBuffer insert(int offset, long l);
            public StringBuffer insert(int offset, float f);
            public StringBuffer insert(int offset, double d);
            public int length();
            public synchronized StringBuffer reverse();
            public synchronized void setCharAt(int index, char ch);
            public synchronized void setLength(int newLength);
            public String toString();  // Overrides Object
}
```

Passed To:

ChoiceFormat.format(), DateFormat.format(), DecimalFormat.format(), Format.format(), MessageFormat.format(), NumberFormat.format(), SimpleDateFormat.format(), String()

Returned By:

ChoiceFormat.format(), DateFormat.format(), DecimalFormat.format(), Format.format(), MessageFormat.format(), NumberFormat.format(), SimpleDateFormat.format(), StringBuffer.append(), StringBuffer.insert(), StringBuffer.reverse(), StringWriter.getBuffer()

PREVIOUS	HOME	NEXT
java.lang.String (JDK 1.0)	BOOK INDEX	java.lang.StringIndexOutOfBoundsException (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

29.9 java.text.DecimalFormat (JDK 1.1)

This is the concrete `Format` class used by `NumberFormat` for all locales that use base 10 numbers. Most applications do not need to use this class directly--they can use the static methods of `NumberFormat` to obtain a default `NumberFormat` object for a desired locale, and may then perform minor locale-independent customizations on that object.

Applications that require highly-customized number formatting and parsing may create custom `DecimalFormat` objects by passing a suitable pattern to the `DecimalFormat()` constructor method. The `applyPattern()` method can be used to change this pattern. A pattern consists of a string of characters from the following table. For example: "$#,##0.00;($#,##0.00)".

Character	Meaning
#	A digit; zeros show as absent
0	A digit; zeros show as 0
.	The locale-specific decimal separator
,	The locale-specific grouping separator
-	The locale-specific negative prefix
%	Show value as a percentage
;	Separates positive number format (on left) from optional negative number format (on right)
'	Quote a reserved character, so it appears literally in the output
other	Appears literally in output

A `DecimalFormatSymbols` object may be optionally specified when creating a `DecimalFormat` object. If one is not specified, a `DecimalFormatSymbols` object suitable for the default locale is used.

```
public class DecimalFormat extends NumberFormat {
    // Public Constructors
            public DecimalFormat();
            public DecimalFormat(String pattern);
            public DecimalFormat(String pattern, DecimalFormatSymbols symbols);
    // Public Instance Methods
            public void applyLocalizedPattern(String pattern);
            public void applyPattern(String pattern);
            public Object clone();  // Overrides NumberFormat
            public boolean equals(Object obj);  // Overrides NumberFormat
            public StringBuffer format(double number, StringBuffer result,
FieldPosition fieldPosition);  // Defines NumberFormat
            public StringBuffer format(long number, StringBuffer result,
FieldPosition fieldPosition);  // Defines NumberFormat
            public DecimalFormatSymbols getDecimalFormatSymbols();
            public int getGroupingSize();
            public int getMultiplier();
            public String getNegativePrefix();
            public String getNegativeSuffix();
            public String getPositivePrefix();
            public String getPositiveSuffix();
            public int hashCode();  // Overrides NumberFormat
            public boolean isDecimalSeparatorAlwaysShown();
            public Number parse(String text, ParsePosition status);  // Defines
NumberFormat
            public void setDecimalFormatSymbols(DecimalFormatSymbols newSymbols);
            public void setDecimalSeparatorAlwaysShown(boolean newValue);
            public void setGroupingSize(int newValue);
            public void setMultiplier(int newValue);
            public void setNegativePrefix(String newValue);
            public void setNegativeSuffix(String newValue);
            public void setPositivePrefix(String newValue);
            public void setPositiveSuffix(String newValue);
            public String toLocalizedPattern();
            public String toPattern();
}
```

Hierarchy:

Object->Format(Serializable, Cloneable)->NumberFormat(Cloneable)->DecimalFormat

PREVIOUS	HOME	NEXT
java.text.DateFormatSymbols (JDK 1.1)	BOOK INDEX	java.text.DecimalFormatSymbols (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

29.18 java.text.SimpleDateFormat (JDK 1.1)

This is the concrete `Format` subclass used by `DateFormat` to handle formatting and parsing of dates. Most applications should not use this class directly--instead, they should obtain a localized `DateFormat` object by calling one of the static methods of `DateFormat`.

`SimpleDateFormat` formats dates and times according to a pattern, which specifies the positions of the various fields of the date, and a `DateFormatSymbols` object, which specifies important auxiliary data, such as the names of months. Applications that require highly customized date or time formatting can create a custom `SimpleDateFormat` object by specifying the desired pattern. This creates a `SimpleDateFormat` object that uses the `DateFormatSymbols` object for the default locale. You may also specify a locale explicitly to use the `DateFormatSymbols` object for that locale. Or, you can even provide an explicit `DateFormatSymbols` object of your own, if you need to format dates and times for an unsupported locale.

You can use the `applyPattern()` method of a `SimpleDateFormat` to change the formatting pattern used by the object. The syntax of this pattern is described in the following table. Any characters in the format string that do not appear in this table appear literally in the formatted date.

Field	**Full Form**	**Short Form**
Year	`yyyy` (4 digits)	`yy` (2 digits)
Month	`MMM` (name)	`MM` (2 digits), `M` (1 or 2 digits)
Day of week	`EEEE`	`EE`
Day of month	`dd` (2 digits)	`d` (1 or 2 digits)
Hour (1-12)	`hh` (2 digits)	`h` (1 or 2 digits)
Hour (0-23)	`HH` (2 digits)	`H` (1 or 2 digits)
Hour (0-11)	`KK`	`K`
Hour (1-24)	`kk`	`k`
Minute	`mm`	
Second	`ss`	
Millisecond	`SSS`	
AM/PM	`a`	
Time zone	`zzzz`	`zz`
Day of week in month	`F` (e.g., 3rd Thursday)	
Day in year	`DDD` (3 digits)	`D` (1, 2, or 3 digits)
Week in year	`ww`	
Era (e.g., BC/AD)	G	

```
public class SimpleDateFormat extends DateFormat {
    // Public Constructors
            public SimpleDateFormat();
            public SimpleDateFormat(String pattern);
            public SimpleDateFormat(String pattern, Locale loc);
            public SimpleDateFormat(String pattern, DateFormatSymbols formatData);
    // Public Instance Methods
            public void applyLocalizedPattern(String pattern);
            public void applyPattern(String pattern);
            public Object clone();  // Overrides DateFormat
            public boolean equals(Object obj);  // Overrides DateFormat
            public StringBuffer format(Date date, StringBuffer toAppendTo, 
FieldPosition pos);  // Defines DateFormat
            public DateFormatSymbols getDateFormatSymbols();
            public int hashCode();  // Overrides DateFormat
            public Date parse(String text, ParsePosition pos);  // Defines DateFormat
            public void setDateFormatSymbols(DateFormatSymbols newFormatSymbols);
            public String toLocalizedPattern();
            public String toPattern();
}
```

Hierarchy:

Object->Format(Serializable, Cloneable)->DateFormat(Cloneable)->SimpleDateFormat

PREVIOUS | HOME | NEXT
java.text.RuleBasedCollator (JDK 1.1) | BOOK INDEX | java.text.StringCharacterIterator (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

29.3 java.text.ChoiceFormat (JDK 1.1)

This class is a subclass of `Format` that converts a number to a `String` in a way that is reminiscent of a `switch` statement or an enumerated type. Each `ChoiceFormat` object has an array of doubles known as its "limits" and an array of strings known as its "formats." When the `format()` method is called to format a number *x*, the `ChoiceFormat` finds an index *i* such that:

```
limits[i] <= x < limits[i+1]
```

If *x* is less than the first element of the array, the first element is used, and if it is greater than the last, the last element is used. Once the index *i* has been determined, it is used as the index into the array of strings, and the indexed string is returned as the result of the `format()` method.

A `ChoiceFormat` object may also be created by encoding its "limits" and "formats" into a single string known as its "pattern." A typical pattern looks like the one that follows, used to return the singular or plural form of a word, based on the numeric value passed to the `format()` method:

```
ChoiceFormat cf = new ChoiceFormat("0#errors|1#error|2#errors");
```

A `ChoiceFormat` object created in this way returns the string "errors" when it formats the number 0, or any number greater than or equal to 2. It returns "error" when it formats the number 1. In the syntax shown here, note the pound sign (#) used to separate the limit number from the string that corresponds to that case and the vertical bar (`|`) used to separate the individual cases. You can use the `applyPattern()` method to change the pattern used by a `ChoiceFormat` object; use `toPattern()` to query the pattern it uses.

```
public class ChoiceFormat extends NumberFormat {
    // Public Constructors
            public ChoiceFormat(String newPattern);
            public ChoiceFormat(double[] limits, String[] formats);
    // Class Methods
            public static final double nextDouble(double d);
            public static double nextDouble(double d, boolean positive);
            public static final double previousDouble(double d);
    // Public Instance Methods
            public void applyPattern(String newPattern);
            public Object clone();  // Overrides NumberFormat
            public boolean equals(Object obj);  // Overrides NumberFormat
            public StringBuffer format(long number, StringBuffer toAppendTo,
            FieldPosition status);  // Defines NumberFormat
            public StringBuffer format(double number, StringBuffer toAppendTo,
            FieldPosition status);  // Defines NumberFormat
            public Object[] getFormats();
            public double[] getLimits();
            public int hashCode();  // Overrides NumberFormat
            public Number parse(String text, ParsePosition status);  // Defines
NumberFormat
            public void setChoices(double[] limits, String[] formats);
            public String toPattern();
}
```

Hierarchy:

Object->Format(Serializable, Cloneable)->NumberFormat(Cloneable)->ChoiceFormat

29.13 java.text.MessageFormat (JDK 1.1)

This class formats and substitutes objects into specified positions in a message string (also known as the "pattern" string). It provides the closest Java equivalent to the `printf()` function of the C programming language.

If a message is to be displayed only a single time, the simplest way to use the `MessageFormat` class is through the static `format()` method which is passed a message or pattern string and an array of argument objects to be formatted and substituted into the string. If the message is to be displayed several times, it makes more sense to create a `MessageFormat` object, supplying the pattern string, and then to call the `format()` instance method of this object, supplying the array of objects to be formatted into the message.

The message or pattern string used by the `MessageFormat` should contain digits enclosed in curly braces to indicate where each argument should be substituted. The sequence "`{0}`" indicates that the first object should be converted to a string (if necessary) and inserted at that point, while the sequence "`{3}`" indicates that the fourth object should be inserted, for example. If the object to be inserted is not a string, `MessageFormat` checks to see if it is a `Date` or a subclass of `Number`. If so, it uses a default `DateFormat` or `NumberFormat` object to convert the value to a string. If not, it simply invokes the object's `toString()` method to convert it.

A digit within curly braces in a pattern string may be optionally followed by a comma, and one of the words "date", "time", "number", or "choice", to indicate that the corresponding argument should be formatted as a date, time, number, or choice before being substituted into the pattern string. Any of these keywords may additionally be followed by a comma and additional pattern information to be used in formatting the date, time, number, or choice. (See `SimpleDateFormat`, `DecimalFormat`, and `ChoiceFormat` for more information.)

You can use the `setLocale()` method to specify a non-default `Locale` that the `MessageFormat` should use when obtaining `DateFormat` and `NumberFormat` objects to format dates, time, and numbers inserted into the pattern.

You can change the `Format` object used at a particular position in the pattern with the `setFormat()` method. You can set a new pattern for the `MessageFormat` object by calling `applyPattern()`, and you can obtain a string that represents the current formatting pattern by calling `toPattern()`.

`MessageFormat` also supports a `parse()` method that can parse an array of objects out of a specified string, according to the specified pattern.

```
public class MessageFormat extends Format {
    // Public Constructor
            public MessageFormat(String pattern);
    // Class Methods
            public static String format(String pattern, Object[] arguments);
    // Public Instance Methods
```

```
            public void applyPattern(String newPattern);
            public Object clone();  // Overrides Format
            public boolean equals(Object obj);  // Overrides Object
            public final StringBuffer format(Object[] source, StringBuffer result, 
FieldPosition ignore);
            public final StringBuffer format(Object source, StringBuffer result, 
FieldPosition ignore);  // Defines Format
            public Format[] getFormats();
            public Locale getLocale();
            public int hashCode();  // Overrides Object
            public Object[] parse(String source, ParsePosition status);
            public Object[] parse(String source) throws ParseException;
            public Object parseObject(String text, ParsePosition status);  // Defines 
Format
            public void setFormat(int variable, Format newFormat);
            public void setFormats(Format[] newFormats);
            public void setLocale(Locale theLocale);
            public String toPattern();
}
```

Hierarchy:

Object->Format(Serializable, Cloneable)->MessageFormat

PREVIOUS
java.text.Format (JDK 1.1)

HOME
BOOK INDEX

NEXT
java.text.NumberFormat (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

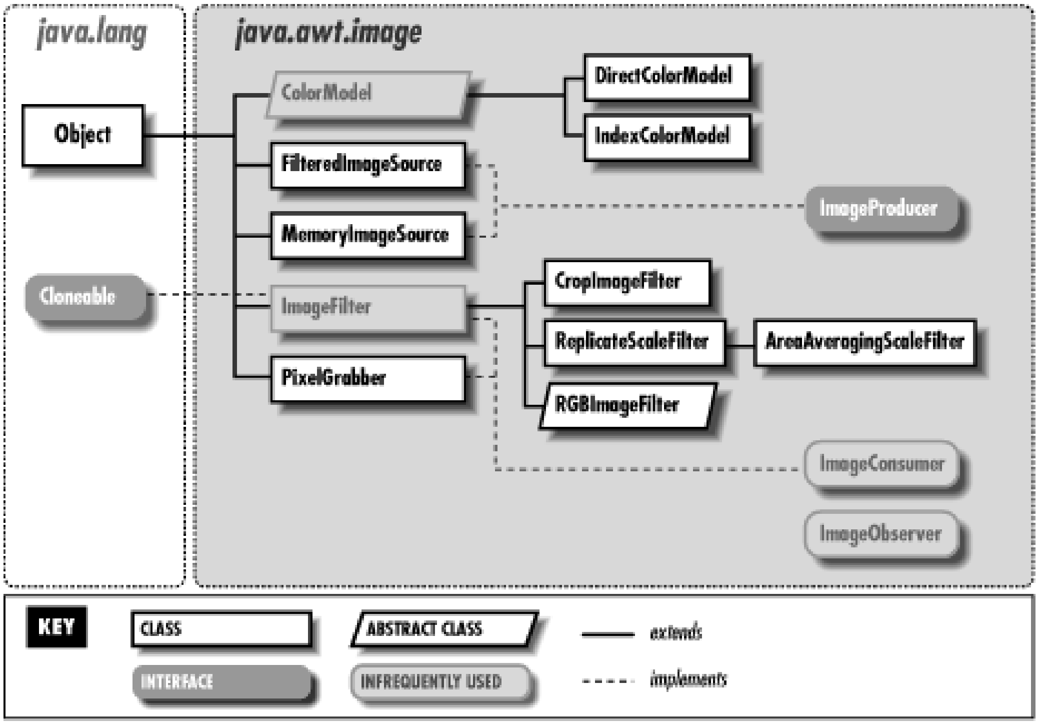
21. The java.awt.image Package

Contents:

The `java.awt.image` package is, by any standard, a confusing one. The purpose of the package is to support image processing, and the classes in the package provide a powerful infrastructure for that purpose. (see Figure 21.1.) Most of the classes are part of the infrastructure, however, and are not normally used by ordinary applications that have only simple image manipulation requirements.

To understand this package, it is first important to note that the `Image` class itself is part of the `java.awt` package, not the `java.awt.image` package. Furthermore, the `java.awt.image` classes are not the source of images; they simply serve to manipulate images that come from somewhere else. The `Applet.getImage()` method is perhaps the most common method for obtaining an image in Java--it downloads the image from a specified URL. In a stand-alone application, the `URL.getContent()` method can be used to obtain an `ImageProducer` object, which can then be passed to the `Component.createImage()` method to obtain an `Image` object.

Figure 21.1: The java.awt.image package



The ImageProducer interface is one you'll encounter frequently in java.awt.image. It represents an image source. If you've created an Image object with Applet.getImage(), you can obtain the ImageProducer for that Image (which has not been downloaded yet) with Image.getSource(). Conversely, given an ImageProducer object, you can create an Image from it with the createImage() method of any Component (such as an Applet). Once you have an ImageProducer object, you can manipulate it with the other java.awt.image classes.

FilteredImageSource is the most important class for image manipulation. It is itself a type of ImageProducer that, when created, applies a specified ImageFilter object to some other specified ImageProducer object. The FilteredImageSource thus configured can be used as an ImageProducer to display a filtered image. CropImageFilter is a predefined type of ImageFilter that you can use to extract a specified rectangle out of a larger image.

RGBImageFilter is another subclass of ImageFilter that makes it easy to filter the colors of an image. To do so, you must subclass RGBImageFilter and provide the definition of a single simple method that manipulates the image colors. In order to manipulate image colors, you will probably need to be familiar with the ColorModel class and its two subclasses, DirectColorModel and IndexColorModel. An instance of ColorModel or of one of its subclasses converts a pixel value to the red, green, and blue components of the color it represents.

Finally, two other classes in the java.awt.image package are worth noting. MemoryImageSource is a type of ImageProducer that generates an image from an array of bytes or integers in memory. PixelGrabber does the reverse--it captures pixels from an ImageProducer and stores them into an array. You can use these classes separately or together to perform your own custom image manipulation.

21.1 java.awt.image.AreaAveragingScaleFilter (JDK 1.1)

This class implements an ImageFilter that scales an image to a specified pixel size. It uses a scaling algorithm that averages adjacent pixel values when shrinking an image, which produces relatively smooth scaled images. Its superclass, ReplicateScaleFilter, implements a faster, less smooth scaling algorithm.

The methods of this class are ImageConsumer methods intended for communication between the image filter and the FilteredImageSource that uses it. Applications do not usually call these methods directly.

The easiest way to use this filter is to call `Image.getScaledInstance()`, specifying an appropriate hint constant.

```
public class AreaAveragingScaleFilter extends ReplicateScaleFilter {
    // Public Constructor
            public AreaAveragingScaleFilter(int width, int height);
    // Public Instance Methods
            public void setHints(int hints);  // Overrides ImageFilter
            public void setPixels(int x, int y, int w, int h, ColorModel model,
byte[] pixels, int off, int scansize);
            public void setPixels'u'// Overrides ReplicateScaleFilter
            public void setPixels(int x, int y, int w, int h, ColorModel model, int[]
pixels, int off, int scansize);
            public void setPixels'u'// Overrides ReplicateScaleFilter
}
```

Hierarchy:

```
Object->ImageFilter(ImageConsumer, Cloneable)-> ReplicateScaleFilter-
>AreaAveragingScaleFilter
```

PREVIOUS | HOME | NEXT

java.awt.event.WindowListener (JDK 1.1) | BOOK INDEX | java.awt.image.ColorModel (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

26. The java.lang.reflect Package

Contents:

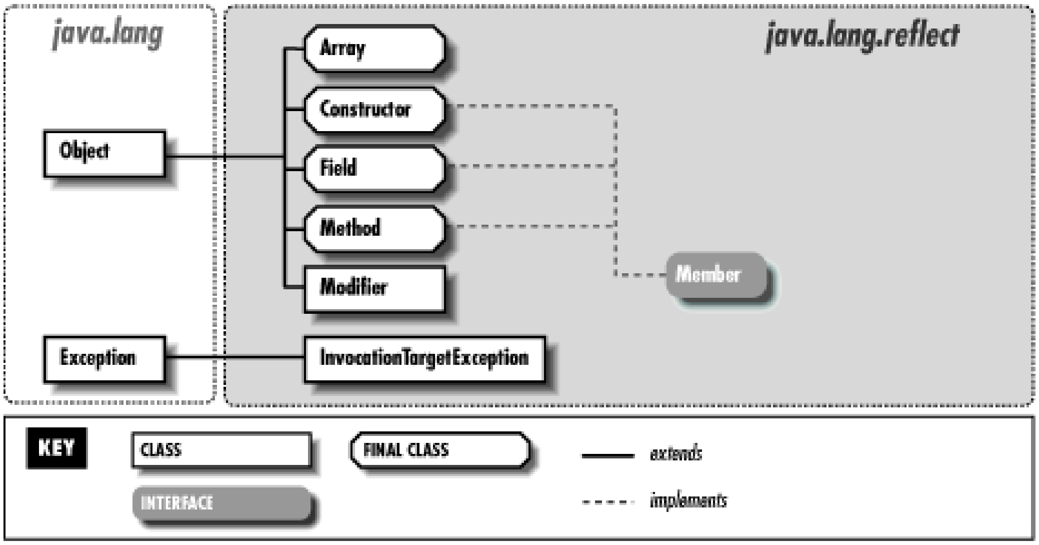
The `java.lang.reflect` package contains the classes and interfaces that, along with `java.lang.Class`, comprise the Java Reflection API. This package is new in Java 1.1. Figure 26.1 shows the class hierarchy.

The `Constructor`, `Field`, and `Method` classes represent constructors, fields, and methods of a class. Because these types all represent members of a class, they each implement the `Member` interface, which defines a simple set of methods that can be invoked for any class member. These classes allow information about the class members to be obtained, and allow methods and constructors to be invoked and fields to be queried and set.

Class member modifiers are represented as integers that specify a number of bit flags. The `Modifier` class defines static methods that are useful in interpreting the meanings of these flags. The `Array` class defines static methods for creating arrays and reading and writing array elements.

See Chapter 12, *Reflection* for examples of using these classes.

Figure 26.1: The java.lang.reflect package



26.1 java.lang.reflect.Array (JDK 1.1)

This class contains methods that allow you to set and query the values of array elements, to determine the length of an array, and to create new instances of arrays. Note that the `Array` class is used for manipulating array values, not array types; Java data types, including array types, are represented by `java.lang.Class`. Since the `Array` class represents a Java value, unlike the `Field`, `Method`, and `Constructor` classes which represent class members, the `Array` class is significantly different (despite some surface similarities) from those other classes in this package. Most notably, all the methods of `Array` are static and apply to all array values, rather than applying only to a specific field, method, or constructor.

The `get()` method returns the value of the specified element of the specified array as an `Object`. If the array elements are of a primitive type, the value is converted to a wrapper object before being returned. You can also use `getInteger()` and related methods to query array elements and return them as specific primitive types. The `set()` method and its primitive type variants perform the opposite operation. Also, the `getLength()` method returns the length of the array.

The `newInstance()` methods create new arrays. One version of this method is passed the number of elements in the array and the type of those elements. The other version of this method creates multidimensional arrays. Besides just specifying the component type of the array, it is passed an array of numbers. The length of this array specifies the number of dimensions for the array to be created, and the values of each array element specify the size of each dimension of the created array.

```
public final class Array extends Object {
    // No Constructor
    // Class Methods
            public static native Object get(Object array, int index) throws 
IllegalArgumentException, ArrayIndexOutOfBoundsException;
            public static native boolean getBoolean(Object array, int index) throws 
IllegalArgumentException, ArrayIndexOutOfBoundsException;
            public static native byte getByte(Object array, int index) throws 
IllegalArgumentException, ArrayIndexOutOfBoundsException;
            public static native char getChar(Object array, int index) throws 
IllegalArgumentException, ArrayIndexOutOfBoundsException;
            public static native double getDouble(Object array, int index) throws 
IllegalArgumentException, ArrayIndexOutOfBoundsException;
            public static native float getFloat(Object array, int index) throws 
IllegalArgumentException, ArrayIndexOutOfBoundsException;
            public static native int getInt(Object array, int index) throws 
IllegalArgumentException, ArrayIndexOutOfBoundsException;
```

```
            public static native int getLength(Object array) throws
IllegalArgumentException;
            public static native long getLong(Object array, int index) throws
IllegalArgumentException, ArrayIndexOutOfBoundsException;
            public static native short getShort(Object array, int index) throws
IllegalArgumentException, ArrayIndexOutOfBoundsException;
            public static Object newInstance(Class componentType, int length) throws
NegativeArraySizeException;
            public static Object newInstance(Class componentType, int[] dimensions)
throws IllegalArgumentException, NegativeArraySizeException;
            public static native void set(Object array, int index, Object value)
throws IllegalArgumentException, ArrayIndexOutOfBoundsException;
            public static native void setBoolean(Object array, int index, boolean z)
throws IllegalArgumentException, ArrayIndexOutOfBoundsException;
            public static native void setByte(Object array, int index, byte b)
Xsthrows IllegalArgumentException, ArrayIndexOutOfBoundsException;
            public static native void setChar(Object array, int index, char c) throws
IllegalArgumentException, ArrayIndexOutOfBoundsException;
            public static native void setDouble(Object array, int index, double d)
throws IllegalArgumentException, ArrayIndexOutOfBoundsException;
            public static native void setFloat(Object array, int index, float f)
throws IllegalArgumentException, ArrayIndexOutOfBoundsException;
            public static native void setInt(Object array, int index, int i) throws
IllegalArgumentException, ArrayIndexOutOfBoundsException;
            public static native void setLong(Object array, int index, long l) throws
IllegalArgumentException, ArrayIndexOutOfBoundsException;
            public static native void setShort(Object array, int index, short s)
throws IllegalArgumentException, ArrayIndexOutOfBoundsException;
}
```

PREVIOUS | HOME | NEXT
java.lang.Void (JDK 1.1) | BOOK INDEX | java.lang.reflect.Constructor (JDK 1.1)

25.60 java.lang.System (JDK 1.0)

This class defines methods that provide a platform-independent interface to system functions. All of the methods and variables of this class are static. The class may not be instantiated.

`getProperty()` looks up a named property on the system properties list, returning the optionally specified default value if no property definition was found. `getProperties()` returns the entire properties list. `setProperties()` sets a `Properties` object on the properties list.

The `in`, `out`, and `err` variables are the system standard input, output, and error streams. `arraycopy()` copies an array or a portion of an array into a destination array. `currentTimeMillis()` returns the current time in milliseconds since midnight GMT on January 1, 1970.

`exit()`, `gc()`, `load()`, `loadLibrary()`, and `runFinalization()` invoke the methods of the same name in the `Runtime` object. See `Runtime` for details.

```
public final class System extends Object {
    // No Constructor
    // Constants
            public static final PrintStream err;
            public static final InputStream in;
            public static final PrintStream out;
    // Class Methods
            public static native void arraycopy(Object src, int src_position, Object
dst, int dst_position, int length);
            public static native long currentTimeMillis();
            public static void exit(int status);
            public static void gc();
            public static Properties getProperties();
            public static String getProperty(String key);
            public static String getProperty(String key, String def);
            public static SecurityManager getSecurityManager();
        #   public static String getenv(String name);
        1.1public static native int identityHashCode(Object x);
            public static void load(String filename);
            public static void loadLibrary(String libname);
            public static void runFinalization();
        1.1public static void runFinalizersOnExit(boolean value);
        1.1public static void setErr(PrintStream err);
        1.1public static void setIn(InputStream in);
        1.1public static void setOut(PrintStream out);
            public static void setProperties(Properties props);
            public static void setSecurityManager(SecurityManager s);
}
```

PREVIOUS | HOME | NEXT

java.lang.StringIndexOutOfBoundsException (JDK 1.0) | BOOK INDEX | java.lang.Thread (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

23.6 java.beans.FeatureDescriptor (JDK 1.1)

The `FeatureDescriptor` class is the base class for `MethodDescriptor` and `PropertyDescriptor`, as well as other classes used by the JavaBeans introspection mechanism. It provides basic information about a feature (method, property, event, etc.) of a bean. Typically, the methods that begin with `get` and `is` are used by application builders or other tools to query the features of a bean. The `set` methods, on the other hand, may be used by bean authors to define information about the bean.

`setName()` specifies the locale-independent, programmatic name of the feature. `setDisplayName()` specifies a localized, human-readable name. `setShortDescription()` specifies a short localized string (about 40 characters) that describes the feature. Both the short description and the localized name default to the value of the programmatic name. `setExpert()` and `setHidden()` allow you to indicate that the feature is for use only by experts, or for use only by the builder tool, and should be hidden from users of the builder. Finally, the `setValue()` method allows you to associate an arbitrary named value with the feature.

```
public class FeatureDescriptor extends Object {
    // Public Constructor
            public FeatureDescriptor();
    // Public Instance Methods
            public Enumeration attributeNames();
            public String getDisplayName();
            public String getName();
            public String getShortDescription();
            public Object getValue(String attributeName);
            public boolean isExpert();
            public boolean isHidden();
            public void setDisplayName(String displayName);
            public void setExpert(boolean expert);
            public void setHidden(boolean hidden);
            public void setName(String name);
            public void setShortDescription(String text);
            public void setValue(String attributeName, Object value);
}
```

Extended By:

BeanDescriptor, EventSetDescriptor, MethodDescriptor, ParameterDescriptor, PropertyDescriptor

PREVIOUS | HOME | NEXT
java.beans.EventSetDescriptor (JDK 1.1) | BOOK INDEX | java.beans.IndexedPropertyDescriptor (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

24. The java.io Package

Contents:

java.io.LineNumberReader (JDK 1.1)
java.io.NotActiveException (JDK 1.1)
java.io.NotSerializableException (JDK 1.1)
java.io.ObjectInput (JDK 1.1)
java.io.ObjectInputStream (JDK 1.1)
java.io.ObjectInputValidation (JDK 1.1)
java.io.ObjectOutput (JDK 1.1)
java.io.ObjectOutputStream (JDK 1.1)
java.io.ObjectStreamClass (JDK 1.1)
java.io.ObjectStreamException (JDK 1.1)
java.io.OptionalDataException (JDK 1.1)
java.io.OutputStream (JDK 1.0)
java.io.OutputStreamWriter (JDK 1.1)
java.io.PipedInputStream (JDK 1.0)
java.io.PipedOutputStream (JDK 1.0)
java.io.PipedReader (JDK 1.1)
java.io.PipedWriter (JDK 1.1)
java.io.PrintStream (JDK 1.0)
java.io.PrintWriter (JDK 1.1)
java.io.PushbackInputStream (JDK 1.0)
java.io.PushbackReader (JDK 1.1)
java.io.RandomAccessFile (JDK 1.0)
java.io.Reader (JDK 1.1)
java.io.SequenceInputStream (JDK 1.0)
java.io.Serializable (JDK 1.1)
java.io.StreamCorruptedException (JDK 1.1)
java.io.StreamTokenizer (JDK 1.0)
java.io.StringBufferInputStream (JDK 1.0; Deprecated.)
java.io.StringReader (JDK 1.1)
java.io.StringWriter (JDK 1.1)
java.io.SyncFailedException (JDK 1.1)
java.io.UnsupportedEncodingException (JDK 1.1)
java.io.UTFDataFormatException (JDK 1.0)
java.io.WriteAbortedException (JDK 1.1)
java.io.Writer (JDK 1.1)

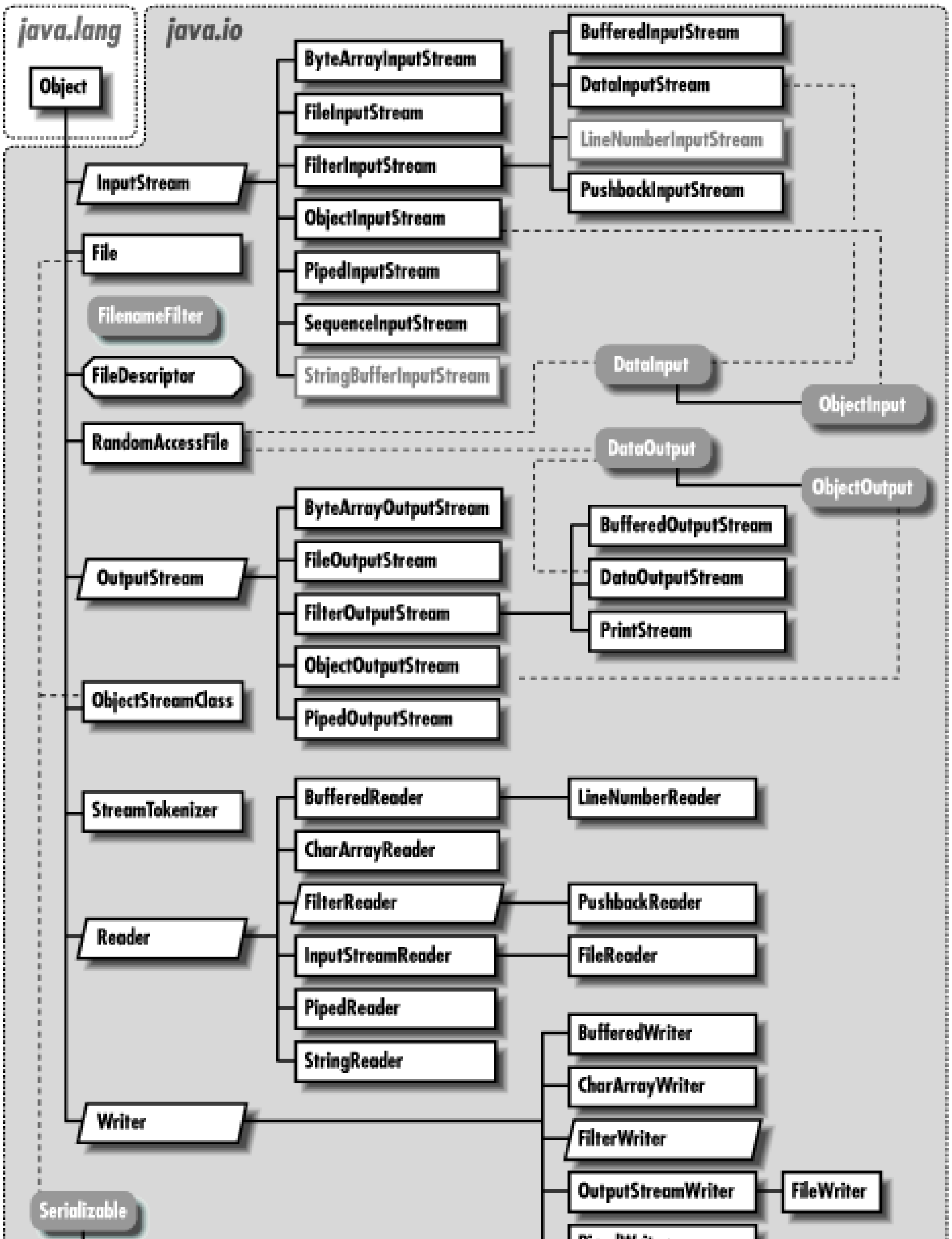
The `java.io` package contains a relatively large number of classes, but, as you can see from Figure 24.1 and Figure 24.2, the classes form a fairly structured hierarchy. Most of the package consists of byte streams--subclasses of `InputStream` or `OutputStream`--and (in Java 1.1) character streams--subclasses of `Reader` or `Writer`. Each of these stream types has a very specific purpose, and despite its size, `java.io` is a straightforward package to understand and to use.

Before we consider the stream classes in the package, we'll consider the important non-stream classes. `File` represents a file or directory name in a system independent way, and provides methods for listing directories, querying file attributes, and for renaming and deleting files. `FilenameFilter` is an interface that defines a method that accepts or rejects specified filenames. It is used by the `java.awt.FileDialog` dialog box and by the `File` class to specify what types of files should be included in directory listings. `RandomAccessFile` allows you to read or write from or to arbitrary locations of a file. Often, though, you'll prefer sequential access to a file and should use one of the stream classes.

`InputStream` and `OutputStream` are abstract classes that define methods for reading and writing bytes. Their subclasses allow bytes to be read from and written to a variety of sources and sinks. `FileInputStream` and `FileOutputStream` read from and write to files. `ByteArrayInputStream` and `ByteArrayOutputStream` read from and write to an array of bytes in memory. `PipedInputStream` reads bytes from a `PipedOutputStream`, and `PipedOutputStream` writes bytes to a `PipedInputStream`. These classes work together to implement a "pipe" for communication between threads.

`FilterInputStream` and `FilterOutputStream` are special--they filter input and output bytes. When a `FilterInputStream` is created, an `InputStream` is specified for it to filter. When you call the `read()` method of a `FilterInputStream`, it calls the `read()` method of its specified stream, processes the bytes it reads somehow, and then returns the filtered bytes. Similarly, you specify an `OutputStream` to be filtered when you create a `FilterOutputStream`. Calling the `write()` method of a `FilterOutputStream` causes it to process your bytes in some way and then pass those filtered bytes to the `write()` method of its `OutputStream`.

Figure 24.1: The java.io package



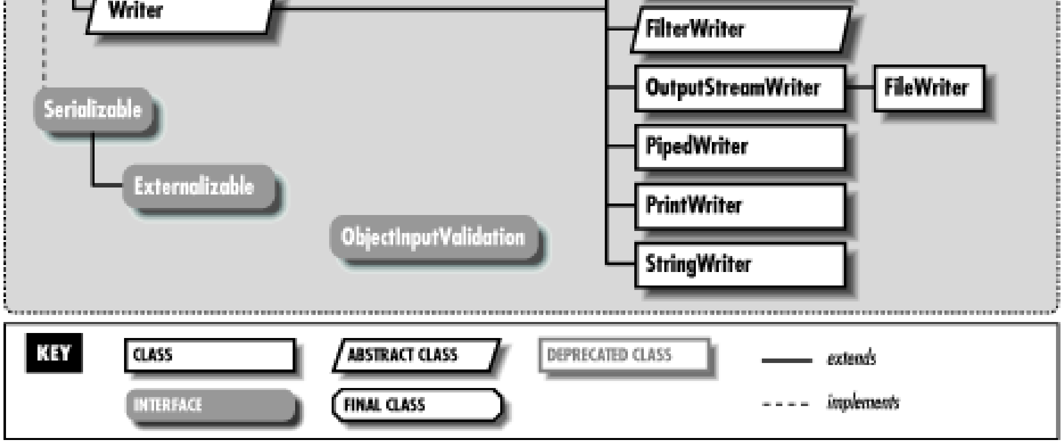
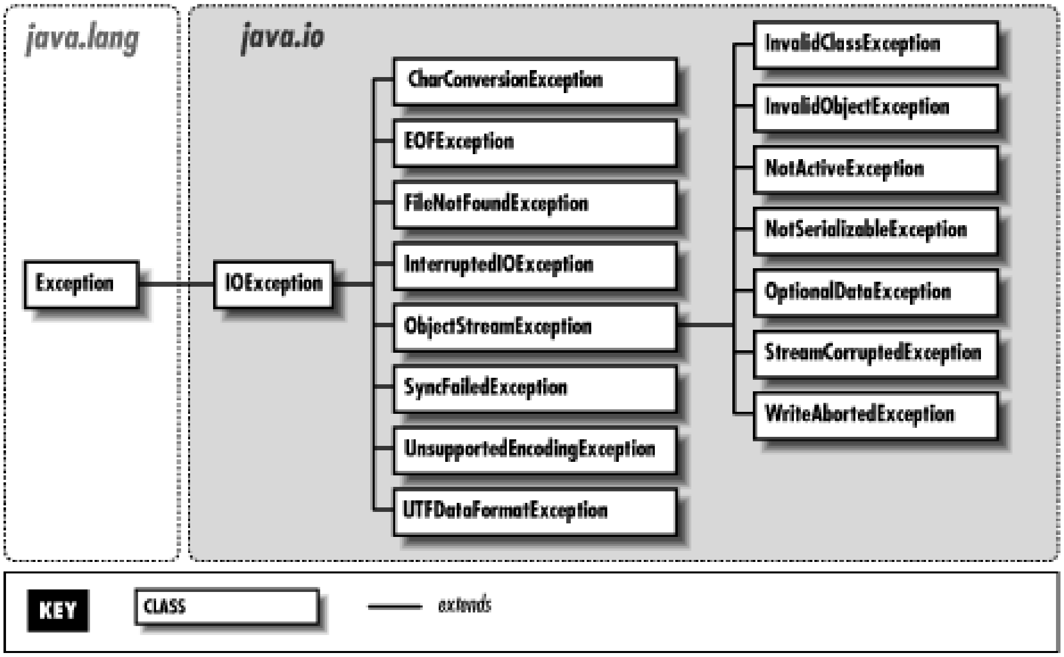


Figure 24.2: The exception classes of the java.io package



`FilterInputStream` and `FilterOutputStream` do not perform any filtering themselves; this is done by their subclasses. `BufferedInputStream` and `BufferedOutputStream` provide input and output buffering and can increase I/O efficiency. `DataInputStream` reads raw bytes from a stream and interprets them in various binary formats. It has various methods to read primitive Java data types in their standard binary formats. `DataOutputStream` allows you to write Java primitive data types in binary format.

In Java 1.1, the byte streams just described are complemented by an analogous set of character input and output streams. `Reader` is the superclass of all character input streams, and `Writer` is the superclass of all character output streams. These character streams supersede the byte streams for all textual I/O. They are more efficient than the byte streams, and they correctly handle the conversion between local encodings and Unicode text, making them invaluable for internationalized programs. Most of the `Reader` and `Writer` streams have obvious byte stream analogs. `BufferedReader` is a commonly used stream. It provides buffering for efficiency, and also has a `readLine()` method to read one line of text at a time. `PrintWriter` is another very common stream--its methods allow output of a textual representation of any primitive Java type or of any object (via the object's `toString()` method). See Chapter 11, *Internationalization* for a discussion of the use of character streams in internationalized programs.

The `ObjectInputStream` and `ObjectOutputStream` classes are special. These byte stream classes are new in Java 1.1 and are part of the Object Serialization API. See Chapter 9, *Object Serialization* for further details.

24.1 java.io.BufferedInputStream (JDK 1.0)

This class is a `FilterInputStream` that provides input data buffering--efficiency is increased by reading in large amounts of data and storing them in an internal buffer. When data is requested, it is usually available from the buffer. Thus, most calls to read data do not have to make a call to actually read data from a disk, network, or other slow source. Create a `BufferedInputStream` by specifying the `InputStream` that is to have buffering applied in the call to the constructor. See also `BufferedReader`.

```
public class BufferedInputStream extends FilterInputStream {
    // Public Constructors
            public BufferedInputStream(InputStream in);
            public BufferedInputStream(InputStream in, int size);
    // Protected Instance Variables
            protected byte[] buf;
            protected int count;
            protected int marklimit;
            protected int markpos;
            protected int pos;
    // Public Instance Methods
            public synchronized int available() throws IOException;  // Overrides 
FilterInputStream
            public synchronized void mark(int readlimit);  // Overrides 
FilterInputStream
            public boolean markSupported();  // Overrides FilterInputStream
            public synchronized int read() throws IOException;  // Overrides 
FilterInputStream
            public synchronized int read(byte[] b, int off, int len) throws 
IOException;  // Overrides FilterInputStream
            public synchronized void reset() throws IOException;  // Overrides 
FilterInputStream
            public synchronized long skip(long n) throws IOException;  // Overrides 
FilterInputStream
}
```

Hierarchy:

Object->InputStream->FilterInputStream->BufferedInputStream

PREVIOUS | HOME | NEXT
java.beans.Visibility (JDK 1.1) | BOOK INDEX | java.io.BufferedOutputStream (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

24.5 java.io.ByteArrayInputStream (JDK 1.0)

This class is a subclass of `InputStream` in which input data come from a specified array of `byte` values. This is useful when you want to read data in memory as if it were coming from a file or pipe or socket. Note that the specified array of bytes is not copied when a `ByteArrayInputStream` is created. See also `CharArrayReader`.

```
public class ByteArrayInputStream extends InputStream {
    // Public Constructors
            public ByteArrayInputStream(byte[] buf);
            public ByteArrayInputStream(byte[] buf, int offset, int length);
    // Protected Instance Variables
            protected byte[] buf;
            protected int count;
        1.1  protected int mark;
            protected int pos;
    // Public Instance Methods
            public synchronized int available();  // Overrides InputStream
        1.1  public void mark(int markpos);  // Overrides InputStream
        1.1  public boolean markSupported();  // Overrides InputStream
            public synchronized int read();  // Defines InputStream
            public synchronized int read(byte[] b, int off, int len);  // Overrides
InputStream
            public synchronized void reset();  // Overrides InputStream
            public synchronized long skip(long n);  // Overrides InputStream
}
```

Hierarchy:

Object->InputStream->ByteArrayInputStream

24.18 java.io.FileInputStream (JDK 1.0)

This class is a subclass of `InputStream` that reads bytes from a file specified by name or by a `File` or `FileDescriptor` object. `read()` reads a byte or array of bytes from the file. It returns -1 when the end of file has been reached. To read binary data, you typically use this class in conjunction with a `BufferedInputStream` and `DataInputStream`. To read text, you typically use it with an `InputStreamReader` and `BufferedReader`. Call `close()` to close the file when input is no longer needed.

```
public class FileInputStream extends InputStream {
    // Public Constructors
            public FileInputStream(String name) throws FileNotFoundException;
            public FileInputStream(File file) throws FileNotFoundException;
            public FileInputStream(FileDescriptor fdObj);
    // Public Instance Methods
            public native int available() throws IOException;  // Overrides 
InputStream
            public native void close() throws IOException;  // Overrides InputStream
            public final FileDescriptor getFD() throws IOException;
            public native int read() throws IOException;  // Defines InputStream
            public int read(byte[] b) throws IOException;  // Overrides InputStream
            public int read(byte[] b, int off, int len) throws IOException;  // 
Overrides InputStream
            public native long skip(long n) throws IOException;  // Overrides 
InputStream
    // Protected Instance Methods
            protected void finalize() throws IOException;  // Overrides Object
}
```

Hierarchy:

Object->InputStream->FileInputStream

24.24 java.io.FilterInputStream (JDK 1.0)

This class provides method definitions required to filter data obtained from the `InputStream` specified when the `FilterInputStream` is created. It must be subclassed to perform some sort of filtering operation, and may not be instantiated directly. See the subclasses `BufferedInputStream`, `DataInputStream`, and `PushbackInputStream`.

```
public class FilterInputStream extends InputStream {
    // Protected Constructor
            protected FilterInputStream(InputStream in);
    // Protected Instance Variables
            protected InputStream in;
    // Public Instance Methods
            public int available() throws IOException;  // Overrides InputStream
            public void close() throws IOException;  // Overrides InputStream
            public synchronized void mark(int readlimit);  // Overrides InputStream
            public boolean markSupported();  // Overrides InputStream
            public int read() throws IOException;  // Defines InputStream
            public int read(byte[] b) throws IOException;  // Overrides InputStream
            public int read(byte[] b, int off, int len) throws IOException;  // 
Overrides InputStream
            public synchronized void reset() throws IOException;  // Overrides 
InputStream
            public long skip(long n) throws IOException;  // Overrides InputStream
}
```

Hierarchy:

Object->InputStream->FilterInputStream

Extended By:

BufferedInputStream, CheckedInputStream, DataInputStream, InflaterInputStream, LineNumberInputStream, PushbackInputStream

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

24.28 java.io.InputStream (JDK 1.0)

This abstract class is the superclass of all input streams. It defines the basic input methods that all input stream classes provide.

`read()` reads a single byte or an array or subarray of bytes. It returns the byte read, or the number of bytes read, or -1 if the end-of-file has been reached. `skip()` skips a specified number of bytes of input. `available()` returns the number of bytes that can be read without blocking. `close()` closes the input stream and frees up any system resources associated with it. The stream should not be used after `close()` has been called.

If `markSupported()` returns `true` for a given `InputStream`, that stream supports `mark()` and `reset()` methods. `mark()` marks the current position in the input stream so that `reset()` can return to that position as long as no more than the specified number of bytes have been read between the `mark()` and `reset()`. See also `Reader`.

```
public abstract class InputStream extends Object {
    // Default Constructor: public InputStream()
    // Public Instance Methods
            public int available() throws IOException;
            public void close() throws IOException;
            public synchronized void mark(int readlimit);
            public boolean markSupported();
            public abstract int read() throws IOException;
            public int read(byte[] b) throws IOException;
            public int read(byte[] b, int off, int len) throws IOException;
            public synchronized void reset() throws IOException;
            public long skip(long n) throws IOException;
}
```

Extended By:

ByteArrayInputStream, FileInputStream, FilterInputStream, ObjectInputStream, PipedInputStream, SequenceInputStream, StringBufferInputStream

Passed To:

BufferedInputStream(), CheckedInputStream(), DataInputStream(), FilterInputStream(), GZIPInputStream(), InflaterInputStream(), InputStreamReader(), LineNumberInputStream(), ObjectInputStream(), Properties.load(), PropertyResourceBundle(), PushbackInputStream(), Runtime.getLocalizedInputStream(), SequenceInputStream(), StreamTokenizer(), System.setIn(), URLConnection.guessContentTypeFromStream(), ZipInputStream()

Returned By:

Class.getResourceAsStream(), ClassLoader.getResourceAsStream(), ClassLoader.getSystemResourceAsStream(), Process.getErrorStream(), Process.getInputStream(), Runtime.getLocalizedInputStream(), Socket.getInputStream(), SocketImpl.getInputStream(), URL.openStream(), URLConnection.getInputStream(), ZipFile.getInputStream()

Type Of:

FilterInputStream.in, System.in

PREVIOUS	HOME	NEXT
java.io.FilterWriter (JDK 1.1)	BOOK INDEX	java.io.InputStreamReader (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

24.34 java.io.LineNumberInputStream (JDK 1.0; Deprecated.)

This class is a `FilterInputStream` that keeps track of the number of lines of data that have been read. `getLineNumber()` returns the current line number. `setLineNumber()` sets the line number of the current line. Subsequent lines are numbered starting from that number.

This class is deprecated in Java 1.1 because it does not properly convert bytes to characters. Use `LineNumberReader` instead.

```
public class LineNumberInputStream extends FilterInputStream {
    // Public Constructor
            public LineNumberInputStream(InputStream in);
    // Public Instance Methods
            public int available() throws IOException;  // Overrides 
FilterInputStream
            public int getLineNumber();
            public void mark(int readlimit);  // Overrides FilterInputStream
            public int read() throws IOException;  // Overrides FilterInputStream
            public int read(byte[] b, int off, int len) throws IOException;  // 
Overrides FilterInputStream
            public void reset() throws IOException;  // Overrides FilterInputStream
            public void setLineNumber(int lineNumber);
            public long skip(long n) throws IOException;  // Overrides 
FilterInputStream
}
```

Hierarchy:

Object->InputStream->FilterInputStream->LineNumberInputStream

24.38 java.io.ObjectInput (JDK 1.1)

This interface extends the `DataInput` interface and adds methods for deserializing objects and reading bytes and arrays of bytes.

```
public abstract interface ObjectInput extends DataInput {
    // Public Instance Methods
            public abstract int available() throws IOException;
            public abstract void close() throws IOException;
            public abstract int read() throws IOException;
            public abstract int read(byte[] b) throws IOException;
            public abstract int read(byte[] b, int off, int len) throws IOException;
            public abstract Object readObject() throws ClassNotFoundException, 
IOException;
            public abstract long skip(long n) throws IOException;
}
```

Implemented By:

ObjectInputStream

Passed To:

Externalizable.readExternal()

24.39 java.io.ObjectInputStream (JDK 1.1)

`ObjectInputStream` is used to deserialize objects, arrays, and other values from a stream that was previously created with an `ObjectOutputStream`. The `readObject()` method deserializes objects and arrays (which should then be cast to the appropriate type); various other methods are used to read primitive data values from the stream. Note that only objects that implement the `Serializable` interface or the `Externalizable` interface can be serialized and deserialized.

The `defaultReadObject()` method may only be called from the `readObject()` method of an object that is currently being deserialized. It allows an object to perform additional processing after deserializing itself. The `registerValidation()` method may also only be called from the `readObject()` method of an object being deserialized. It registers an `ObjectInputValidation` object (typically the object being deserialized) to be notified when a complete tree of objects has been deserialized and the original call to the `readObject()` method of the `ObjectInputStream` is about to return to its caller.

The remaining methods include miscellaneous stream manipulation methods and several protected methods for use by subclasses that want to customize the deserialization behavior of `ObjectInputStream`.

```
public class ObjectInputStream extends InputStream implements ObjectInput {
    // Public Constructor
            public ObjectInputStream(InputStream in) throws IOException,
StreamCorruptedException;
    // Public Instance Methods
            public int available() throws IOException;  // Overrides InputStream
            public void close() throws IOException;  // Overrides InputStream
            public final void defaultReadObject() throws IOException,
ClassNotFoundException, NotActiveException;
            public int read() throws IOException;  // Defines InputStream
            public int read(byte[] data, int offset, int length) throws IOException;
// Overrides InputStream
            public boolean readBoolean() throws IOException;  // From DataInput
            public byte readByte() throws IOException;  // From DataInput
            public char readChar() throws IOException;  // From DataInput
            public double readDouble() throws IOException;  // From DataInput
            public float readFloat() throws IOException;  // From DataInput
            public void readFully(byte[] data) throws IOException;  // From DataInput
            public void readFully(byte[] data, int offset, int size) throws
IOException;  // From DataInput
            public int readInt() throws IOException;  // From DataInput
            public String readLine() throws IOException;  // From DataInput
            public long readLong() throws IOException;  // From DataInput
            public final Object readObject() throws OptionalDataException,
            public final Object readObject() 'u'ClassNotFoundException, IOException;
// From ObjectInput
            public short readShort() throws IOException;  // From DataInput
            public String readUTF() throws IOException;  // From DataInput
            public int readUnsignedByte() throws IOException;  // From DataInput
            public int readUnsignedShort() throws IOException;  // From DataInput
            public synchronized void registerValidation(ObjectInputValidation obj,
int prio)
            public synchronized void registerValidation'u'throws NotActiveException,
InvalidObjectException;
            public int skipBytes(int len) throws IOException;  // From DataInput
    // Protected Instance Methods
            protected final boolean enableResolveObject(boolean enable) throws
SecurityException;
            protected void readStreamHeader() throws IOException,
StreamCorruptedException;
            protected Class resolveClass(ObjectStreamClass v) throws IOException,
ClassNotFoundException;
            protected Object resolveObject(Object obj) throws IOException;
}
```

Hierarchy:

Object->InputStream->ObjectInputStream(ObjectInput(DataInput))

PREVIOUS HOME NEXT

java.io.ObjectInput (JDK 1.1) BOOK INDEX java.io.ObjectInputValidation (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

24.48 java.io.PipedInputStream (JDK 1.0)

This class is an `InputStream` that implements one-half of a pipe, and is useful for communication between threads. A `PipedInputStream` must be connected to a `PipedOutputStream` object, which may be specified when the `PipedInputStream` is created or with the `connect()` method. Data read from a `PipedInputStream` object are received from the `PipedOutputStream` to which it is connected.

See `InputStream` for information on the low-level methods for reading data from a `PipedInputStream`. A `FilterInputStream` may be used to provide a higher-level interface for reading data from a `PipedInputStream`.

```
public class PipedInputStream extends InputStream {
    // Public Constructors
            public PipedInputStream(PipedOutputStream src) throws IOException;
            public PipedInputStream();
    // Constants
        1.1  protected static final int PIPE_SIZE;
    // Protected Instance Variables
        1.1  protected byte[] buffer;
        1.1  protected int in;
        1.1  protected int out;
    // Public Instance Methods
            public synchronized int available() throws IOException;  // Overrides 
InputStream
            public void close() throws IOException;  // Overrides InputStream
            public void connect(PipedOutputStream src) throws IOException;
            public synchronized int read() throws IOException;  // Defines 
InputStream
            public synchronized int read(byte[] b, int off, int len) throws 
IOException;  // Overrides InputStream
    // Protected Instance Methods
        1.1  protected synchronized void receive(int b) throws IOException;
}
```

Hierarchy:

Object->InputStream->PipedInputStream

Passed To:

PipedOutputStream(), PipedOutputStream.connect()

PREVIOUS | HOME | NEXT

java.io.OutputStreamWriter (JDK 1.1) | BOOK INDEX | java.io.PipedOutputStream (JDK 1.0)

24.54 java.io.PushbackInputStream (JDK 1.0)

This class is a `FilterInputStream` that implements a one-byte pushback buffer or, in Java 1.1, a pushback buffer of a specified length. The `unread()` methods "push" bytes back into the stream--these bytes are the first ones read by the next call to a `read()` method. This class is sometimes useful when writing parsers.

See also `PushbackReader`.

```
public class PushbackInputStream extends FilterInputStream {
    // Public Constructors
        1.1  public PushbackInputStream(InputStream in, int size);
            public PushbackInputStream(InputStream in);
    // Protected Instance Variables
        1.1  protected byte[] buf;
        1.1  protected int pos;
    // Public Instance Methods
            public int available() throws IOException;  // Overrides 
FilterInputStream
            public boolean markSupported();  // Overrides FilterInputStream
            public int read() throws IOException;  // Overrides FilterInputStream
            public int read(byte[] b, int off, int len) throws IOException;  // 
Overrides FilterInputStream
            public void unread(int b) throws IOException;
        1.1  public void unread(byte[] b, int off, int len) throws IOException;
        1.1  public void unread(byte[] b) throws IOException;
}
```

Hierarchy:

Object->InputStream->FilterInputStream->PushbackInputStream

24.58 java.io.SequenceInputStream (JDK 1.0)

This class provides a way of seamlessly concatenating the data from two or more input streams. It provides an `InputStream` interface to a sequence of `InputStream` objects. Data are read from the streams in the order in which the streams are specified. When the end of one stream is reached, data are automatically read from the next stream. This class might be useful, for example, when implementing an "include file" facility for a parser of some sort.

```
public class SequenceInputStream extends InputStream {
    // Public Constructors
            public SequenceInputStream(Enumeration e);
            public SequenceInputStream(InputStream s1, InputStream s2);
    // Public Instance Methods
        1.1  public int available() throws IOException;  // Overrides InputStream
            public void close() throws IOException;  // Overrides InputStream
            public int read() throws IOException;  // Defines InputStream
            public int read(byte[] buf, int pos, int len) throws IOException;  //
Overrides InputStream
}
```

Hierarchy:

Object->InputStream->SequenceInputStream

24.62 java.io.StringBufferInputStream (JDK 1.0; Deprecated.)

This class is a subclass of `InputStream` in which input bytes come from the characters of a specified `String` object.

This class does not correctly convert the characters of a `StringBuffer` into bytes, and is deprecated in Java 1.1. Use `StringReader` instead to correctly convert characters into bytes, or use `ByteArrayInputStream` to read bytes from an array of bytes.

```
public class StringBufferInputStream extends InputStream {
    // Public Constructor
            public StringBufferInputStream(String s);
    // Protected Instance Variables
            protected String buffer;
            protected int count;
            protected int pos;
    // Public Instance Methods
            public synchronized int available();  // Overrides InputStream
            public synchronized int read();  // Defines InputStream
            public synchronized int read(byte[] b, int off, int len);  // Overrides
InputStream
            public synchronized void reset();  // Overrides InputStream
            public synchronized long skip(long n);  // Overrides InputStream
}
```

Hierarchy:

Object->InputStream->StringBufferInputStream

23.23 java.beans.Visibility (JDK 1.1)

This interface is intended to be implemented by advanced beans that may run either with or without a GUI (graphical user interface) present. The methods it defines allow a bean to specify whether it requires a GUI and allows the environment to notify the bean whether a GUI is available.

If a bean absolutely requires a GUI, it should return `true` from `needsGui()`. If a bean is running without a GUI, it should return `true` from `avoidingGui()`. If no GUI is available, the bean may be notified through a call to `dontUseGui()`, and if a GUI is available, the bean may be notified through a call to `okToUseGui()`.

```
public abstract interface Visibility {
    // Public Instance Methods
            public abstract boolean avoidingGui();
            public abstract void dontUseGui();
            public abstract boolean needsGui();
            public abstract void okToUseGui();
}
```

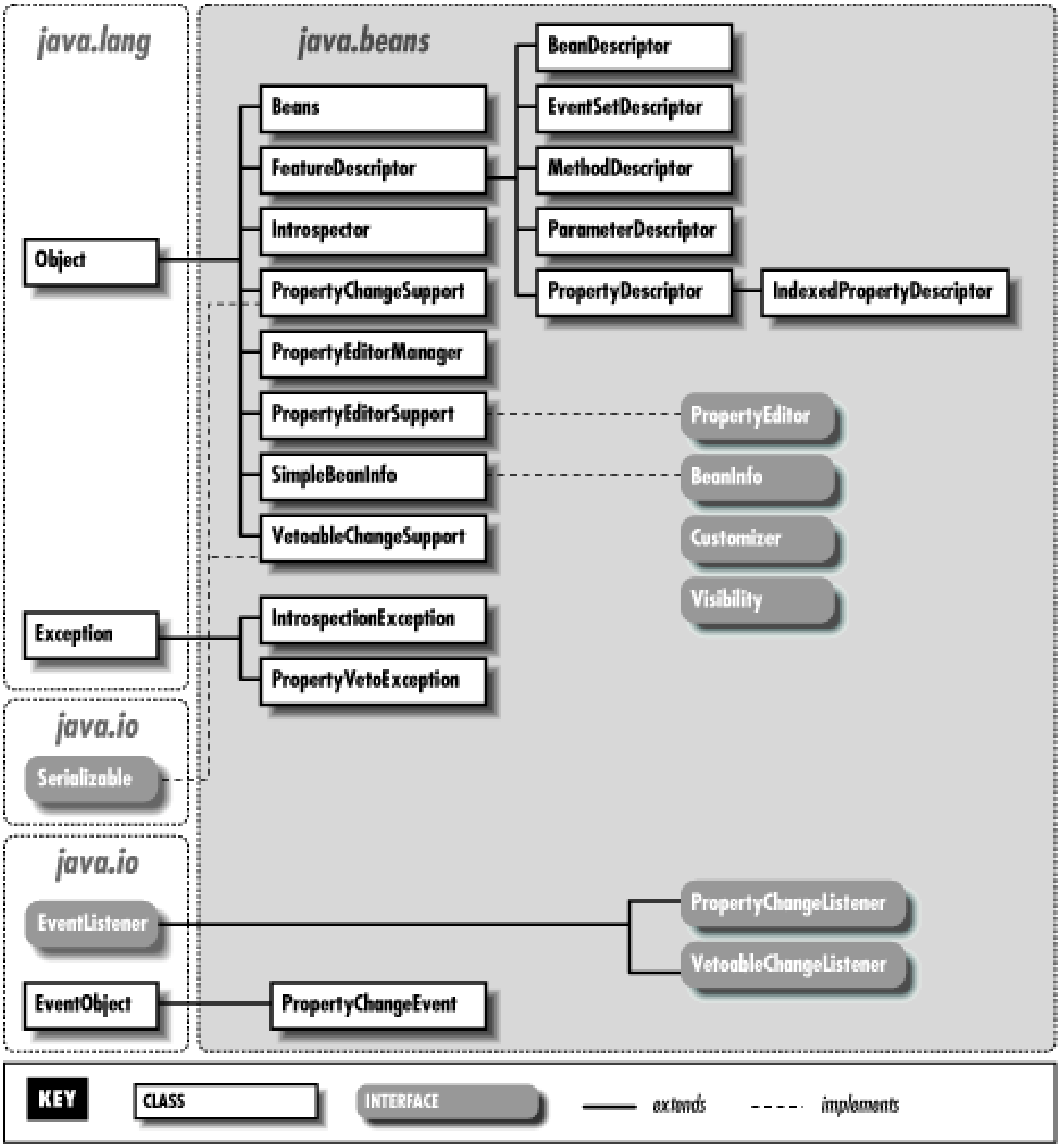
23. The java.beans Package

Contents:

The `java.beans` package contains the classes and interfaces that constitute the JavaBeans API. It is new in Java 1.1. Figure 23.1 shows the class hierarchy for this package.

The classes and interfaces in this package are useful to programmers who are developing "beanbox" tools to manipulate beans, and to programmers who are writing their own beans. Programmers who are building applications using beans do not have to be familiar with java.beans. See Chapter 10, *Java Beans* for more details on writing custom beans.

Figure 23.1: The java.beans package



23.1 java.beans.BeanDescriptor (JDK 1.1)

A BeanDescriptor object is a type of FeatureDescriptor that describes a Java bean. The BeanInfo class for a Java bean optionally creates and initializes a BeanDescriptor object to describe the bean. Typically, only application builders and similar tools use the BeanDescriptor.

To create a BeanDescriptor, you must specify the class of the bean, and optionally, the class of a Customizer for the bean. You can use the methods of FeatureDescriptor to provide additional information about the bean.

```
public class BeanDescriptor extends FeatureDescriptor {
    // Public Constructors
            public BeanDescriptor(Class beanClass);
            public BeanDescriptor(Class beanClass, Class customizerClass);
    // Public Instance Methods
            public Class getBeanClass();
            public Class getCustomizerClass();
}
```

Hierarchy:

Object->FeatureDescriptor->BeanDescriptor

Returned By:

BeanInfo.getBeanDescriptor(), SimpleBeanInfo.getBeanDescriptor()

18.60 java.awt.Toolkit (JDK 1.0)

This abstract class defines methods that, when implemented, create platform-dependent "peers" for each of the `java.awt` `Component` types. Java supports its platform-independent GUI interface by implementing a subclass of `Toolkit` for each platform. Portable programs should never use these methods to create peers directly--they should use the `Component` classes themselves. A `Toolkit` object cannot be instantiated directly. `Component.getToolkit()` returns the `Toolkit` being used for a particular component.

The `Toolkit` class defines a few methods that you can use directly: the static method `getDefaultToolkit()` returns the default `Toolkit` that is in use. `getScreenSize()` returns the screen size in pixels, and `getScreenResolution()` returns the resolution in dots-per-inch. `getFontList()` returns the names of supported fonts. `sync()` flushes all pending graphics output, which can be useful for animation. In Java 1.1, `getPrintJob()`, `getSystemClipboard()`, and `getSystemEventQueue()` are also of interest.

```
public abstract class Toolkit extends Object {
    // Default Constructor: public Toolkit()
    // Class Methods
            public static synchronized Toolkit getDefaultToolkit();
        1.1  protected static Container getNativeContainer(Component c);
        1.1  public static String getProperty(String key, String defaultValue);
    // Public Instance Methods
        1.1  public abstract void beep();
            public abstract int checkImage(Image image, int width, int height, 
ImageObserver observer);
            public abstract Image createImage(ImageProducer producer);
        1.1  public Image createImage(byte[] imagedata);
        1.1  public abstract Image createImage(byte[] imagedata, int imageoffset, int 
imagelength);
            public abstract ColorModel getColorModel();
            public abstract String[] getFontList();
            public abstract FontMetrics getFontMetrics(Font font);
            public abstract Image getImage(String filename);
            public abstract Image getImage(URL url);
        1.1  public int getMenuShortcutKeyMask();
        1.1  public abstract PrintJob getPrintJob(Frame frame, String jobtitle, 
Properties props);
            public abstract int getScreenResolution();
            public abstract Dimension getScreenSize();
        1.1  public abstract Clipboard getSystemClipboard();
        1.1  public final EventQueue getSystemEventQueue();
            public abstract boolean prepareImage(Image image, int width, int height, 
ImageObserver observer);
            public abstract void sync();
    // Protected Instance Methods
            protected abstract ButtonPeer createButton(Button target);
            protected abstract CanvasPeer createCanvas(Canvas target);
            protected abstract CheckboxPeer createCheckbox(Checkbox target);
            protected abstract CheckboxMenuItemPeer
createCheckboxMenuItem(CheckboxMenuItem target);
            protected abstract ChoicePeer createChoice(Choice target);
        1.1  protected LightweightPeer createComponent(Component target);
            protected abstract DialogPeer createDialog(Dialog target);
            protected abstract FileDialogPeer createFileDialog(FileDialog target);
            protected abstract FramePeer createFrame(Frame target);
            protected abstract LabelPeer createLabel(Label target);
            protected abstract ListPeer createList(List target);
            protected abstract MenuPeer createMenu(Menu target);
            protected abstract MenuBarPeer createMenuBar(MenuBar target);
            protected abstract MenuItemPeer createMenuItem(MenuItem target);
            protected abstract PanelPeer createPanel(Panel target);
        1.1  protected abstract PopupMenuPeer createPopupMenu(PopupMenu target);
        1.1  protected abstract ScrollPanePeer createScrollPane(ScrollPane target);
            protected abstract ScrollbarPeer createScrollbar(Scrollbar target);
            protected abstract TextAreaPeer createTextArea(TextArea target);
            protected abstract TextFieldPeer createTextField(TextField target);
            protected abstract WindowPeer createWindow(Window target);
        1.1  protected abstract FontPeer getFontPeer(String name, int style);
        1.1  protected abstract EventQueue getSystemEventQueueImpl();
        1.1  protected void loadSystemColors(int[] systemColors);
}
```

Returned By:

`Component.getToolkit(), ComponentPeer.getToolkit(), Toolkit.getDefaultToolkit(), Window.getToolkit()`

22.7 java.awt.peer.ContainerPeer (JDK 1.0)

```
public abstract interface ContainerPeer extends ComponentPeer {
    // Public Instance Methods
        1.1  public abstract void beginValidate();
        1.1  public abstract void endValidate();
        1.1  public abstract Insets getInsets();
            public abstract Insets insets();
}
```

Extended By:

PanelPeer, ScrollPanePeer, WindowPeer

31.7 java.util.zip.Deflater (JDK 1.1)

This class implements the general ZLIB data compression algorithm used by the *gzip* and *PKZip* compression programs. The constants defined by this class are used to specify the compression strategy to be used and the compression speed/strength trade-off level to be used. If you set the *nowrap* argument to the constructor to `true`, then the ZLIB header and checksum data are omitted from the compressed output, which is the format that both *gzip* and *PKZip* use.

The important methods of this class are `setInput()`, which specifies input data to be compressed, and `deflate()`, which compresses the data and returns the compressed output. The remaining methods exist so that `Deflater` can be used for stream-based compression, as it is in higher-level classes such as `GZIPOutputStream` and `ZipOutputStream`. These stream classes are sufficient in most cases. Most applications do not need to use `Deflater` directly.

The `Inflater` class uncompresses data compressed with a `Deflater` object.

```
public class Deflater extends Object {
    // Public Constructors
            public Deflater(int level, boolean nowrap);
            public Deflater(int level);
            public Deflater();
    // Constants
            public static final int BEST_COMPRESSION;
            public static final int BEST_SPEED;
            public static final int DEFAULT_COMPRESSION;
            public static final int DEFAULT_STRATEGY;
            public static final int DEFLATED;
            public static final int FILTERED;
            public static final int HUFFMAN_ONLY;
            public static final int NO_COMPRESSION;
    // Public Instance Methods
            public synchronized native int deflate(byte[] b, int off, int len);
            public int deflate(byte[] b);
            public synchronized native void end();
            public synchronized void finish();
            public synchronized boolean finished();
            public synchronized native int getAdler();
            public synchronized native int getTotalIn();
            public synchronized native int getTotalOut();
            public boolean needsInput();
            public synchronized native void reset();
            public synchronized native void setDictionary(byte[] b, int off, int 
len);
            public void setDictionary(byte[] b);
            public synchronized void setInput(byte[] b, int off, int len);
            public void setInput(byte[] b);
            public synchronized void setLevel(int Level);
            public synchronized void setStrategy(int strategy);
    // Protected Instance Methods
            protected void finalize();  // Overrides Object
}
```

Passed To:

DeflaterOutputStream()

Type Of:

DeflaterOutputStream.def

PREVIOUS | HOME | NEXT

java.util.zip.DataFormatException (JDK 1.1) | BOOK INDEX | java.util.zip.DeflaterOutputStream (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

28.7 java.net.DatagramSocketImpl (JDK 1.1)

This abstract class defines the methods necessary to implement communication through datagram and multicast sockets. System programmers may create subclasses of this class when they need to implement datagram or multicast sockets in a nonstandard network environment, such as behind a firewall or on a network that uses a nonstandard transport protocol.

Normal applications never need to use or subclass this class.

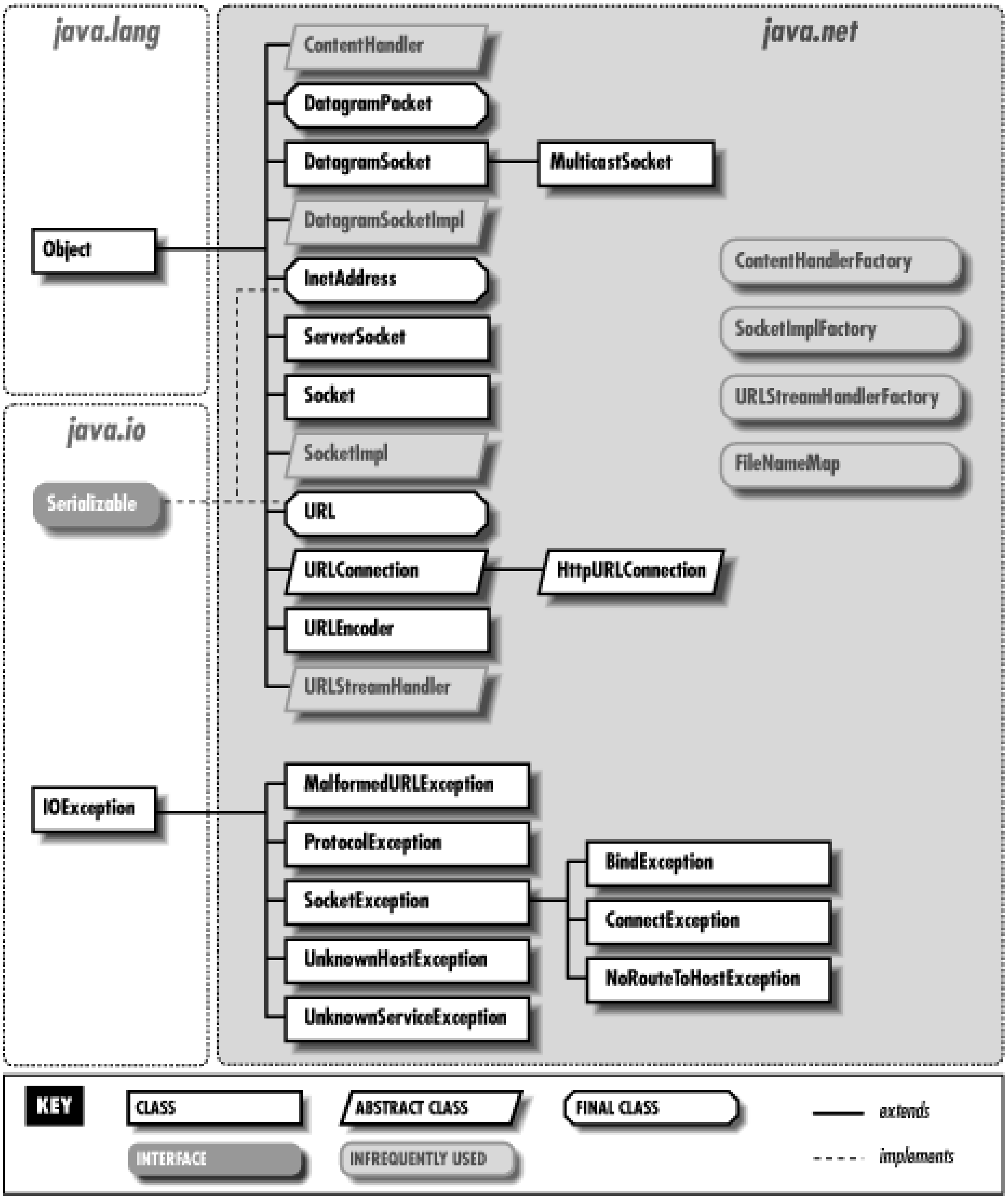
```
public abstract class DatagramSocketImpl extends Object {
    // Default Constructor: public DatagramSocketImpl()
    // Protected Instance Variables
            protected FileDescriptor fd;
            protected int localPort;
    // Protected Instance Methods
            protected abstract void bind(int lport, InetAddress laddr) throws 
SocketException;
            protected abstract void close();
            protected abstract void create() throws SocketException;
            protected FileDescriptor getFileDescriptor();
            protected int getLocalPort();
            protected abstract byte getTTL() throws IOException;
            protected abstract void join(InetAddress inetaddr) throws IOException;
            protected abstract void leave(InetAddress inetaddr) throws IOException;
            protected abstract int peek(InetAddress i) throws IOException;
            protected abstract void receive(DatagramPacket p) throws IOException;
            protected abstract void send(DatagramPacket p) throws IOException;
            protected abstract void setTTL(byte ttl) throws IOException;
}
```

28. The java.net Package

Contents:

The `java.net` package provides a powerful and flexible infrastructure for networking. Figure 28.1 shows the class hierarchy for this package. Many of the classes in this package are part of the networking infrastructure and are not used by normal applications; these complicated classes can make the package a difficult one to understand. In this overview we describe only the classes that an application might normally use.

Figure 28.1: The java.net package



The `URL` class represents an Internet Uniform Resource Locator. It provides a very simple interface to networking--the object referred to by the URL can be downloaded with a single call, or streams may be opened to read from or write to the object.

At a slightly more complex level, the `URLConnection` object may be obtained from a given `URL` object. The `URLConnection` class provides additional methods that allow you to work with URLs in more sophisticated ways.

If you want to do more than simply download an object referenced by a URL, you can do your own networking with the `Socket` class. This class allows you to connect to a specified port on a specified Internet host and read and write data using the `InputStream` and `OutputStream` classes of the `java.io` package. If you want to implement a server to accept connections from clients, you can use the related `ServerSocket` class. Both `Socket` and `ServerSocket` use the `InetAddress` address class, which represents an Internet address.

The `java.net` package allows you to do low-level networking with `DatagramPacket` objects which may be sent and received over the network through a `DatagramSocket` object. In Java 1.1, the package has been extended to include a `MulticastSocket` class that supports multicast networking.

28.1 java.net.BindException (JDK 1.1)

This exception signals that a socket could not be bound to a local address and port. This often means that the port is already in use.

```
public class BindException extends SocketException {
    // Public Constructors
            public BindException(String msg);
            public BindException();
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->IOException->SocketException->BindException

18.14 java.awt.Color (JDK 1.0)

The `Color` object describes a color. The `Color()` constructors describe a color as red, green, and blue components between 0 and 255, or as floating-point values between 0.0 and 1.0. The class method `Color.getHSBColor()` creates a color using the hue/saturation/brightness color model.

`brighter()` and `darker()` are useful methods to create shading effects. The `getColor()` methods look up a color name in the properties database and convert the resulting integer color value into a `Color` object. Two of these methods provide a default value to be used in case the specified color name is not found.

```
public class Color extends Object implements Serializable {
    // Public Constructors
            public Color(int r, int g, int b);
            public Color(int rgb);
            public Color(float r, float g, float b);
    // Constants
            public static final Color black;
            public static final Color blue;
            public static final Color cyan;
            public static final Color darkGray;
            public static final Color gray;
            public static final Color green;
            public static final Color lightGray;
            public static final Color magenta;
            public static final Color orange;
            public static final Color pink;
            public static final Color red;
            public static final Color white;
            public static final Color yellow;
    // Class Methods
            public static int HSBtoRGB(float hue, float saturation, float
brightness);
            public static float[] RGBtoHSB(int r, int g, int b, float[] hsbvals);
        1.1 public static Color decode(String nm) throws NumberFormatException;
            public static Color getColor(String nm);
            public static Color getColor(String nm, Color v);
            public static Color getColor(String nm, int v);
            public static Color getHSBColor(float h, float s, float b);
    // Public Instance Methods
```

```
            public Color brighter();
            public Color darker();
            public boolean equals(Object obj);  // Overrides Object
            public int getBlue();
            public int getGreen();
            public int getRGB();
            public int getRed();
            public int hashCode();  // Overrides Object
            public String toString();  // Overrides Object
}
```

Extended By:

`SystemColor`

Passed To:

`Color.getColor(), Component.setBackground(), Component.setForeground(), ComponentPeer.setBackground(), ComponentPeer.setForeground(), Graphics.drawImage(), Graphics.setColor(), Graphics.setXORMode()`

Returned By:

`Color.brighter(), Color.darker(), Color.decode(), Color.getColor(), Color.getHSBColor(), Component.getBackground(), Component.getForeground(), Graphics.getColor()`

Type Of:

`Color.black, Color.blue, Color.cyan, Color.darkGray, Color.gray, Color.green, Color.lightGray, Color.magenta, Color.orange, Color.pink, Color.red, Color.white, Color.yellow`

PREVIOUS | HOME | NEXT

java.awt.Choice (JDK 1.0) | BOOK INDEX | java.awt.Component (JDK 1.0)

18.24 java.awt.Font (JDK 1.0)

This class represents a font in a platform-independent way. The constructor accepts a font name, style, and point size. In Java 1.0, supported font names are: "TimesRoman," "Helvetica," "Courier," "Dialog," and "DialogInput." In Java 1.1, "Serif," "SansSerif," and "Monospaced" should be used in preference to the first three names. The style may be one of the constants `PLAIN`, `BOLD`, or `ITALIC`, or the sum `BOLD+ITALIC`.

The class method `getFont()` looks up the specified name in the system properties list and returns the font specified as the value of that property. It takes an optional `Font` default to use if the named font property is not found. This allows user customizability.

Use the `FontMetrics` class if you need to know how tall a font is or how wide a string drawn using that font will be.

```
public class Font extends Object implements Serializable {
    // Public Constructor
            public Font(String name, int style, int size);
    // Constants
            public static final int BOLD;
            public static final int ITALIC;
            public static final int PLAIN;
    // Protected Instance Variables
            protected String name;
            protected int size;
            protected int style;
    // Class Methods
        1.1  public static Font decode(String str);
            public static Font getFont(String nm);
            public static Font getFont(String nm, Font font);
    // Public Instance Methods
            public boolean equals(Object obj);  // Overrides Object
            public String getFamily();
            public String getName();
        1.1  public FontPeer getPeer();
            public int getSize();
            public int getStyle();
            public int hashCode();  // Overrides Object
            public boolean isBold();
            public boolean isItalic();
            public boolean isPlain();
            public String toString();  // Overrides Object
}
```

Passed To:

`Component.getFontMetrics(), Component.setFont(), ComponentPeer.getFontMetrics(), ComponentPeer.setFont(), Font.getFont(), FontMetrics(), Graphics.getFontMetrics(), Graphics.setFont(), MenuComponent.setFont(), Toolkit.getFontMetrics()`

Returned By:

`Component.getFont(), Font.decode(), Font.getFont(), FontMetrics.getFont(), Graphics.getFont(), MenuComponent.getFont(), MenuContainer.getFont()`

Type Of:

`FontMetrics.font`

25.5 java.lang.Boolean (JDK 1.0)

This class provides an immutable object wrapper around the `boolean` primitive type. Note that the `TRUE` and `FALSE` constants are `Boolean` objects; they are not the same as the `true` and `false` `boolean` values. In Java 1.1, this class defines a `Class` constant that represents the `boolean` type.

`booleanValue()` returns the `boolean` value of a `Boolean` object. The class method `getBoolean()` retrieves the `boolean` value of a named property from the system property list. The class method `valueOf()` parses a string and returns the `Boolean` value it represents.

```
public final class Boolean extends Object implements Serializable {
    // Public Constructors
            public Boolean(boolean value);
            public Boolean(String s);
    // Constants
            public static final Boolean FALSE;
            public static final Boolean TRUE;
        1.1public static final Class TYPE;
    // Class Methods
            public static boolean getBoolean(String name);
            public static Boolean valueOf(String s);
    // Public Instance Methods
            public boolean booleanValue();
            public boolean equals(Object obj);  // Overrides Object
            public int hashCode();  // Overrides Object
            public String toString();  // Overrides Object
}
```

Returned By:

Boolean.valueOf()

Type Of:

Boolean.FALSE, Boolean.TRUE

PREVIOUS HOME NEXT

java.lang.ArrayStoreException (JDK 1.0) BOOK INDEX java.lang.Byte (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

18.33 java.awt.Insets (JDK 1.0)

This class holds four values that represent the top, left, bottom, and right margins, in pixels, of a `Container` or other `Component`. Objects of this type may be specified in a `GridBagConstraints` layout object, and are returned by `Container.insets()`, which queries the margins of a container.

```
public class Insets extends Object implements Cloneable, Serializable {
    // Public Constructor
            public Insets(int top, int left, int bottom, int right);
    // Public Instance Variables
            public int bottom;
            public int left;
            public int right;
            public int top;
    // Public Instance Methods
            public Object clone();  // Overrides Object
        1.1  public boolean equals(Object obj);  // Overrides Object
            public String toString();  // Overrides Object
}
```

Returned By:

`Container.getInsets(), Container.insets(), ContainerPeer.getInsets(), ContainerPeer.insets()`

Type Of:

`GridBagConstraints.insets`

java.awt.Image (JDK 1.0) BOOK INDEX java.awt.ItemSelectable (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

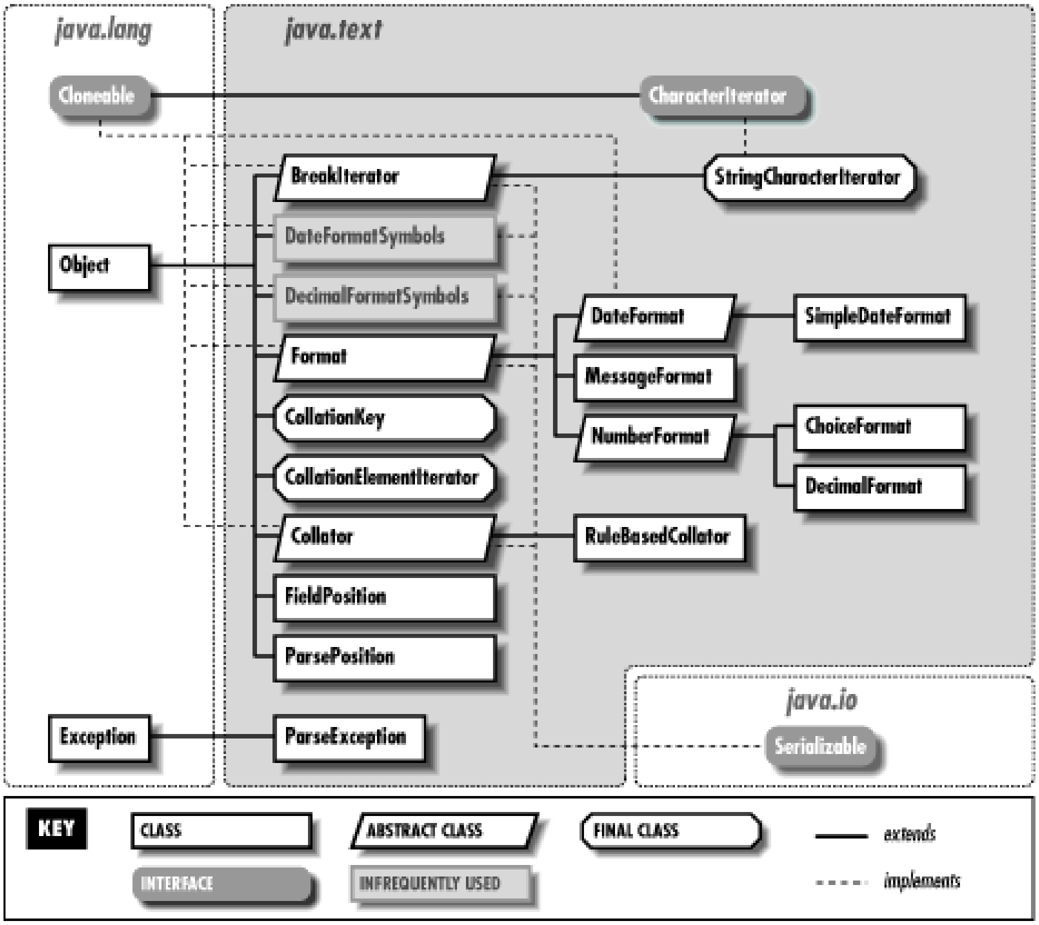
29. The java.text Package

Contents:

The `java.text` package consists of classes and interfaces that are useful for writing internationalized programs that handle local customs, such as date and time formatting and string alphabetization, correctly. This package is new in Java 1.1. Figure 29.1 shows its class hierarchy.

The `NumberFormat` class formats numbers, monetary quantities, and percentages as appropriate for the default or specified locale. `DateFormat` formats dates and times in a locale-specific way. The concrete `DecimalFormat` and `SimpleDateFormat` subclasses of these classes can be used for customized number, date, and time formatting. `MessageFormat` allows substitution of dynamic values, including formatted numbers and dates, into static message strings. `ChoiceFormat` formats a number using an enumerated set of string values. `Collator` compares strings according to the customary sorting order for a locale. `BreakIterator` scans text to find word, line, and sentence boundaries following locale-specfic rules.

Figure 29.1: The java.text package



29.1 java.text.BreakIterator (JDK 1.1)

This class is used to determine character, word, sentence, and line breaks in a block of text in a way that is independent of locale and text-encoding. As an abstract class, `BreakIterator` cannot be instantiated directly. Instead, you must use one of the class methods `getCharacterInstance()`, `getWordInstance()`, `getSentenceInstance()`, or `getLineInstance()` to return an instance of a nonabstract subclass of `BreakIterator`. These various "factory" methods return a `BreakIterator` object that is configured to locate the requested boundary types, and is localized to work for the optionally specified locale.

Once you have obtained an appropriate `BreakIterator` object, you use `setText()` to specify the text that it is to locate boundaries in. To locate boundaries in a Java `String` object, simply specify the string. To locate boundaries in text that uses some other encoding, you must specify a `CharacterIterator` object for that text so that the `BreakIterator` object can locate the individual characters of the text.

Having set the text to be searched, you can determine the character positions of characters, words, sentences, or line breaks with the `first()`, `last()`, `next()`, `previous()`, `current()`, and `following()` methods, which perform the obvious functions. Note that these methods do not return text itself, but merely the position of the appropriate word, sentence, or line break.

```
public abstract class BreakIterator extends Object implements Cloneable, Serializable
{
    // Protected Constructor
            protected BreakIterator();
    // Constants
            public static final int DONE;
    // Class Methods
            public static synchronized Locale[] getAvailableLocales();
            public static BreakIterator getCharacterInstance();
            public static BreakIterator getCharacterInstance(Locale where);
            public static BreakIterator getLineInstance();
            public static BreakIterator getLineInstance(Locale where);
            public static BreakIterator getSentenceInstance();
            public static BreakIterator getSentenceInstance(Locale where);
            public static BreakIterator getWordInstance();
            public static BreakIterator getWordInstance(Locale where);
    // Public Instance Methods
            public Object clone();  // Overrides Object
            public abstract int current();
            public abstract int first();
            public abstract int following(int offset);
            public abstract CharacterIterator getText();
            public abstract int last();
            public abstract int next(int n);
            public abstract int next();
            public abstract int previous();
            public void setText(String newText);
            public abstract void setText(CharacterIterator newText);
}
```

Returned By:

BreakIterator.getCharacterInstance(), BreakIterator.getLineInstance(), BreakIterator.getSentenceInstance(), BreakIterator.getWordInstance()

PREVIOUS | HOME | NEXT
java.net.URLStreamHandlerFactory (JDK 1.0) | BOOK INDEX | java.text.CharacterIterator (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

24.2 java.io.BufferedOutputStream (JDK 1.0)

This class is a `FilterOutputStream` that provides output data buffering--output efficiency is increased by storing values to be written in a buffer and actually writing them out only when the buffer fills up or when the `flush()` method is called. Create a `BufferedOutputStream` by specifying the `OutputStream` that is to have buffering applied in the call to the constructor. See also `BufferedWriter`.

```
public class BufferedOutputStream extends FilterOutputStream {
    // Public Constructors
            public BufferedOutputStream(OutputStream out);
            public BufferedOutputStream(OutputStream out, int size);
    // Protected Instance Variables
            protected byte[] buf;
            protected int count;
    // Public Instance Methods
            public synchronized void flush() throws IOException;  // Overrides 
FilterOutputStream
            public synchronized void write(int b) throws IOException;  // Overrides 
FilterOutputStream
            public synchronized void write(byte[] b, int off, int len) throws 
IOException;  // Overrides FilterOutputStream
}
```

Hierarchy:

Object->OutputStream->FilterOutputStream->BufferedOutputStream

24.6 java.io.ByteArrayOutputStream (JDK 1.0)

This class is a subclass of `OutputStream` in which data that are written are stored in an internal `byte` array. The internal array grows as necessary, and can be retrieved with `toByteArray()` or `toString()`. The `reset()` method discards any data currently stored in the internal array and begins storing data from the beginning again. See also `CharArrayWriter`.

```
public class ByteArrayOutputStream extends OutputStream {
    // Public Constructors
            public ByteArrayOutputStream();
            public ByteArrayOutputStream(int size);
    // Protected Instance Variables
            protected byte[] buf;
            protected int count;
    // Public Instance Methods
            public synchronized void reset();
            public int size();
            public synchronized byte[] toByteArray();
            public String toString();  // Overrides Object
        1.1  public String toString(String enc) throws UnsupportedEncodingException;
        #   public String toString(int hibyte);
            public synchronized void write(int b);  // Defines OutputStream
            public synchronized void write(byte[] b, int off, int len);  // Overrides 
OutputStream
            public synchronized void writeTo(OutputStream out) throws IOException;
}
```

Hierarchy:

Object->OutputStream->ByteArrayOutputStream

24.7 java.io.CharArrayReader (JDK 1.1)

This class is a character input stream that uses a character array as the source of the characters it returns. You create a `CharArrayReader` by specifying the character array, or portion of an array, that it is to read from.

`CharArrayReader` defines the usual `Reader` methods, and supports the `mark()` and `reset()` methods.

Note that the character array you pass to the `CharArrayReader` is not copied by this class. This means that changes you make to the elements of the array after you create the input stream do affect the values read from the array.

`CharArrayReader()` is the character-array analog of `ByteArrayInputStream`, and is similar to `StringReader`.

```
public class CharArrayReader extends Reader {
    // Public Constructors
            public CharArrayReader(char[] buf);
            public CharArrayReader(char[] buf, int offset, int length);
    // Protected Instance Variables
            protected char[] buf;
            protected int count;
            protected int markedPos;
            protected int pos;
    // Public Instance Methods
            public void close();  // Defines Reader
            public void mark(int readAheadLimit) throws IOException;  // Overrides
Reader
            public boolean markSupported();  // Overrides Reader
            public int read() throws IOException;  // Overrides Reader
            public int read(char[] b, int off, int len) throws IOException;  //
Defines Reader
            public boolean ready() throws IOException;  // Overrides Reader
            public void reset() throws IOException;  // Overrides Reader
            public long skip(long n) throws IOException;  // Overrides Reader
}
```

Hierarchy:

Object->Reader->CharArrayReader

24.8 java.io.CharArrayWriter (JDK 1.1)

This class is a character output stream that uses an internal character array as the destination of characters written to it. When you create a `CharArrayWriter`, you may optionally specify an initial size for the character array, but you do not specify the character array itself--this array is managed internally by the `CharArrayWriter`, and grows as necessary to accommodate all the characters written to it. The `toString()` and `toCharArray()` methods return a copy of all characters written to the stream, either as a string or as an array of characters.

`CharArrayWriter` defines the standard `write()`, `flush()`, and `close()` methods that all `Writer` subclasses do. It also defines a few other useful methods. `size()` returns the number of characters that have been written to the stream. `reset()` resets the stream to its initial state, with an empty character array; this is more efficient than creating a new `CharArrayWriter`. Finally, `writeTo()` writes the contents of the internal character array to some other specified character stream.

`CharArrayWriter` is the character-stream analog of `ByteArrayOutputStream`, and is quite similar to `StringWriter`.

```
public class CharArrayWriter extends Writer {
    // Public Constructors
            public CharArrayWriter();
            public CharArrayWriter(int initialSize);
    // Protected Instance Variables
            protected char[] buf;
            protected int count;
    // Public Instance Methods
            public void close();  // Defines Writer
            public void flush();  // Defines Writer
            public void reset();
            public int size();
            public char[] toCharArray();
            public String toString();  // Overrides Object
            public void write(int c);  // Overrides Writer
            public void write(char[] c, int off, int len);  // Defines Writer
            public void write(String str, int off, int len);  // Overrides Writer
            public void writeTo(Writer out) throws IOException;
}
```

Hierarchy:

Object->Writer->CharArrayWriter

PREVIOUS | HOME | NEXT
java.io.CharArrayReader (JDK 1.1) | BOOK INDEX | java.io.CharConversionException (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

31.8 java.util.zip.DeflaterOutputStream (JDK 1.1)

This class is a subclass of `java.io.FilterOutputStream`; it "filters" a stream of data by compressing ("deflating") it and then writes the compressed data to another output stream. To create a `DeflaterOutputStream`, you must specify the stream that it is to write to, and also a `Deflater` object that is to perform the compression. You can set various options on the `Deflater` object to specify just what type of compression is to be performed. Once a `DeflaterOutputStream` is created, its `write()` and `close()` methods are the same as those of other output streams.

The `InflaterInputStream` class can be used to read data written with the `DeflaterOutputStream`.

Note that a `DeflaterOutputStream` writes raw compressed data; applications often prefer one of its subclasses, `GZIPOutputStream` or `ZipOutputStream`, which wraps the raw compressed data within a standard file format.

```
public class DeflaterOutputStream extends FilterOutputStream {
    // Public Constructors
            public DeflaterOutputStream(OutputStream out, Deflater def, int size);
            public DeflaterOutputStream(OutputStream out, Deflater def);
            public DeflaterOutputStream(OutputStream out);
    // Protected Instance Variables
            protected byte[] buf;
            protected Deflater def;
    // Public Instance Methods
            public void close() throws IOException;  // Overrides FilterOutputStream
            public void finish() throws IOException;
            public void write(int b) throws IOException;  // Overrides
FilterOutputStream
            public void write(byte[] b, int off, int len) throws IOException;  //
Overrides FilterOutputStream
    // Protected Instance Methods
            protected void deflate() throws IOException;
}
```

Hierarchy:

Object->OutputStream->FilterOutputStream->DeflaterOutputStream

Extended By:

GZIPOutputStream, ZipOutputStream

31.12 java.util.zip.InflaterInputStream (JDK 1.1)

This class is a subclass of `java.io.FilterInputStream`; it reads a specified stream of compressed input data (typically one that was written with `DeflaterOutputStream` or a subclass) and "filters" that data by uncompressing ("inflating") it. To create an `InflaterInputStream`, you must specify the input stream it is to read from, and also an `Inflater` object that is to perform the uncompresssion. Once an `InflaterInputStream` is created, the `read()` and `skip()` methods are the same as those of other input streams.

Note that the `InflaterInputStream` uncompresses raw data. Applications often prefer one of its subclasses, `GZIPInputStream` or `ZipInputStream`, which work with compressed data written in the standard *gzip* and *PKZip* file formats.

```
public class InflaterInputStream extends FilterInputStream {
    // Public Constructors
            public InflaterInputStream(InputStream in, Inflater inf, int size);
            public InflaterInputStream(InputStream in, Inflater inf);
            public InflaterInputStream(InputStream in);
    // Protected Instance Variables
            protected byte[] buf;
            protected Inflater inf;
            protected int len;
    // Public Instance Methods
            public int read() throws IOException;  // Overrides FilterInputStream
            public int read(byte[] b, int off, int len) throws IOException;  // 
Overrides FilterInputStream
            public long skip(long n) throws IOException;  // Overrides 
FilterInputStream
    // Protected Instance Methods
            protected void fill() throws IOException;
}
```

Hierarchy:

Object->InputStream->FilterInputStream->InflaterInputStream

Extended By:

GZIPInputStream, ZipInputStream

22. The java.awt.peer Package

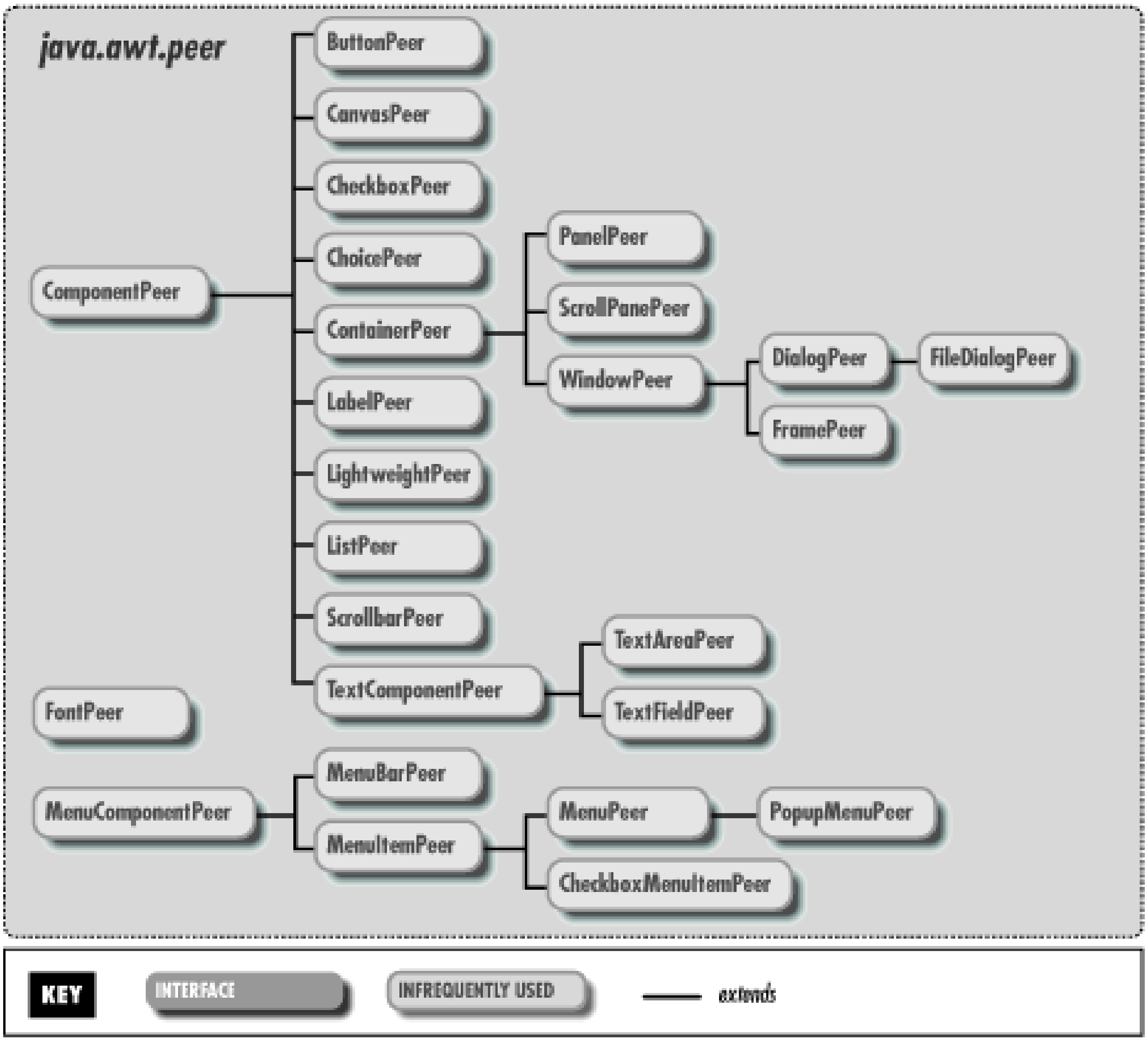
Contents:

The `java.awt.peer` package consists entirely of interface definitions. The hierarchy of these interfaces is shown in Figure 22.1. Each `java.awt.peer` interface corresponds to one of the `java.awt Component` or `MenuComponent` classes, and as you can see from the figure, the hierarchy of this package is identical to the hierarchy of those portions of the `java.awt` package.

The interfaces in this package define the methods that must be supported by the GUI components on a specific platform. Porting the `java.awt` GUI components to a new platform is a matter of implementing each of the methods in each of the interfaces in this package on top of the native GUI components of that platform. The `Toolkit` object in the `java.awt` package collects the implementations of these peer interfaces for a given platform. `Toolkit` contains methods that create instances of each of the interfaces in this package. Normal applications never need to instantiate these peers directly; instead they use the `java.awt Component` classes, which create peers as needed.

Because these peer interfaces are rarely used, and because the methods are quite similar to those of the corresponding `java.awt` component, there is no additional commentary for the individual interface definitions below.

Figure 22.1: The java.awt.peer package



22.1 java.awt.peer.ButtonPeer (JDK 1.0)

```
public abstract interface ButtonPeer extends ComponentPeer {
    // Public Instance Methods
            public abstract void setLabel(String label);
}
```

Returned By:

Toolkit.createButton()

24.30 java.io.InterruptedIOException (JDK 1.0)

An `IOException` that signals that an input or output operation was interrupted. The `bytesTransferred` variable contains the number of bytes read or written before the operation was interrupted.

```
public class InterruptedIOException extends IOException {
    // Public Constructors
            public InterruptedIOException();
            public InterruptedIOException(String s);
    // Public Instance Variables
            public int bytesTransferred;
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->IOException->InterruptedIOException

18.25 java.awt.FontMetrics (JDK 1.0)

This class represents font metrics for a specified `Font`. The methods allow you to determine the overall metrics for the font (ascent, descent, etc.), and also to compute the width of strings that are to be displayed in a particular font. You can obtain a `FontMetrics` object for a font with the `getFontMetrics()` method of `Component` or `Toolkit`.

```
public abstract class FontMetrics extends Object implements Serializable {
    // Protected Constructor
            protected FontMetrics(Font font);
    // Protected Instance Variables
            protected Font font;
    // Public Instance Methods
            public int bytesWidth(byte[] data, int off, int len);
            public int charWidth(int ch);
            public int charWidth(char ch);
            public int charsWidth(char[] data, int off, int len);
            public int getAscent();
            public int getDescent();
            public Font getFont();
            public int getHeight();
            public int getLeading();
            public int getMaxAdvance();
            public int getMaxAscent();
        #   public int getMaxDecent();
            public int getMaxDescent();
            public int[] getWidths();
            public int stringWidth(String str);
            public String toString();  // Overrides Object
}
```

Returned By:

`Component.getFontMetrics(), ComponentPeer.getFontMetrics(),` `Graphics.getFontMetrics(), Toolkit.getFontMetrics()`

25.6 java.lang.Byte (JDK 1.1)

This class provides an object wrapper around the `byte` primitive type. It defines useful constants for the minimum and maximum values that can be stored by the `byte` type, and also a `Class` object constant that represents the `byte` type. It also provides various methods for converting `Byte` values to and from strings and other numeric types.

Most of the static methods of this class are used to convert a `String` to a `Byte` object or a `byte` value: the four `parseByte()` and `valueOf()` methods parse a number from the specified string, using an optionally specified radix, and return it in one of these two forms. The `decode()` method parses a byte specified in base 10, base 8, or base 16 and returns it as a `Byte`. If the string begins with "0x" or "#", it is interpreted as a hexadecimal number. If it begins with "0", it is interpreted as an octal number. Otherwise, it is interpreted as a decimal number.

Note that this class has two different `toString()` methods. One is static and converts a `byte` primitive value to a String. The other is the usual `toString()` method that converts a `Byte` object to a string. Most of the remaining methods convert a `Byte` to various primitive numeric types.

```
public final class Byte extends Number {
    // Public Constructors
            public Byte(byte value);
            public Byte(String s) throws NumberFormatException;
    // Constants
            public static final byte MAX_VALUE;
            public static final byte MIN_VALUE;
            public static final Class TYPE;
    // Class Methods
            public static Byte decode(String nm) throws NumberFormatException;
            public static byte parseByte(String s) throws NumberFormatException;
            public static byte parseByte(String s, int radix) throws 
NumberFormatException;
            public static String toString(byte b);
            public static Byte valueOf(String s, int radix) throws 
NumberFormatException;
            public static Byte valueOf(String s) throws NumberFormatException;
    // Public Instance Methods
            public byte byteValue();  // Overrides Number
            public double doubleValue();  // Defines Number
            public boolean equals(Object obj);  // Overrides Object
            public float floatValue();  // Defines Number
            public int hashCode();  // Overrides Object
            public int intValue();  // Defines Number
            public long longValue();  // Defines Number
            public short shortValue();  // Overrides Number
            public String toString();  // Overrides Object
}
```

Hierarchy:

Object->Number(Serializable)->Byte

Returned By:

Byte.decode(), Byte.valueOf()

25.17 java.lang.Double (JDK 1.0)

This class provides an immutable object wrapper around the `double` primitive data type. `valueOf()` converts a string to a `Double`, `doubleValue()` returns the primitive `double` value of a `Double` object, and there are other methods for returning a `Double` value as a variety of other primitive types.

This class also provides some useful constants and static methods for testing `double` values. `MIN_VALUE` and `MAX_VALUE` are the smallest (closest to zero) and largest representable `double` values. `isInfinite()` in class method and instance method forms tests whether a `double` or a `Double` has an infinite value. Similarly, `isNaN()` tests whether a `double` or `Double` is not-a-number--this is a comparison that cannot be done directly because the `NaN` constant never tests equal to any other value, including itself. `doubleToLongBits()` and `longBitsToDouble()` allow you to manipulate the bit representation of a `double` directly.

```
public final class Double extends Number {
    // Public Constructors
            public Double(double value);
            public Double(String s) throws NumberFormatException;
    // Constants
            public static final double MAX_VALUE;
            public static final double MIN_VALUE;
            public static final double NEGATIVE_INFINITY;
            public static final double NaN;
            public static final double POSITIVE_INFINITY;
        1.1public static final Class TYPE;
    // Class Methods
            public static native long doubleToLongBits(double value);
            public static boolean isInfinite(double v);
            public static boolean isNaN(double v);
            public static native double longBitsToDouble(long bits);
            public static String toString(double d);
            public static Double valueOf(String s) throws NumberFormatException;
    // Public Instance Methods
        1.1public byte byteValue();  // Overrides Number
            public double doubleValue();  // Defines Number
            public boolean equals(Object obj);  // Overrides Object
            public float floatValue();  // Defines Number
            public int hashCode();  // Overrides Object
            public int intValue();  // Defines Number
            public boolean isInfinite();
            public boolean isNaN();
            public long longValue();  // Defines Number
        1.1public short shortValue();  // Overrides Number
            public String toString();  // Overrides Object
}
```

Hierarchy:

Object->Number(Serializable)->Double

Returned By:

Double.valueOf()

PREVIOUS HOME NEXT
java.lang.Compiler (JDK 1.0) BOOK INDEX java.lang.Error (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

25.21 java.lang.Float (JDK 1.0)

This class provides an immutable object wrapper around the `float` primitive data type. `valueOf()` converts a string to a `Float`, `floatValue()` returns the primitive `float` value of a `Float` object, and there are methods for returning a `Float` value as a variety of other primitive types.

This class also provides some useful constants and static methods for testing `float` values. `MIN_VALUE` and `MAX_VALUE` are the smallest (closest to zero) and largest representable `double` values. `isInfinite()` in class method and instance method forms tests whether a `float` or a `Float` has an infinite value. Similarly, `isNaN()` tests whether a `float` or `Float` is not-a-number--this is a comparison that cannot be done directly because the `NaN` constant never tests equal to any other value, including itself. `floatToIntBits()` and `intBitsToFloat()` allow you to manipulate the bit representation of a `float` directly.

```
public final class Float extends Number {
    // Public Constructors
            public Float(float value);
            public Float(double value);
            public Float(String s) throws NumberFormatException;
    // Constants
            public static final float MAX_VALUE;
            public static final float MIN_VALUE;
            public static final float NEGATIVE_INFINITY;
            public static final float NaN;
            public static final float POSITIVE_INFINITY;
        1.1public static final Class TYPE;
    // Class Methods
            public static native int floatToIntBits(float value);
            public static native float intBitsToFloat(int bits);
            public static boolean isInfinite(float v);
            public static boolean isNaN(float v);
            public static String toString(float f);
            public static Float valueOf(String s) throws NumberFormatException;
    // Public Instance Methods
        1.1public byte byteValue();  // Overrides Number
            public double doubleValue();  // Defines Number
            public boolean equals(Object obj);  // Overrides Object
            public float floatValue();  // Defines Number
            public int hashCode();  // Overrides Object
            public int intValue();  // Defines Number
            public boolean isInfinite();
            public boolean isNaN();
            public long longValue();  // Defines Number
        1.1public short shortValue();  // Overrides Number
            public String toString();  // Overrides Object
}
```

Hierarchy:

Object->Number(Serializable)->Float

Returned By:

Float.valueOf()

PREVIOUS | HOME | NEXT

java.lang.ExceptionInInitializerError (JDK 1.1) | BOOK INDEX | java.lang.IllegalAccessError (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

25.32 java.lang.Integer (JDK 1.0)

This class provides an immutable object wrapper around the `int` primitive data type. This class also contains useful minimum and maximum constants and useful conversion methods. `parseInt()` and `valueOf()` convert a string to an `int` or to an `Integer`, respectively. Each can take a radix argument to specify the base that the value is represented in. `decode()` also converts a `String` to an `Integer`. It assumes a hexadecimal number if the string begins with "0X" or "0x," or an octal number if the string begins with "0". Otherwise, a decimal number is assumed.

`toString()` converts in the other direction, and the `static` version takes a radix argument. `toBinaryString()`, `toOctalString()`, and `toHexString()` convert an `int` to a string using base 2, base 8, and base 16. These methods treat the integer as an unsigned value.

Other routines return the value of an `Integer` as various primitive types, and finally, the `getInteger()` methods return the integer value of a named property from the system property list or the specified default value.

```
public final class Integer extends Number {
    // Public Constructors
            public Integer(int value);
            public Integer(String s) throws NumberFormatException;
    // Constants
            public static final int MAX_VALUE;
            public static final int MIN_VALUE;
        1.1public static final Class TYPE;
    // Class Methods
        1.1public static Integer decode(String nm) throws NumberFormatException;
            public static Integer getInteger(String nm);
            public static Integer getInteger(String nm, int val);
            public static Integer getInteger(String nm, Integer val);
            public static int parseInt(String s, int radix) throws 
NumberFormatException;
            public static int parseInt(String s) throws NumberFormatException;
            public static String toBinaryString(int i);
            public static String toHexString(int i);
            public static String toOctalString(int i);
            public static String toString(int i, int radix);
            public static String toString(int i);
            public static Integer valueOf(String s, int radix) throws 
NumberFormatException;
```

```
            public static Integer valueOf(String s) throws NumberFormatException;
    // Public Instance Methods
        1.1public byte byteValue();  // Overrides Number
            public double doubleValue();  // Defines Number
            public boolean equals(Object obj);  // Overrides Object
            public float floatValue();  // Defines Number
            public int hashCode();  // Overrides Object
            public int intValue();  // Defines Number
            public long longValue();  // Defines Number
        1.1public short shortValue();  // Overrides Number
            public String toString();  // Overrides Object
}
```

Hierarchy:

Object->Number(Serializable)->Integer

Passed To:

Integer.getInteger()

Returned By:

Integer.decode(), Integer.getInteger(), Integer.valueOf()

PREVIOUS | HOME | NEXT

java.lang.InstantiationException (JDK 1.0) | BOOK INDEX | java.lang.InternalError (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

25.36 java.lang.Long (JDK 1.0)

This class provides an immutable object wrapper around the `long` primitive data type. This class also contains useful minimum and maximum constants and useful conversion methods. `parseLong()` and `valueOf()` convert a string to a `long` or to a `Long`, respectively. Each can take a radix argument to specify the base that the value is represented in. `toString()` converts in the other direction and may also take a radix argument. `toBinaryString()`, `toOctalString()`, and `toHexString()` convert a `long` to a string using base 2, base 8, and base 16. These methods treat the `long` as an unsigned value.

Other routines return the value of a `Long` as various primitive types, and finally, the `getLong()` methods return the `long` value of a named property or the value of the specified default.

```
public final class Long extends Number {
    // Public Constructors
            public Long(long value);
            public Long(String s) throws NumberFormatException;
    // Constants
            public static final long MAX_VALUE;
            public static final long MIN_VALUE;
        1.1public static final Class TYPE;
    // Class Methods
            public static Long getLong(String nm);
            public static Long getLong(String nm, long val);
            public static Long getLong(String nm, Long val);
            public static long parseLong(String s, int radix) throws
NumberFormatException;
            public static long parseLong(String s) throws NumberFormatException;
            public static String toBinaryString(long i);
            public static String toHexString(long i);
            public static String toOctalString(long i);
            public static String toString(long i, int radix);
            public static String toString(long i);
            public static Long valueOf(String s, int radix) throws
NumberFormatException;
            public static Long valueOf(String s) throws NumberFormatException;
    // Public Instance Methods
        1.1public byte byteValue();  // Overrides Number
            public double doubleValue();  // Defines Number
            public boolean equals(Object obj);  // Overrides Object
            public float floatValue();  // Defines Number
            public int hashCode();  // Overrides Object
            public int intValue();  // Defines Number
            public long longValue();  // Defines Number
        1.1public short shortValue();  // Overrides Number
            public String toString();  // Overrides Object
}
```

Hierarchy:

Object->Number(Serializable)->Long

Passed To:

Long.getLong()

Returned By:

Long.getLong(), Long.valueOf()

25.45 java.lang.Number (JDK 1.0)

This is an abstract class that is the superclass of `Byte`, `Short`, `Integer`, `Long`, `Float`, and `Double`. It defines the conversion functions that those types all implement.

```
public abstract class Number extends Object implements Serializable {
    // Default Constructor: public Number()
    // Public Instance Methods
        1.1public byte byteValue();
            public abstract double doubleValue();
            public abstract float floatValue();
            public abstract int intValue();
            public abstract long longValue();
        1.1public short shortValue();
}
```

Extended By:

BigDecimal, BigInteger, Byte, Double, Float, Integer, Long, Short

Returned By:

ChoiceFormat.parse(), DecimalFormat.parse(), NumberFormat.parse()

25.55 java.lang.Short (JDK 1.1)

This class provides an object wrapper around the `short` primitive type. It defines useful constants for the minimum and maximum values that can be stored by the `short` type, and also a `Class` object constant that represents the `short` type. It also provides various methods for converting `Short` values to and from strings and other numeric types.

Most of the static methods of this class are used to convert a `String` to a `Short` object or a `short` value: the four `parseShort()` and `valueOf()` methods parse a number from the specified string, using an optionally specified radix, and return it in one of these two forms. The `decode()` method parses a number specified in base 10, base 8, or base 16 and returns it as a `Short`. If the string begins with "0x" or "#", it is interpreted as a hexadecimal number, or if it begins with "0", it is interpreted as an octal number. Otherwise, it is interpreted as a decimal number.

Note that this class has two different `toString()` methods. One is static and converts a `short` primitive value to a String. The other is the usual `toString()` method that converts a `Short` object to a string. Most of the remaining methods convert a `Short` to various primitive numeric types.

```
public final class Short extends Number {
    // Public Constructors
            public Short(short value);
            public Short(String s) throws NumberFormatException;
    // Constants
            public static final short MAX_VALUE;
            public static final short MIN_VALUE;
            public static final Class TYPE;
    // Class Methods
            public static Short decode(String nm) throws NumberFormatException;
            public static short parseShort(String s) throws NumberFormatException;
            public static short parseShort(String s, int radix) throws 
NumberFormatException;
            public static String toString(short s);
            public static Short valueOf(String s, int radix) throws 
NumberFormatException;
            public static Short valueOf(String s) throws NumberFormatException;
    // Public Instance Methods
            public byte byteValue();  // Overrides Number
            public double doubleValue();  // Defines Number
            public boolean equals(Object obj);  // Overrides Object
            public float floatValue();  // Defines Number
            public int hashCode();  // Overrides Object
            public int intValue();  // Defines Number
            public long longValue();  // Defines Number
            public short shortValue();  // Overrides Number
            public String toString();  // Overrides Object
}
```

Hierarchy:

Object->Number(Serializable)->Short

Returned By:

Short.decode(), Short.valueOf()

PREVIOUS | HOME | NEXT
java.lang.SecurityManager (JDK 1.0) | BOOK INDEX | java.lang.StackOverflowError (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

30.12 java.util.Locale (JDK 1.1)

The `Locale` class represents a locale: a political, geographical, or cultural region that typically has a distinct language and distinct customs and conventions for such things as formatting dates, times, and numbers.

The `Locale` class defines a number of constants that represent commonly used locales. `Locale` also defines a static `getDefault()` method that returns the default `Locale` object, which represents a locale value inherited from the host system. Furthermore, many locale-sensitive classes, such as `DateFormat`, provide `getAvailableLocales()` methods that return a list of supported `Locale` objects. If none of these methods for obtaining a `Locale` object is suitable, you can also explicitly create your own `Locale` object. To do this, you must specify a language code, a country code, and an optional variant string.

The `Locale` class does not implement any internationalization behavior itself; it merely serves as a locale identifier for those classes that can localize their behavior. Given a `Locale` object, you can invoke the various `getDisplay` methods to obtain a description of the locale suitable for display to a user. Note that these methods may themselves take a `Locale` argument so that the names of languages and countries can be localized as appropriate.

```
public final class Locale extends Object implements Cloneable, Serializable {
    // Public Constructors
            public Locale(String language, String country, String variant);
            public Locale(String language, String country);
    // Constants
            public static final Locale CANADA;
            public static final Locale CANADA_FRENCH;
            public static final Locale CHINA;
            public static final Locale CHINESE;
            public static final Locale ENGLISH;
            public static final Locale FRANCE;
            public static final Locale FRENCH;
            public static final Locale GERMAN;
            public static final Locale GERMANY;
            public static final Locale ITALIAN;
            public static final Locale ITALY;
            public static final Locale JAPAN;
            public static final Locale JAPANESE;
            public static final Locale KOREA;
            public static final Locale KOREAN;
            public static final Locale PRC;
            public static final Locale SIMPLIFIED_CHINESE;
            public static final Locale TAIWAN;
            public static final Locale TRADITIONAL_CHINESE;
            public static final Locale UK;
            public static final Locale US;
    // Class Methods
            public static synchronized Locale getDefault();
            public static synchronized void setDefault(Locale newLocale);
    // Public Instance Methods
            public Object clone();  // Overrides Object
            public boolean equals(Object obj);  // Overrides Object
            public String getCountry();
            public final String getDisplayCountry();
            public String getDisplayCountry(Locale inLocale);
            public final String getDisplayLanguage();
            public String getDisplayLanguage(Locale inLocale);
            public final String getDisplayName();
            public String getDisplayName(Locale inLocale);
            public final String getDisplayVariant();
            public String getDisplayVariant(Locale inLocale);
            public String getISO3Country() throws MissingResourceException;
            public String getISO3Language() throws MissingResourceException;
            public String getLanguage();
            public String getVariant();
            public synchronized int hashCode();  // Overrides Object
            public final String toString();  // Overrides Object
}
```

Passed To:

BreakIterator.getCharacterInstance(), BreakIterator.getLineInstance(), BreakIterator.getSentenceInstance(), BreakIterator.getWordInstance(), Calendar(), Calendar.getInstance(), Collator.getInstance(), Component.setLocale(), DateFormat.getDateInstance(), DateFormat.getDateTimeInstance(), DateFormat.getTimeInstance(), DateFormatSymbols(), DecimalFormatSymbols(), GregorianCalendar(), Locale.getDisplayCountry(), Locale.getDisplayLanguage(), Locale.getDisplayName(), Locale.getDisplayVariant(), Locale.setDefault(), MessageFormat.setLocale(), NumberFormat.getCurrencyInstance(), NumberFormat.getInstance(), NumberFormat.getNumberInstance(), NumberFormat.getPercentInstance(), ResourceBundle.getBundle(), SimpleDateFormat(), String.toLowerCase(), String.toUpperCase()

Returned By:

Applet.getLocale(), BreakIterator.getAvailableLocales(), Calendar.getAvailableLocales(), Collator.getAvailableLocales(), Component.getLocale(), DateFormat.getAvailableLocales(), Locale.getDefault(), MessageFormat.getLocale(), NumberFormat.getAvailableLocales(), Window.getLocale()

PREVIOUS | HOME | NEXT
java.util.ListResourceBundle (JDK 1.1) | BOOK INDEX | java.util.MissingResourceException (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

21.13 java.awt.image.RGBImageFilter (JDK 1.0)

This abstract class is an `ImageFilter` that provides an easy way to implement filters that modify the colors of an image. To create a color filter that modifies the colors of an image, you should subclass `RGBImageFilter` and provide a definition of `filterRGB()` that converts the input pixel value (in the default RGB color model) to an output value. If the conversion does not depend on the location of the pixel, set the `canFilterIndexColorModel` variable to `true` so that the `RGBImageFilter` can save time by filtering the colormap of an image that uses an `IndexColorModel` instead of filtering each pixel of the image.

```
public abstract class RGBImageFilter extends ImageFilter {
    // Default Constructor: public RGBImageFilter()
    // Protected Instance Variables
            protected boolean canFilterIndexColorModel;
            protected ColorModel newmodel;
            protected ColorModel origmodel;
    // Public Instance Methods
            public IndexColorModel filterIndexColorModel(IndexColorModel icm);
            public abstract int filterRGB(int x, int y, int rgb);
            public void filterRGBPixels(int x, int y, int w, int h, int[] pixels, int 
off, int scansize);
            public void setColorModel(ColorModel model);  // Overrides ImageFilter
            public void setPixels(int x, int y, int w, int h, ColorModel model, 
            public void setPixels'u'byte[] pixels, int off, int scansize);  // 
Overrides ImageFilter
            public void setPixels(int x, int y, int w, int h, ColorModel model, 
            public void setPixels'u'int[] pixels, int off, int scansize);  // 
Overrides ImageFilter
            public void substituteColorModel(ColorModel oldcm, ColorModel newcm);
}
```

Hierarchy:

```
Object->ImageFilter(ImageConsumer, Cloneable)->RGBImageFilter
```

29.6 java.text.Collator (JDK 1.1)

This class is used to compare, order, and sort strings in a way that is appropriate for the default locale or some other specified locale. Because it is an abstract class, it cannot be instantiated directly. Instead, you must use the static `getInstance()` method to obtain an instance of a `Collator` subclass that is appropriate for the default or specified locale. You can use `getAvailableLocales()` to determine whether a `Collator` object is available for a desired locale.

Once an appropriate `Collator` object has been obtained, you can use the `compare()` method to compare strings. The possible return values of this method are -1, 0, and 1, which indicate, respectively, that the first string is collated before the second, that the two are equivalent for collation purposes, and that the first string is collated after the second. The `equals()` method is a convenient shortcut for testing two strings for collation equivalence.

When sorting an array of strings, each string in the array is typically compared more than once. Using the `compare()` method in this case is inefficient. A more efficient method for comparing strings multiple times is to use `getCollationKey()` for each string to create `CollationKey` objects. These objects can then be compared to each other more quickly than the strings themselves could be compared.

You can customize the way the `Collator` object performs comparisons by calling `setStrength()`. If you pass the constant `PRIMARY` to this method, the comparison only looks at primary differences in the strings--it compares letters but ignores accents and case differences. If you pass the constant `SECONDARY`, it ignores case differences but does not ignore accents. And if you pass `TERTIARY` (the default), the `Collator` object takes both accents and case differences into account in its comparison.

```
public abstract class Collator extends Object implements Cloneable, Serializable {
    // Protected Constructor
            protected Collator();
    // Constants
            public static final int CANONICAL_DECOMPOSITION;
            public static final int FULL_DECOMPOSITION;
            public static final int IDENTICAL;
            public static final int NO_DECOMPOSITION;
            public static final int PRIMARY;
            public static final int SECONDARY;
            public static final int TERTIARY;
    // Class Methods
            public static synchronized Locale[] getAvailableLocales();
            public static synchronized Collator getInstance();
            public static synchronized Collator getInstance(Locale desiredLocale);
    // Public Instance Methods
            public Object clone();  // Overrides Object
            public abstract int compare(String source, String target);
            public boolean equals(String source, String target);
            public boolean equals(Object that);  // Overrides Object
            public abstract CollationKey getCollationKey(String source);
            public synchronized int getDecomposition();
            public synchronized int getStrength();
            public abstract synchronized int hashCode();  // Overrides Object
            public synchronized void setDecomposition(int decompositionMode);
            public synchronized void setStrength(int newStrength);
}
```

Extended By:

RuleBasedCollator

Returned By:

Collator.getInstance()

24.16 java.io.File (JDK 1.0)

This class supports a platform-independent definition of file and directory names. It also provides methods to list the files in a directory, to check the existence, readability, writeability, type, size, and modification time of files and directories, to make new directories, to rename files and directories, and to delete files and directories. The constants defined by this class are the platform-dependent directory and path separator characters, available as a `String` or `char`.

`getName()` returns the name of the `File` with any directory names omitted. `getPath()` returns the full name of the file, including the directory name. `getParent()` returns the directory of the `File`. If the `File` is an absolute specification, then `getAbsolutePath()` returns the complete filename. Otherwise, if the `File` is a relative file specification, it returns the relative filename appended to the current working directory.

`isAbsolute()` tests whether the `File` is an absolute specification. `exists()`, `canWrite()`, `canRead()`, `isFile()`, and `isDirectory()` perform the obvious tests on the specified `File`. `length()` returns the length of the file. `lastModified()` returns the modification time of the file (which should be used for comparison with other file times only, and not interpreted as any particular time format).

`list()` returns the name of all entries in a directory that are not rejected by an optional `FilenameFilter`. `mkdir()` creates a directory. `mkdirs()` creates all the directories in a `File` specification. `renameTo()` renames a file or directory. `delete()` deletes a file or directory. Note that there is no method to create a file; that is done with a `FileOutputStream`.

```
public class File extends Object implements Serializable {
    // Public Constructors
            public File(String path);
            public File(String path, String name);
            public File(File dir, String name);
    // Constants
```

```
            public static final String pathSeparator;
            public static final char pathSeparatorChar;
            public static final String separator;
            public static final char separatorChar;
    // Public Instance Methods
            public boolean canRead();
            public boolean canWrite();
            public boolean delete();
            public boolean equals(Object obj);  // Overrides Object
            public boolean exists();
            public String getAbsolutePath();
        1.1  public String getCanonicalPath() throws IOException;
            public String getName();
            public String getParent();
            public String getPath();
            public int hashCode();  // Overrides Object
            public native boolean isAbsolute();
            public boolean isDirectory();
            public boolean isFile();
            public long lastModified();
            public long length();
            public String[] list();
            public String[] list(FilenameFilter filter);
            public boolean mkdir();
            public boolean mkdirs();
            public boolean renameTo(File dest);
            public String toString();  // Overrides Object
}
```

Passed To:

File(), File.renameTo(), FileInputStream(), FilenameFilter.accept(), FileOutputStream(), FileReader(), FileWriter(), RandomAccessFile(), ZipFile()

PREVIOUS | HOME | NEXT
java.io.Externalizable (JDK 1.1) | BOOK INDEX | java.io.FileDescriptor (JDK 1.0)

20.18 java.awt.event.KeyEvent (JDK 1.1)

An event of this type indicates that the user has pressed or released a key or typed a character.

Call `getID()` to determine the particular type of key event that has occurred. The constant `KEY_PRESSED` indicates that a key has been pressed, while the constant `KEY_RELEASED` indicates that a key has been released. Not all keystrokes actually correspond to or generate Unicode characters. Modifier keys and function keys, for example, do not correspond to characters. Furthermore, for internationalized input, multiple keystrokes are sometimes required to generate a single character of input. Therefore, `getID()` returns a third constant, `KEY_TYPED`, to indicate a `KeyEvent` that actually contains a character value.

For `KEY_PRESSED` and `KEY_RELEASED` key events, use `getKeyCode()` to obtain the "virtual key code" of the key that was pressed or released. `KeyEvent` defines a number of `VK_` constants that represent these "virtual keys." Note that not all keys on all keyboards have corresponding constants in the `KeyEvent` class, and not all keyboards can generate all of the virtual key codes defined by the class. In JDK 1.1, the `VK_` constants for letter keys, number keys, and some other keys have the same values as the ASCII encodings of the letters and numbers. You should not rely on this to always be the case, however. If the key that was pressed or released corresponds directly to a Unicode character, you can obtain that character by calling `getKeyChar()`. If there is not a corresponding Unicode character, this method returns the constant `CHAR_UNDEFINED`. The `isActionKey()` method returns `true` if the key that was pressed or released does not have a corresponding character.

For `KEY_TYPED` key events, use `getKeyChar()` to return the Unicode character that was typed. If you call `getKeyCode()` for this type of key event, it returns `VK_UNDEFINED`.

See `InputEvent` for information on inherited methods you can use to obtain the keyboard modifiers that were down during the event and other important methods.

Use `getComponent()`, inherited from `ComponentEvent`, to determine what component the event occurred over.

The static method `getKeyText()` returns a (possibly localized) textual name for a given key code. The static method `getKeyModifiersText()` returns a (possibly localized) textual description for a set of modifiers.

The `KeyEvent` has methods that allow you to change the key code, key character, or modifiers of an event. These methods, along with the `consume()` method, allow a `KeyListener` to perform filtering of key events before they are passed to the underlying AWT component.

```
public class KeyEvent extends InputEvent {
    // Public Constructors
            public KeyEvent(Component source, int id, long when, int modifiers, int
keyCode, char keyChar);
            public KeyEvent(Component source, int id, long when, int modifiers, int
keyCode);
    // Constants
    // Event Type Constants
            public static final int KEY_FIRST;
            public static final int KEY_LAST;
            public static final int KEY_PRESSED;
            public static final int KEY_RELEASED;
            public static final int KEY_TYPED;
    // Undefined Key and Character
            public static final int VK_UNDEFINED;
            public static final char CHAR_UNDEFINED;
    // Alphanumeric Keys
            public static final int VK_A, VK_B, VK_C, VK_D, VK_E, VK_F, VK_G, VK_H,
VK_I;
            public static final int VK_J, VK_K, VK_L, VK_M, VK_N, VK_O, VK_P, VK_Q,
VK_R;
            public static final int VK_S, VK_T, VK_U, VK_V, VK_W, VK_X, VK_Y, VK_Z;
            public static final int VK_SPACE;
            public static final int VK_0, VK_1, VK_2, VK_3, VK_4, VK_5, VK_6, VK_7,
VK_8, VK_9;
            public static final int VK_NUMPAD0, VK_NUMPAD1, VK_NUMPAD2, VK_NUMPAD3,
VK_NUMPAD4;
            public static final int VK_NUMPAD5, VK_NUMPAD6, VK_NUMPAD7, VK_NUMPAD8,
VK_NUMPAD9;
    // Control Keys
            public static final int VK_BACK_SPACE, VK_ENTER, VK_ESCAPE, VK_TAB;
    // Modifier Keys
            public static final int VK_ALT, VK_CAPS_LOCK, VK_CONTROL, VK_META,
VK_SHIFT;
    // Function Keys
            public static final int VK_F1, VK_F2, VK_F3, VK_F4, VK_F5, VK_F6;
            public static final int VK_F7, VK_F8, VK_F9, VK_F10, VK_F11, VK_F12;
            public static final int VK_PRINTSCREEN, VK_SCROLL_LOCK, VK_PAUSE;
            public static final int VK_DELETE, VK_INSERT;
            public static final int VK_PAGE_DOWN, VK_PAGE_UP;
            public static final int VK_DOWN, VK_LEFT, VK_RIGHT, VK_UP;
            public static final int VK_END, VK_HOME;
            public static final int VK_ACCEPT, VK_NUM_LOCK, VK_CANCEL;
            public static final int VK_CLEAR, VK_CONVERT, VK_FINAL;
            public static final int VK_HELP, VK_KANA, VK_KANJI;
            public static final int VK_MODECHANGE, VK_NONCONVERT;
    // Punctuation Keys
            public static final int VK_ADD, VK_BACK_QUOTE, VK_BACK_SLASH;
            public static final int VK_CLOSE_BRACKET, VK_COMMA, VK_DECIMAL;
            public static final int VK_DIVIDE, VK_EQUALS, VK_MULTIPLY;
            public static final int VK_OPEN_BRACKET, VK_PERIOD, VK_QUOTE;
            public static final int VK_SEMICOLON, VK_SEPARATER, VK_SLASH;
            public static final int VK_SUBTRACT;
    // Class Methods
```

```
            public static String getKeyModifiersText(int modifiers);
            public static String getKeyText(int keyCode);
    // Public Instance Methods
            public char getKeyChar();
            public int getKeyCode();
            public boolean isActionKey();
            public String paramString();  // Overrides ComponentEvent
            public void setKeyChar(char keyChar);
            public void setKeyCode(int keyCode);
            public void setModifiers(int modifiers);
}
```

Hierarchy:

```
Object->EventObject(Serializable)->AWTEvent->ComponentEvent->InputEvent->KeyEvent
```

Passed To:

```
AWTEventMulticaster.keyPressed(), AWTEventMulticaster.keyReleased(),
AWTEventMulticaster.keyTyped(), Component.processKeyEvent(),
KeyAdapter.keyPressed(), KeyAdapter.keyReleased(), KeyAdapter.keyTyped(),
KeyListener.keyPressed(), KeyListener.keyReleased(), KeyListener.keyTyped()
```

25.57 java.lang.String (JDK 1.0)

The String class represents a string of characters. A String object is created by the Java compiler whenever it encounters a string in doublequotes--this method of creation is usually simpler than using a constructor. A number of the methods of this class provide useful string manipulation functions. length() returns the number of characters in a string. charAt() extracts a character from a string. compareTo() compares two strings. equalsIgnoreCase() tests string for equality, ignoring case. startsWith() and endsWith() compare the start and end of a string to a specified value. indexOf() and lastIndexOf() search forward and backwards in a string for a specified character or substring. substring() returns a substring of a string. replace() creates a new copy of the string with one character replaced by another. toUpperCase() and toLowerCase() convert the case of a string. trim() strips whitespace from the start and end of a string. concat() concatenates two strings, which can also be done with the + operator.

Note that String objects are immutable--there is no setCharAt() method to change the contents. The methods above that return a String do not modify the string they are passed, but instead return a modified copy of the string. Use a StringBuffer if you want to manipulate the contents of a string, or use toCharArray() to convert a string to an array of char values.

The static valueOf() methods convert various Java primitive types to strings.

```
public final class String extends Object implements Serializable {
    // Public Constructors
            public String();
            public String(String value);
            public String(char[] value);
            public String(char[] value, int offset, int count);
        #   public String(byte[] ascii, int hibyte, int offset, int count);
        #   public String(byte[] ascii, int hibyte);
        1.1public String(byte[] bytes, int offset, int length, String enc) throws 
UnsupportedEncodingException;
        1.1public String(byte[] bytes, String enc) throws 
UnsupportedEncodingException;
        1.1public String(byte[] bytes, int offset, int length);
        1.1public String(byte[] bytes);
            public String(StringBuffer buffer);
    // Class Methods
            public static String copyValueOf(char[] data, int offset, int count);
            public static String copyValueOf(char[] data);
            public static String valueOf(Object obj);
            public static String valueOf(char[] data);
            public static String valueOf(char[] data, int offset, int count);
            public static String valueOf(boolean b);
            public static String valueOf(char c);
            public static String valueOf(int i);
            public static String valueOf(long l);
            public static String valueOf(float f);
            public static String valueOf(double d);
    // Public Instance Methods
            public char charAt(int index);
            public int compareTo(String anotherString);
            public String concat(String str);
            public boolean endsWith(String suffix);
            public boolean equals(Object anObject);  // Overrides Object
            public boolean equalsIgnoreCase(String anotherString);
        #   public void getBytes(int srcBegin, int srcEnd, byte[] dst, int dstBegin);
        1.1public byte[] getBytes(String enc) throws UnsupportedEncodingException;
        1.1public byte[] getBytes();
            public void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin);
            public int hashCode();  // Overrides Object
            public int indexOf(int ch);
            public int indexOf(int ch, int fromIndex);
            public int indexOf(String str);
            public int indexOf(String str, int fromIndex);
            public native String intern();
            public int lastIndexOf(int ch);
            public int lastIndexOf(int ch, int fromIndex);
            public int lastIndexOf(String str);
            public int lastIndexOf(String str, int fromIndex);
            public int length();
            public boolean regionMatches(int toffset, String other, int ooffset, int
len);
            public boolean regionMatches(boolean ignoreCase, int toffset, String
other, int ooffset, int len);
            public String replace(char oldChar, char newChar);
            public boolean startsWith(String prefix, int toffset);
            public boolean startsWith(String prefix);
            public String substring(int beginIndex);
            public String substring(int beginIndex, int endIndex);
            public char[] toCharArray();
        1.1public String toLowerCase(Locale locale);
            public String toLowerCase();
            public String toString();  // Overrides Object
        1.1public String toUpperCase(Locale locale);
            public String toUpperCase();
            public String trim();
}
```

Passed To:

Many methods

Returned By:

```
Many methods
```

Type Of:

BorderLayout.CENTER, BorderLayout.EAST, BorderLayout.NORTH, BorderLayout.SOUTH, BorderLayout.WEST, File.pathSeparator, File.separator, Font.name, HttpURLConnection.method, HttpURLConnection.responseMessage, InvalidClassException.classname, StreamTokenizer.sval, StringBufferInputStream.buffer

PREVIOUS java.lang.StackOverflowError (JDK 1.0)

HOME
BOOK INDEX

NEXT java.lang.StringBuffer (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

25.7 java.lang.Character (JDK 1.0)

This class provides an immutable object wrapper around the primitive `char` data type. `charValue()` returns the `char` value of a `Character` object. A number of class methods provide the Java/Unicode equivalent of the C <*ctype.h*> character macros for checking the type of characters and converting to uppercase and lowercase letters. `getType()` returns the character type. The return value is one of the constants defined by the class, which represents a number of broad Unicode character categories.

`digit()` returns the integer equivalent of a given character for a given radix (e.g., radix 16 for hexadecimal). `forDigit()` returns the character that corresponds to the specified value for the specified radix.

```
public final class Character extends Object implements Serializable {
    // Public Constructor
            public Character(char value);
    // Constants
            public static final int MAX_RADIX, MIN_RADIX;
            public static final char MAX_VALUE, MIN_VALUE;
        1.1public static final Class TYPE;
    // Character Type Constants
        1.1public static final byte COMBINING_SPACING_MARK, CONNECTOR_PUNCTUATION,
CONTROL;
        1.1public static final byte CURRENCY_SYMBOL, DASH_PUNCTUATION,
DECIMAL_DIGIT_NUMBER;
        1.1public static final byte ENCLOSING_MARK, END_PUNCTUATION, FORMAT;
        1.1public static final byte LETTER_NUMBER, LINE_SEPARATOR, LOWERCASE_LETTER;
        1.1public static final byte MATH_SYMBOL, MODIFIER_LETTER, MODIFIER_SYMBOL;
        1.1public static final byte NON_SPACING_MARK, OTHER_LETTER, OTHER_NUMBER;
        1.1public static final byte OTHER_PUNCTUATION, OTHER_SYMBOL,
PARAGRAPH_SEPARATOR;
        1.1public static final byte PRIVATE_USE, SPACE_SEPARATOR, START_PUNCTUATION;
        1.1public static final byte SURROGATE, TITLECASE_LETTER, UNASSIGNED;
        1.1public static final byte UPPERCASE_LETTER;
    // Class Methods
            public static int digit(char ch, int radix);
            public static char forDigit(int digit, int radix);
        1.1public static int getNumericValue(char ch);
        1.1public static int getType(char ch);
            public static boolean isDefined(char ch);
            public static boolean isDigit(char ch);
        1.1public static boolean isISOControl(char ch);
        1.1public static boolean isIdentifierIgnorable(char ch);
        1.1public static boolean isJavaIdentifierPart(char ch);
        1.1public static boolean isJavaIdentifierStart(char ch);
        #   public static boolean isJavaLetter(char ch);
        #   public static boolean isJavaLetterOrDigit(char ch);
            public static boolean isLetter(char ch);
            public static boolean isLetterOrDigit(char ch);
            public static boolean isLowerCase(char ch);
        #   public static boolean isSpace(char ch);
        1.1public static boolean isSpaceChar(char ch);
            public static boolean isTitleCase(char ch);
        1.1public static boolean isUnicodeIdentifierPart(char ch);
        1.1public static boolean isUnicodeIdentifierStart(char ch);
            public static boolean isUpperCase(char ch);
        1.1public static boolean isWhitespace(char ch);
            public static char toLowerCase(char ch);
            public static char toTitleCase(char ch);
            public static char toUpperCase(char ch);
    // Public Instance Methods
            public char charValue();
            public boolean equals(Object obj);  // Overrides Object
            public int hashCode();  // Overrides Object
            public String toString();  // Overrides Object
}
```

25.54 java.lang.SecurityManager (JDK 1.0)

This abstract class defines the methods necessary to implement a security policy for the execution of untrusted code. Before performing potentially sensitive operations, Java calls methods of the `SecurityManager` object currently in effect to determine whether the operations are permitted. These methods throw a `SecurityException` if the operation is not permitted.

Normal applications do not need to use or subclass the `SecurityManager` class. It is typically only used by Web browsers, applet viewers, and other programs that need to run untrusted code in a controlled environment.

```
public abstract class SecurityManager extends Object {
    // Protected Constructor
            protected SecurityManager();
    // Protected Instance Variables
            protected boolean inCheck;
    // Public Instance Methods
            public void checkAccept(String host, int port);
            public void checkAccess(Thread g);
            public void checkAccess(ThreadGroup g);
        1.1public void checkAwtEventQueueAccess();
            public void checkConnect(String host, int port);
            public void checkConnect(String host, int port, Object context);
            public void checkCreateClassLoader();
            public void checkDelete(String file);
            public void checkExec(String cmd);
            public void checkExit(int status);
            public void checkLink(String lib);
            public void checkListen(int port);
        1.1public void checkMemberAccess(Class clazz, int which);
        1.1public void checkMulticast(InetAddress maddr);
        1.1public void checkMulticast(InetAddress maddr, byte ttl);
            public void checkPackageAccess(String pkg);
            public void checkPackageDefinition(String pkg);
        1.1public void checkPrintJobAccess();
            public void checkPropertiesAccess();
            public void checkPropertyAccess(String key);
            public void checkRead(FileDescriptor fd);
            public void checkRead(String file);
            public void checkRead(String file, Object context);
        1.1public void checkSecurityAccess(String action);
            public void checkSetFactory();
        1.1public void checkSystemClipboardAccess();
            public boolean checkTopLevelWindow(Object window);
            public void checkWrite(FileDescriptor fd);
            public void checkWrite(String file);
            public boolean getInCheck();
            public Object getSecurityContext();
        1.1public ThreadGroup getThreadGroup();
    // Protected Instance Methods
            protected native int classDepth(String name);
            protected native int classLoaderDepth();
            protected native ClassLoader currentClassLoader();
        1.1protected Class currentLoadedClass();
            protected native Class[] getClassContext();
            protected boolean inClass(String name);
            protected boolean inClassLoader();
}
```

Passed To:

System.setSecurityManager()

Returned By:

System.getSecurityManager()

24.52 java.io.PrintStream (JDK 1.0)

This class is a FilterOutputStream that implements a number of methods for displaying textual representation of Java primitive data types. The print() methods output a standard textual representation of each data type. The println() methods do the same, and follow that representation with a newline. The methods convert various Java primitive types to String representations and then output the resulting string. When an Object is passed to a print() or println(), it is converted to a String by calling its toString() method.

PrintStream is the OutputStream type that makes it easiest to output text. As such, it is the most commonly used of the output streams. The System.out variable is a PrintStream.

Note that in Java 1.0 this class does not handle Unicode characters correctly--it discards the top 8 bits of all 16-bit characters, and thus works only with Latin-1 (ISO8859-1) characters. Although this problem has been fixed in Java 1.1, PrintStream has been superseded in Java 1.1 with PrintWriter. The constructors of this class have been deprecated, but the class itself has not, because it is still used by the System.out and System.err standard output streams.

PrintStream and its PrintWriter replacement output textual representations of Java data types. Use DataOutputStream to output binary representations of data.

```
public class PrintStream extends FilterOutputStream {
    // Public Constructors
        #   public PrintStream(OutputStream out);
        #   public PrintStream(OutputStream out, boolean autoFlush);
    // Public Instance Methods
            public boolean checkError();
            public void close();  // Overrides FilterOutputStream
            public void flush();  // Overrides FilterOutputStream
            public void print(boolean b);
            public void print(char c);
            public void print(int i);
            public void print(long l);
            public void print(float f);
            public void print(double d);
            public void print(char[] s);
            public void print(String s);
            public void print(Object obj);
            public void println();
            public void println(boolean x);
            public void println(char x);
            public void println(int x);
            public void println(long x);
            public void println(float x);
            public void println(double x);
            public void println(char[] x);
            public void println(String x);
            public void println(Object x);
            public void write(int b);  // Overrides FilterOutputStream
            public void write(byte[] buf, int off, int len);  // Overrides
FilterOutputStream
    // Protected Instance Methods
        1.1  protected void setError();
}
```

Hierarchy:

Object->OutputStream->FilterOutputStream->PrintStream

Passed To:

Component.list(), Container.list(), Properties.list(), System.setErr(), System.setOut(), Throwable.printStackTrace()

Type Of:

System.err, System.out

PREVIOUS | HOME | NEXT
java.io.PipedWriter (JDK 1.1) | BOOK INDEX | java.io.PrintWriter (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

24.53 java.io.PrintWriter (JDK 1.1)

This class is a character output stream that implements a number of `print()` and `println()` methods that output textual representations of primitive values and objects. When you create a `PrintWriter` object, you specify a character or byte output stream that it should write its characters to, and also optionally specify whether the `PrintWriter` stream should be automatically flushed whenever `println()` is called. If you specify a byte output stream as the destination, the `PrintWriter()` constructor automatically creates the necessary `OutputStreamWriter` object to convert characters to bytes using the default encoding.

`PrintWriter` implements the normal `write()`, `flush()`, and `close()` methods that all `Writer` subclasses do. It is more common to use the higher-level `print()` and `println()` methods, each of which converts its argument to a string before outputting it. `println()` has the additional behavior of terminating the line (and optionally flushing the buffer) after printing its argument.

The methods of `PrintWriter` never throw exceptions. Instead, when errors occur, they set an internal flag that you can check by calling `checkError()`. `checkError()` first flushes the internal stream, and then returns `true` if any exception has occurred while writing values to that stream. Once an error has occurred on a `PrintWriter` object, all subsequent calls to `checkError()` return `true`; there is no way to reset the error flag.

`PrintWriter` is the character stream analog to `PrintStream`, which it supersedes. You can usually trivially replace any `PrintStream` objects in a program with `PrintWriter` objects. This is particularly important for internationalized programs. The only valid remaining use for the `PrintStream` class is for the `System.out` and `System.err` standard output streams. See `PrintStream` for details.

```
public class PrintWriter extends Writer {
    // Public Constructors
            public PrintWriter(Writer out);
            public PrintWriter(Writer out, boolean autoFlush);
            public PrintWriter(OutputStream out);
            public PrintWriter(OutputStream out, boolean autoFlush);
    // Public Instance Methods
            public boolean checkError();
            public void close();  // Defines Writer
            public void flush();  // Defines Writer
            public void print(boolean b);
            public void print(char c);
            public void print(int i);
            public void print(long l);
            public void print(float f);
            public void print(double d);
            public void print(char[] s);
            public void print(String s);
            public void print(Object obj);
            public void println();
            public void println(boolean x);
            public void println(char x);
            public void println(int x);
            public void println(long x);
            public void println(float x);
            public void println(double x);
            public void println(char[] x);
            public void println(String x);
            public void println(Object x);
            public void write(int c);  // Overrides Writer
            public void write(char[] buf, int off, int len);  // Defines Writer
            public void write(char[] buf);  // Overrides Writer
            public void write(String s, int off, int len);  // Overrides Writer
            public void write(String s);  // Overrides Writer
    // Protected Instance Methods
            protected void setError();
}
```

Hierarchy:

Object->Writer->PrintWriter

Passed To:

Component.list(), Container.list(), Properties.list(), Throwable.printStackTrace()

PREVIOUS | HOME | NEXT
java.io.PrintStream (JDK 1.0) | BOOK INDEX | java.io.PushbackInputStream (JDK 1.0)

22.6 java.awt.peer.ComponentPeer (JDK 1.0)

```
public abstract interface ComponentPeer {
    // Public Instance Methods
            public abstract int checkImage(Image img, int w, int h, ImageObserver o);
            public abstract Image createImage(ImageProducer producer);
            public abstract Image createImage(int width, int height);
            public abstract void disable();
            public abstract void dispose();
            public abstract void enable();
            public abstract ColorModel getColorModel();
            public abstract FontMetrics getFontMetrics(Font font);
            public abstract Graphics getGraphics();
        1.1  public abstract Point getLocationOnScreen();
        1.1  public abstract Dimension getMinimumSize();
        1.1  public abstract Dimension getPreferredSize();
            public abstract Toolkit getToolkit();
        1.1  public abstract void handleEvent(AWTEvent e);
            public abstract void hide();
        1.1  public abstract boolean isFocusTraversable();
            public abstract Dimension minimumSize();
            public abstract void paint(Graphics g);
            public abstract Dimension preferredSize();
            public abstract boolean prepareImage(Image img, int w, int h, 
ImageObserver o);
            public abstract void print(Graphics g);
            public abstract void repaint(long tm, int x, int y, int width, int 
height);
            public abstract void requestFocus();
            public abstract void reshape(int x, int y, int width, int height);
            public abstract void setBackground(Color c);
        1.1  public abstract void setBounds(int x, int y, int width, int height);
        1.1  public abstract void setCursor(Cursor cursor);
        1.1  public abstract void setEnabled(boolean b);
            public abstract void setFont(Font f);
            public abstract void setForeground(Color c);
        1.1  public abstract void setVisible(boolean b);
            public abstract void show();
}
```

Extended By:

ButtonPeer, CanvasPeer, CheckboxPeer, ChoicePeer, ContainerPeer, LabelPeer, LightweightPeer, ListPeer, ScrollbarPeer, TextComponentPeer

Returned By:

Component.getPeer()

PREVIOUS | HOME | NEXT

java.awt.peer.ChoicePeer (JDK 1.0) | BOOK INDEX | java.awt.peer.ContainerPeer (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

22.21 java.awt.peer.ScrollPanePeer (JDK 1.1)

```
public abstract interface ScrollPanePeer extends ContainerPeer {
    // Public Instance Methods
            public abstract void childResized(int w, int h);
            public abstract int getHScrollbarHeight();
            public abstract int getVScrollbarWidth();
            public abstract void setScrollPosition(int x, int y);
            public abstract void setUnitIncrement(Adjustable adj, int u);
            public abstract void setValue(Adjustable adj, int v);
}
```

Hierarchy:

(ScrollPanePeer(ContainerPeer(ComponentPeer)))

Returned By:

Toolkit.createScrollPane()

24.31 java.io.InvalidClassException (JDK 1.1)

Signals that the serialization mechanism has encountered one of several possible problems with the class of an object that is being serialized or deserialized. The `classname` field should contain the name of the class in question, and the `getMessage()` method is overridden to return this class name with the message.

```
public class InvalidClassException extends ObjectStreamException {
    // Public Constructors
            public InvalidClassException(String reason);
            public InvalidClassException(String cname, String reason);
    // Public Instance Variables
            public String classname;
    // Public Instance Methods
            public String getMessage();  // Overrides Throwable
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->IOException->ObjectStreamException->InvalidClassException

30.10 java.util.Hashtable (JDK 1.0)

This class implements a hashtable data structure, which allows you to associate values with a key and to efficiently look up the value associated with a given key. A hashtable is essentially an associative array, which stores objects with non-numeric array indices.

`put()` associates a value with a key in a `Hashtable`. `get()` retrieves a value for a specified key. `remove()` deletes a key/value association. `keys()` and `elements()` return `Enumeration` objects that allow you to iterate through the complete set of keys and values stored in the table.

```
public class Hashtable extends Dictionary implements Cloneable, Serializable {
    // Public Constructors
            public Hashtable(int initialCapacity, float loadFactor);
            public Hashtable(int initialCapacity);
            public Hashtable();
    // Public Instance Methods
            public synchronized void clear();
            public synchronized Object clone();  // Overrides Object
            public synchronized boolean contains(Object value);
            public synchronized boolean containsKey(Object key);
            public synchronized Enumeration elements();  // Defines Dictionary
            public synchronized Object get(Object key);  // Defines Dictionary
            public boolean isEmpty();  // Defines Dictionary
            public synchronized Enumeration keys();  // Defines Dictionary
            public synchronized Object put(Object key, Object value);  // Defines 
Dictionary
            public synchronized Object remove(Object key);  // Defines Dictionary
            public int size();  // Defines Dictionary
            public synchronized String toString();  // Overrides Object
    // Protected Instance Methods
            protected void rehash();
}
```

Hierarchy:

Object->Dictionary->Hashtable(Cloneable, Serializable)

Extended By:

Properties

Passed To:

CropImageFilter.setProperties(), ImageConsumer.setProperties(), ImageFilter.setProperties(), MemoryImageSource(), PixelGrabber.setProperties(), ReplicateScaleFilter.setProperties()

Type Of:

GridBagLayout.comptable

PREVIOUS
java.util.GregorianCalendar (JDK 1.1)

HOME
BOOK INDEX

NEXT
java.util.ListResourceBundle (JDK 1.1)

18.27 java.awt.Graphics (JDK 1.0)

This abstract class defines a device-independent interface to graphics. It specifies methods for doing line drawing, area filling, image painting, area copying, and graphics output clipping. Specific subclasses of `Graphics` are implemented for different platforms and different graphics output devices. A `Graphics` object cannot be created directly through a constructor--it must be obtained with the `getGraphics()` method of a `Component` or an `Image`, or copied from an existing `Graphics` object with `create()`. When a `Graphics` object is no longer needed, you should call `dispose()` to free up the window system resources it uses.

```
public abstract class Graphics extends Object {
    // Protected Constructor
            protected Graphics();
    // Public Instance Methods
            public abstract void clearRect(int x, int y, int width, int height);
            public abstract void clipRect(int x, int y, int width, int height);
            public abstract void copyArea(int x, int y, int width, int height, int
dx, int dy);
            public abstract Graphics create();
            public Graphics create(int x, int y, int width, int height);
            public abstract void dispose();
            public void draw3DRect(int x, int y, int width, int height, boolean
raised);
            public abstract void drawArc(int x, int y, int width, int height, int
startAngle, int arcAngle);
            public void drawBytes(byte[] data, int offset, int length, int x, int y);
            public void drawChars(char[] data, int offset, int length, int x, int y);
            public abstract boolean drawImage(Image img, int x, int y, ImageObserver
observer);
            public abstract boolean drawImage(Image img, int x, int y, int width, int
height, ImageObserver observer);
            public abstract boolean drawImage(Image img, int x, int y, Color bgcolor,
ImageObserver observer);
            public abstract boolean drawImage(Image img, int x, int y, int width, int
height, Color bgcolor,
            public abstract boolean drawImage'u'ImageObserver observer);
        1.1  public abstract boolean drawImage(Image img, int dx1, int dy1, int dx2,
int dy2, int sx1, int sy1,
        1.1  public abstract boolean drawImage'u'int sx2, int sy2, ImageObserver
observer);
        1.1  public abstract boolean drawImage(Image img, int dx1, int dy1, int dx2,
int dy2, int sx1, int sy1,
        1.1  public abstract boolean drawImage'u'int sx2, int sy2, Color bgcolor,
ImageObserver observer);
            public abstract void drawLine(int x1, int y1, int x2, int y2);
            public abstract void drawOval(int x, int y, int width, int height);
            public abstract void drawPolygon(int[] xPoints, int[] yPoints, int
nPoints);
            public void drawPolygon(Polygon p);
        1.1  public abstract void drawPolyline(int[] xPoints, int[] yPoints, int
nPoints);
            public void drawRect(int x, int y, int width, int height);
            public abstract void drawRoundRect(int x, int y, int width, int height,
int arcWidth, int arcHeight);
            public abstract void drawString(String str, int x, int y);
            public void fill3DRect(int x, int y, int width, int height, boolean
raised);
            public abstract void fillArc(int x, int y, int width, int height, int
startAngle, int arcAngle);
            public abstract void fillOval(int x, int y, int width, int height);
            public abstract void fillPolygon(int[] xPoints, int[] yPoints, int
nPoints);
            public void fillPolygon(Polygon p);
            public abstract void fillRect(int x, int y, int width, int height);
            public abstract void fillRoundRect(int x, int y, int width, int height,
int arcWidth, int arcHeight);
            public void finalize();  // Overrides Object
        1.1  public abstract Shape getClip();
        1.1  public abstract Rectangle getClipBounds();
        #   public Rectangle getClipRect();
            public abstract Color getColor();
            public abstract Font getFont();
            public FontMetrics getFontMetrics();
            public abstract FontMetrics getFontMetrics(Font f);
        1.1  public abstract void setClip(int x, int y, int width, int height);
        1.1  public abstract void setClip(Shape clip);
            public abstract void setColor(Color c);
            public abstract void setFont(Font font);
            public abstract void setPaintMode();
            public abstract void setXORMode(Color c1);
            public String toString();  // Overrides Object
            public abstract void translate(int x, int y);
}
```

Passed To:

Canvas.paint(), Component.paint(), Component.paintAll(), Component.print(), Component.printAll(), Component.update(), ComponentPeer.paint(), ComponentPeer.print(), Container.paint(), Container.paintComponents(), Container.print(), Container.printComponents(), PropertyEditor.paintValue(), PropertyEditorSupport.paintValue(), ScrollPane.printComponents()

Returned By:

`Component.getGraphics(), ComponentPeer.getGraphics(), Graphics.create(), Image.getGraphics(), PrintJob.getGraphics()`

PREVIOUS | HOME | NEXT
java.awt.Frame (JDK 1.0) | BOOK INDEX | java.awt.GridBagConstraints (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

19. The java.awt.datatransfer Package

Contents:

This small package contains classes and interfaces that support a generic inter-application data transfer mechanism. It also provides support for cut-and-paste data transfer on top of that mechanism. This package, and all of its classes and interfaces, are new in Java 1.1. Future releases of Java are likely to extend this package with support for data transfer through drag-and-drop.

Figure 19.1 shows the class hierarchy for `java.datatransfer`. `DataFlavor` and `Transferable` define the basic data transfer mechanism. `Clipboard` and `ClipboardOwner` provide support for cut-and-paste. `StringSelection` is a convenience class that makes it particularly easy to transfer textual data between applications.

Figure 19.1: The java.awt.datatransfer package

[Graphic: Figure 19-1]

19.1 java.awt.datatransfer.Clipboard (JDK 1.1)

This class represents a clipboard on which data may be transferred using the cut-and-paste metaphor. When data is "cut," it should be encapsulated in a `Transferable` object and registered with a `Clipboard` object by calling `setContents()`. A `Clipboard` can only hold a single piece of data at a time, so a `ClipboardOwner` object must be specified when data is placed on the clipboard. This object is notified that it no longer "owns" the clipboard when the data is replaced by other, more recent, data.

When a "paste" is requested by the user, an application requests the data on the `Clipboard` by calling `getContents()`, which returns a `Transferable` object. The methods of this object can be used to negotiate a mutually-compatible data format and to actually transfer the data.

A clipboard name is passed to the `Clipboard()` constructor, and may be retrieved with `getName()`. This name is not actually used in Java 1.1, however.

Note that while applications can create their own private `Clipboard` objects for intra-application cut-and-paste, it is more common for them to use the system clipboard to enable cut-and-paste between applications. You can obtain the system clipboard by calling the `getSystemClipboard()` method of the current `Toolkit` object. Untrusted applet code is not allowed to access the system clipboard, so untrusted applets cannot participate in inter-application cut-and-paste.

```
public class Clipboard extends Object {
    // Public Constructor
            public Clipboard(String name);
    // Protected Instance Variables
            protected Transferable contents;
            protected ClipboardOwner owner;
    // Public Instance Methods
            public synchronized Transferable getContents(Object requestor);
            public String getName();
            public synchronized void setContents(Transferable contents,
ClipboardOwner owner);
}
```

Passed To:

`ClipboardOwner.lostOwnership(), StringSelection.lostOwnership()`

Returned By:

`Toolkit.getSystemClipboard()`

PREVIOUS
java.awt.Window (JDK 1.0)

HOME
BOOK INDEX

NEXT
java.awt.datatransfer.ClipboardOwner (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

29.2 java.text.CharacterIterator (JDK 1.1)

This interface defines an API for portably iterating through the characters that comprise a string of text, regardless of the encoding of that text. Such an API is necessary because the number of bytes per character is different for different encodings, and some encodings even use variable-width characters within the same string of text. In addition to allowing iteration, a class that implements the `CharacterIterator` interface for non-Unicode text also performs translation of characters from their native encoding to standard Java Unicode characters.

`CharacterIterator` is similar to `java.util.Enumeration`, but is somewhat more complex than that interface. The `first()` and `last()` methods return the first and last characters in the text, and the `next()` and `prev()` methods allow you to loop forward or backwards through the characters of the text. These methods return the `DONE` constant when they go beyond the first or last character in the text--a test for this constant can be used to terminate a loop.

The `CharacterIterator` interface also allows random access to the characters in a string of text. The `getBeginIndex()` and `getEndIndex()` methods return the character positions for the start and end of the string, and `setIndex()` sets the current position. `getIndex()` returns the index of the current position, and `current()` returns the character at that position.

```
public abstract interface CharacterIterator extends Cloneable {
    // Constants
            public static final char DONE;
    // Public Instance Methods
            public abstract Object clone();  // Overrides Object
            public abstract char current();
            public abstract char first();
            public abstract int getBeginIndex();
            public abstract int getEndIndex();
            public abstract int getIndex();
            public abstract char last();
            public abstract char next();
            public abstract char previous();
            public abstract char setIndex(int position);
}
```

Implemented By:

StringCharacterIterator

Passed To:

BreakIterator.setText()

Returned By:

BreakIterator.getText()

PREVIOUS java.text.BreakIterator (JDK 1.1) | HOME | BOOK INDEX | NEXT java.text.ChoiceFormat (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

29.8 java.text.DateFormatSymbols (JDK 1.1)

This class defines accessor methods for the various pieces of data, such as names of months and days, used by `SimpleDateFormat` to format and parse dates and times. You do not typically use this class unless you are formatting dates for an unsupported locale or in some highly customized way.

```
public class DateFormatSymbols extends Object implements Serializable, Cloneable {
    // Public Constructors
            public DateFormatSymbols();
            public DateFormatSymbols(Locale locale);
    // Public Instance Methods
            public Object clone();  // Overrides Object
            public boolean equals(Object obj);  // Overrides Object
            public String[] getAmPmStrings();
            public String[] getEras();
            public String getLocalPatternChars();
            public String[] getMonths();
            public String[] getShortMonths();
            public String[] getShortWeekdays();
            public String[] getWeekdays();
            public String[][] getZoneStrings();
            public int hashCode();  // Overrides Object
            public void setAmPmStrings(String[] newAmpms);
            public void setEras(String[] newEras);
            public void setLocalPatternChars(String newLocalPatternChars);
            public void setMonths(String[] newMonths);
            public void setShortMonths(String[] newShortMonths);
            public void setShortWeekdays(String[] newShortWeekdays);
            public void setWeekdays(String[] newWeekdays);
            public void setZoneStrings(String[][] newZoneStrings);
}
```

Passed To:

SimpleDateFormat(), SimpleDateFormat.setDateFormatSymbols()

Returned By:

SimpleDateFormat.getDateFormatSymbols()

PREVIOUS
java.text.DateFormat (JDK 1.1)

HOME
BOOK INDEX

NEXT
java.text.DecimalFormat (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

29.10 java.text.DecimalFormatSymbols (JDK 1.1)

This class defines the various characters and strings, such as the decimal point, percent sign, and thousands separator, used by `DecimalFormat` when formatting numbers. You do not typically use this class directly unless you are formatting dates for an unsupported locale or in some highly customized way.

```
public final class DecimalFormatSymbols extends Object implements Cloneable,
Serializable {
    // Public Constructors
            public DecimalFormatSymbols();
            public DecimalFormatSymbols(Locale locale);
    // Public Instance Methods
            public Object clone();  // Overrides Object
            public boolean equals(Object obj);  // Overrides Object
            public char getDecimalSeparator();
            public char getDigit();
            public char getGroupingSeparator();
            public String getInfinity();
            public char getMinusSign();
            public String getNaN();
            public char getPatternSeparator();
            public char getPerMill();
            public char getPercent();
            public char getZeroDigit();
            public int hashCode();  // Overrides Object
            public void setDecimalSeparator(char decimalSeparator);
            public void setDigit(char digit);
            public void setGroupingSeparator(char groupingSeparator);
            public void setInfinity(String infinity);
            public void setMinusSign(char minusSign);
            public void setNaN(String NaN);
            public void setPatternSeparator(char patternSeparator);
            public void setPerMill(char perMill);
            public void setPercent(char percent);
            public void setZeroDigit(char zeroDigit);
}
```

Passed To:

DecimalFormat(), DecimalFormat.setDecimalFormatSymbols()

Returned By:

DecimalFormat.getDecimalFormatSymbols()

29.12 java.text.Format (JDK 1.1)

This abstract class is the base class for all number, date, and string formatting classes in the `java.text` package. It defines two abstract methods that are implemented by subclasses. `format()` converts an object to a string using the formatting rules encapsulated by the `Format` subclass and optionally appends the resulting string to an existing `StringBuffer`. `parseObject()` performs the reverse operation--it parses a formatted string and returns the corresponding object. Status information for these two operations is returned in `FieldPosition` and `ParsePosition` objects. The non-abstract methods of this class are simple shortcuts that rely on implementations of the abstract methods.

See `ChoiceFormat`, `DateFormat`, `MessageFormat`, and `NumberFormat`.

```
public abstract class Format extends Object implements Serializable, Cloneable {
    // Default Constructor: public Format()
    // Public Instance Methods
            public Object clone();  // Overrides Object
            public final String format(Object obj);
            public abstract StringBuffer format(Object obj, StringBuffer toAppendTo,
FieldPosition pos);
            public abstract Object parseObject(String source, ParsePosition status);
            public Object parseObject(String source) throws ParseException;
}
```

Extended By:

DateFormat, MessageFormat, NumberFormat

Passed To:

MessageFormat.setFormat(), MessageFormat.setFormats()

Returned By:

MessageFormat.getFormats()

21.7 java.awt.image.ImageFilter (JDK 1.0)

This class is used in conjunction with a `FilteredImageSource`. It accepts image data through the `ImageConsumer` interface and passes it on to an `ImageConsumer` specified by the controlling `FilteredImageSource`. `ImageFilter` is the superclass of all image filters, and performs no filtering itself. You must subclass it to perform the desired filtering. See `CropImageFilter` and `RGBImageFilter`. The `ImageFilter` methods are the `ImageConsumer` methods invoked by an `ImageProducer`. You should not call them directly.

See `FilteredImageSource` for an example of using an `ImageFilter`.

```
public class ImageFilter extends Object implements ImageConsumer, Cloneable {
    // Default Constructor: public ImageFilter()
    // Protected Instance Variables
            protected ImageConsumer consumer;
    // Public Instance Methods
            public Object clone();  // Overrides Object
            public ImageFilter getFilterInstance(ImageConsumer ic);
            public void imageComplete(int status);  // From ImageConsumer
            public void resendTopDownLeftRight(ImageProducer ip);
            public void setColorModel(ColorModel model);  // From ImageConsumer
            public void setDimensions(int width, int height);  // From ImageConsumer
            public void setHints(int hints);  // From ImageConsumer
            public void setPixels(int x, int y, int w, int h, ColorModel model,
            public void setPixels'u'byte[] pixels, int off, int scansize);  // From
ImageConsumer
            public void setPixels(int x, int y, int w, int h, ColorModel model,
            public void setPixels'u'int[] pixels, int off, int scansize);  // From
ImageConsumer
            public void setProperties(Hashtable props);  // From ImageConsumer
}
```

Extended By:

`CropImageFilter, ReplicateScaleFilter, RGBImageFilter`

Passed To:

`FilteredImageSource()`

Returned By:

`ImageFilter.getFilterInstance()`

29.14 java.text.NumberFormat (JDK 1.1)

This class formats and parses numbers in a locale-specific way. As an abstract class, it cannot be instantiated directly, but it provides a number of static methods that return instances of a concrete subclass which you can use for formatting. The `getInstance()` method returns a `NumberFormat` object suitable for normal formatting of numbers in either the default locale or in a specified locale. `getCurrencyInstance()` and `getPercentInstance()` return `NumberFormat` objects for formatting numbers that represent monetary amounts and percentages in either the default locale or in a specified locale. `getAvailableLocales()` returns an array of locales for which `NumberFormat` objects are available.

Once you have created a suitable `NumberFormat` object, you may customize its locale-independent behavior with `setMaximumFractionDigits()`, `setGroupingUsed()` and similar `set` methods. In order to customize the locale-dependent behavior, you can use `instanceof` to test if the `NumberFormat` object is an instance of `DecimalFormat`, and if so, cast it to that type. The `DecimalFormat` class provides complete control over number formatting. Note, however, that a `NumberFormat` customized in this way may no longer be appropriate for the desired locale.

After creating and customizing a `NumberFormat` object, you can use the various `format()` methods to convert numbers to strings or string buffers, and you can use the `parse()` or `parseObject()` methods to convert strings to numbers.

The constants defined by this class are intended to be used by the `FieldPosition` object.

The `NumberFormat` class in not intended for the display of very large or very small numbers that require exponential notation, and it may not gracefully handle infinite or `NaN` (not-a-number) values.

```
public abstract class NumberFormat extends Format implements Cloneable {
    // Default Constructor: public NumberFormat()
    // Constants
            public static final int FRACTION_FIELD;
            public static final int INTEGER_FIELD;
    // Class Methods
            public static Locale[] getAvailableLocales();
            public static final NumberFormat getCurrencyInstance();
            public static NumberFormat getCurrencyInstance(Locale inLocale);
            public static final NumberFormat getInstance();
            public static NumberFormat getInstance(Locale inLocale);
            public static final NumberFormat getNumberInstance();
            public static NumberFormat getNumberInstance(Locale inLocale);
            public static final NumberFormat getPercentInstance();
            public static NumberFormat getPercentInstance(Locale inLocale);
    // Public Instance Methods
            public Object clone();  // Overrides Format
            public boolean equals(Object obj);  // Overrides Object
            public final StringBuffer format(Object number, StringBuffer toAppendTo,
FieldPosition pos);  // Defines Format
            public final String format(double number);
            public final String format(long number);
            public abstract StringBuffer format(double number, StringBuffer
toAppendTo, FieldPosition pos);
            public abstract StringBuffer format(long number, StringBuffer toAppendTo,
FieldPosition pos);
            public int getMaximumFractionDigits();
            public int getMaximumIntegerDigits();
            public int getMinimumFractionDigits();
            public int getMinimumIntegerDigits();
            public int hashCode();  // Overrides Object
            public boolean isGroupingUsed();
            public boolean isParseIntegerOnly();
            public abstract Number parse(String text, ParsePosition parsePosition);
            public Number parse(String text) throws ParseException;
            public final Object parseObject(String source, ParsePosition
parsePosition);  // Defines Format
            public void setGroupingUsed(boolean newValue);
            public void setMaximumFractionDigits(int newValue);
            public void setMaximumIntegerDigits(int newValue);
            public void setMinimumFractionDigits(int newValue);
            public void setMinimumIntegerDigits(int newValue);
            public void setParseIntegerOnly(boolean value);
}
```

Hierarchy:

Object->Format(Serializable, Cloneable)->NumberFormat(Cloneable)

Extended By:

ChoiceFormat, DecimalFormat

Passed To:

DateFormat.setNumberFormat()

Returned By:

DateFormat.getNumberFormat(), NumberFormat.getCurrencyInstance(), NumberFormat.getInstance(), NumberFormat.getNumberInstance(), NumberFormat.getPercentInstance()

Type Of:

DateFormat.numberFormat

PREVIOUS | HOME | NEXT
java.text.MessageFormat (JDK 1.1) | BOOK INDEX | java.text.ParseException (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

25.47 java.lang.Object (JDK 1.0)

This is the root class in Java. All classes are subclasses of `Object`, and thus all objects can invoke the `public` and `protected` methods of this class. `equals()` tests whether two objects have the same value (i.e., not whether two variables refer to the same object, but whether two distinct objects have byte-for-byte equivalence). For classes that implement the `Cloneable` interface, `clone()` makes a byte-for-byte copy of an `Object`. `getClass()` returns the `Class` object associated with any `Object`, and the `notify()`, `notifyAll()`, and `wait()` methods are used for thread synchronization on a given `Object`.

A number of these `Object` methods should be overridden by subclasses of `Object`. Subclasses should provide their own definition of the `toString()` method so that they can be used with the string concatenation operator and with the `PrintWriter.println()` methods. Defining the `toString()` method for all objects also helps with debugging.

Classes that contain references to other objects may want to override the `equals()` and `clone()` methods (for `Cloneable` objects) so that they recursively call the `equals()` and `clone()` methods of the objects referred to within the original object. Some classes, particularly those that override `equals()`, may also want to override the `hashCode()` method to provide an appropriate hashcode to be used when storing instances in a `Hashtable` data structure.

Classes that allocate system resources other than memory (such as file descriptors or windowing system graphic contexts) should override the `finalize()` method to release these resources when the object is no longer referred to and is about to be garbage collected.

```
public class Object {
    // Default Constructor: public Object()
    // Public Instance Methods
            public boolean equals(Object obj);
            public final native Class getClass();
            public native int hashCode();
            public final native void notify();
            public final native void notifyAll();
            public String toString();
            public final native void wait(long timeout) throws InterruptedException;
            public final void wait(long timeout, int nanos) throws 
InterruptedException;
            public final void wait() throws InterruptedException;
    // Protected Instance Methods
            protected native Object clone() throws CloneNotSupportedException;
            protected void finalize() throws Throwable;
}
```

Extended By:

```
Many classes
```

Passed To:

```
Many methods
```

Returned By:

```
Many methods
```

Type Of:

Event.arg, Event.target, EventObject.source, Image.UndefinedProperty, Reader.lock, ReplicateScaleFilter.outpixbuf, Vector.elementData, Writer.lock

PREVIOUS | HOME | NEXT
java.lang.NumberFormatException (JDK 1.0) | BOOK INDEX | java.lang.OutOfMemoryError (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

29.17 java.text.RuleBasedCollator (JDK 1.1)

This class is a concrete subclass of the abstract `Collator` class. It performs collations using a table of rules that are specified in textual form. Most applications do not use this class directly; instead they call `Collator.getInstance()` to obtain a `Collator` object (typically a `RuleBasedCollator` object) that implements the default collation order for a specified or default locale. You should only need to use this class if you are collating strings for a locale that is not supported by default, or if you need to implement a highly customized collation order.

```
public class RuleBasedCollator extends Collator {
    // Public Constructor
            public RuleBasedCollator(String rules) throws ParseException;
    // Public Instance Methods
            public Object clone();  // Overrides Collator
            public int compare(String source, String target);  // Defines Collator
            public boolean equals(Object obj);  // Overrides Collator
            public CollationElementIterator getCollationElementIterator(String 
source);
            public CollationKey getCollationKey(String source);  // Defines Collator
            public String getRules();
            public int hashCode();  // Defines Collator
}
```

Hierarchy:

Object->Collator(Cloneable, Serializable)->RuleBasedCollator

30.21 java.util.SimpleTimeZone (JDK 1.1)

This concrete subclass of `TimeZone` is a simple implementation of that abstract class, and is suitable for use in locales that use the Gregorian calendar. Programs do not usually need to instantiate this class directly; instead, they use one of the static "factory" methods of `TimeZone` to obtain a suitable `TimeZone` subclass.

You would instantiate this class directly only if you needed to support a time zone with nonstandard daylight savings time rules. In that case, you would use call `setStartRule()` and `setEndRule()` to specify the starting and ending dates of daylight savings time for the time zone.

```
public class SimpleTimeZone extends TimeZone {
    // Public Constructors
            public SimpleTimeZone(int rawOffset, String ID);
            public SimpleTimeZone(int rawOffset, String ID, int startMonth, int 
startDayOfWeekInMonth, int startDayOfWeek, int startTime, int endMonth, int 
endDayOfWeekInMonth, int endDayOfWeek, int endTime);
    // Public Instance Methods
            public Object clone();  // Overrides TimeZone
            public boolean equals(Object obj);  // Overrides Object
            public int getOffset(int era, int year, int month, int day, int 
dayOfWeek, int millis);  // Defines TimeZone
            public int getRawOffset();  // Defines TimeZone
            public synchronized int hashCode();  // Overrides Object
            public boolean inDaylightTime(Date date);  // Defines TimeZone
            public void setEndRule(int month, int dayOfWeekInMonth, int dayOfWeek, 
int time);
            public void setRawOffset(int offsetMillis);  // Defines TimeZone
            public void setStartRule(int month, int dayOfWeekInMonth, int dayOfWeek, 
int time);
            public void setStartYear(int year);
            public boolean useDaylightTime();  // Defines TimeZone
}
```

Hierarchy:

Object->TimeZone(Serializable, Cloneable)->SimpleTimeZone

29.19 java.text.StringCharacterIterator (JDK 1.1)

This class is a trivial implementation of the `CharacterIterator` interface that works for text stored in Java `String` objects. See `CharacterIterator` for details.

```
public final class StringCharacterIterator extends Object implements 
CharacterIterator {
    // Public Constructors
            public StringCharacterIterator(String text);
            public StringCharacterIterator(String text, int pos);
            public StringCharacterIterator(String text, int begin, int end, int pos);
    // Public Instance Methods
            public Object clone();  // Overrides Object
            public char current();  // From CharacterIterator
            public boolean equals(Object obj);  // Overrides Object
            public char first();  // From CharacterIterator
            public int getBeginIndex();  // From CharacterIterator
            public int getEndIndex();  // From CharacterIterator
            public int getIndex();  // From CharacterIterator
            public int hashCode();  // Overrides Object
            public char last();  // From CharacterIterator
            public char next();  // From CharacterIterator
            public char previous();  // From CharacterIterator
            public char setIndex(int p);  // From CharacterIterator
}
```

Hierarchy:

Object->StringCharacterIterator(CharacterIterator(Cloneable))

30.24 java.util.TimeZone (JDK 1.1)

The `TimeZone` class represents a time zone; it is used with the `Calendar` and `DateFormat` classes.

As an abstract class, `TimeZone` cannot be directly instantiated. Instead, you should call the static `getDefault()` method to obtain a `TimeZone` object that represents the time zone inherited from the host operating system. Or, you should call `getTimeZone()` (also static), passing the name of the desired zone. You can obtain a list of the supported time zone names by calling the static `getAvailableIDs()` method.

Once you have a `TimeZone` object, you can call `inDaylightTime()` to determine whether, for a given `Date`, daylight savings time is in effect for that time zone. Call `getID()` to obtain the name of the time zone, and call `getOffset()` for a given date to determine the number of milliseconds to add to GMT to convert to the time zone.

```
public abstract class TimeZone extends Object implements Serializable, Cloneable {
    // Default Constructor: public TimeZone()
    // Class Methods
            public static synchronized String[] getAvailableIDs(int rawOffset);
            public static synchronized String[] getAvailableIDs();
            public static synchronized TimeZone getDefault();
            public static synchronized TimeZone getTimeZone(String ID);
            public static synchronized void setDefault(TimeZone zone);
    // Public Instance Methods
            public Object clone();  // Overrides Object
            public String getID();
            public abstract int getOffset(int era, int year, int month, int day, int 
dayOfWeek, int milliseconds);
            public abstract int getRawOffset();
            public abstract boolean inDaylightTime(Date date);
            public void setID(String ID);
            public abstract void setRawOffset(int offsetMillis);
            public abstract boolean useDaylightTime();
}
```

Extended By:

SimpleTimeZone

Passed To:

Calendar(), Calendar.getInstance(), Calendar.setTimeZone(), DateFormat.setTimeZone(), GregorianCalendar(), TimeZone.setDefault()

Returned By:

Calendar.getTimeZone(), DateFormat.getTimeZone(), TimeZone.getDefault(), TimeZone.getTimeZone()

PREVIOUS	HOME	NEXT
java.util.StringTokenizer (JDK 1.0)	BOOK INDEX	java.util.TooManyListenersException (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

24.3 java.io.BufferedReader (JDK 1.1)

This class applies buffering to a character input stream, thereby improving the efficiency of character input. You create a `BufferedReader` by specifying some other character input stream that it is to buffer input from. (You can also specify a buffer size at this time, although the default size is usually fine.) Typically you use this sort of buffering when you are using a `FileReader` or `InputStreamReader`.

`BufferedReader` defines the standard set of `Reader` methods, and also provides a `readLine()` method that reads a line of text (not including the line-terminators) and returns it as a `String`.

`BufferedReader` is the character-stream analog of `BufferedInputStream`. It also provides a replacement for the deprecated `readLine()` method of `DataInputStream`, which did not properly convert bytes into characters.

```
public class BufferedReader extends Reader {
    // Public Constructors
            public BufferedReader(Reader in, int sz);
            public BufferedReader(Reader in);
    // Public Instance Methods
            public void close() throws IOException;  // Defines Reader
            public void mark(int readAheadLimit) throws IOException;  // Overrides
Reader
            public boolean markSupported();  // Overrides Reader
            public int read() throws IOException;  // Overrides Reader
            public int read(char[] cbuf, int off, int len) throws IOException;  //
Defines Reader
            public String readLine() throws IOException;
            public boolean ready() throws IOException;  // Overrides Reader
            public void reset() throws IOException;  // Overrides Reader
            public long skip(long n) throws IOException;  // Overrides Reader
}
```

Hierarchy:

Object->Reader->BufferedReader

Extended By:

LineNumberReader

PREVIOUS
java.io.BufferedOutputStream (JDK 1.0)

HOME
BOOK INDEX

NEXT
java.io.BufferedWriter (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

24.4 java.io.BufferedWriter (JDK 1.1)

This class applies buffering to a character output stream, improving output efficiency by coalescing many small write requests into a single larger request. You create a `BufferedWriter` by specifying some other character output stream to which it sends its buffered and coalesced output. (You can also specify a buffer size at this time, although the default size is usually satisfactory.) Typically you use this sort of buffering when you are using a `FileWriter` or `OutputStreamWriter`.

`BufferedWriter` defines the standard `write()`, `flush()`, and `close()` methods that all output streams define, but it also adds a `newLine()` method, which outputs the platform-dependent line separator (usually a newline character, a carriage return character, or both) to the stream.

`BufferedWriter` is the character-stream analog of `BufferedOutputStream`.

```
public class BufferedWriter extends Writer {
    // Public Constructors
            public BufferedWriter(Writer out);
            public BufferedWriter(Writer out, int sz);
    // Public Instance Methods
            public void close() throws IOException;  // Defines Writer
            public void flush() throws IOException;  // Defines Writer
            public void newLine() throws IOException;
            public void write(int c) throws IOException;  // Overrides Writer
            public void write(char[] cbuf, int off, int len) throws IOException;  //
Defines Writer
            public void write(String s, int off, int len) throws IOException;  //
Overrides Writer
}
```

Hierarchy:

Object->Writer->BufferedWriter

28.6 java.net.DatagramSocket (JDK 1.0)

This class defines a socket that can receive and send unreliable datagram packets over the network using the UDP protocol. A datagram is a very low-level networking interface: it is simply an array of bytes sent over the network. A datagram does not implement any kind of stream-based communication protocol, and there is no connection established between the sender and the receiver. Datagram packets are called "unreliable" because the protocol does not make any attempt to ensure that they arrived or to resend them if they did not. Thus, packets sent through a `DatagramSocket` are not guaranteed to arrive in the order sent, or to arrive at all. On the other hand, this low-overhead protocol makes datagram transmission very fast.

If a port is specified when the `DatagramSocket` is created, that port is used; otherwise, the system assigns a port. `getLocalPort()` returns the port number in use. `send()` sends a `DatagramPacket` through the socket. The packet must contain the destination address to which it should be sent. `receive()` waits for data to arrive at the socket and stores it, along with the address of the sender, into the specified `DatagramPacket`. `close()` closes the socket and frees the port it used for reuse. Once `close()` has been called, the `DatagramSocket` should not be used again.

See `Socket` and `URL` for higher-level interfaces to networking.

```
public class DatagramSocket extends Object {
    // Public Constructors
            public DatagramSocket() throws SocketException;
            public DatagramSocket(int port) throws SocketException;
        1.1public DatagramSocket(int port, InetAddress laddr) throws SocketException;
    // Public Instance Methods
            public void close();
        1.1public InetAddress getLocalAddress();
            public int getLocalPort();
        1.1public synchronized int getSoTimeout() throws SocketException;
            public synchronized void receive(DatagramPacket p) throws IOException;
            public void send(DatagramPacket p) throws IOException;
        1.1public synchronized void setSoTimeout(int timeout) throws SocketException;
}
```

Extended By:

MulticastSocket

24.20 java.io.FileOutputStream (JDK 1.0)

This class is a subclass of `OutputStream` that writes data to a file specified by name, or by a `File` or `FileDescriptor` object. `write()` writes a byte or array of bytes to the file. To write binary data, you typically use this class in conjunction with a `BufferedOutputStream` and a `DataOutputStream`. To write text, you typically use it with a `PrintWriter`, `BufferedWriter`, and an `OutputStreamWriter`. Use `close()` to close a `FileOutputStream` when no further output will be written to it.

```
public class FileOutputStream extends OutputStream {
    // Public Constructors
            public FileOutputStream(String name) throws IOException;
        1.1  public FileOutputStream(String name, boolean append) throws IOException;
            public FileOutputStream(File file) throws IOException;
            public FileOutputStream(FileDescriptor fdObj);
    // Public Instance Methods
            public native void close() throws IOException;  // Overrides OutputStream
            public final FileDescriptor getFD() throws IOException;
            public native void write(int b) throws IOException;  // Defines
OutputStream
            public void write(byte[] b) throws IOException;  // Overrides
OutputStream
            public void write(byte[] b, int off, int len) throws IOException;  //
Overrides OutputStream
    // Protected Instance Methods
            protected void finalize() throws IOException;  // Overrides Object
}
```

Hierarchy:

Object->OutputStream->FileOutputStream

24.25 java.io.FilterOutputStream (JDK 1.0)

This class provides method definitions required to filter the data to be written to the `OutputStream` specified when the `FilterOutputStream` is created. It must be subclassed to perform some sort of filtering operation and may not be instantiated directly. See the subclasses `BufferedOutputStream` and `DataOutputStream`.

```
public class FilterOutputStream extends OutputStream {
    // Public Constructor
            public FilterOutputStream(OutputStream out);
    // Protected Instance Variables
            protected OutputStream out;
    // Public Instance Methods
            public void close() throws IOException;  // Overrides OutputStream
            public void flush() throws IOException;  // Overrides OutputStream
            public void write(int b) throws IOException;  // Defines OutputStream
            public void write(byte[] b) throws IOException;  // Overrides 
OutputStream
            public void write(byte[] b, int off, int len) throws IOException;  // 
Overrides OutputStream
}
```

Hierarchy:

Object->OutputStream->FilterOutputStream

Extended By:

BufferedOutputStream, CheckedOutputStream, DataOutputStream, DeflaterOutputStream, PrintStream

24.26 java.io.FilterReader (JDK 1.1)

This abstract class is intended to act as a superclass for character input streams that read data from some other character input stream, filter it in some way, and then return the filtered data when their own `read()` methods are called.

`FilterReader` is declared `abstract`, so that it cannot be instantiated. But none of its methods are themselves abstract: they all simply call the requested operation on the input stream passed to the `FilterReader()` constructor. If you were allowed to instantiate a `FilterReader`, you'd find that it is a "null filter"--i.e., it simply reads characters from the specified input stream and returns them without filtering of any kind.

Because `FilterReader` implements a "null filter," it is an ideal superclass for classes that want to implement simple filters, but do not want to override all of the methods of `Reader`. In order to create your own filtered character input stream, you should subclass `FilterReader` and override both of its `read()` methods to perform the desired filtering operation. Note that you can implement one of the `read()` methods in terms of the other, and thus only implement the filtration once. Recall that the other `read()` methods defined by `Reader` are implemented in terms of these methods, so you do not need to override those. In some cases, you may also need to override other methods of `FilterReader` as well, and you may want to provide methods or constructors that are specific to your subclass.

`FilterReader` is the character-stream analog to `FilterInputStream`.

```
public abstract class FilterReader extends Reader {
    // Protected Constructor
            protected FilterReader(Reader in);
    // Protected Instance Variables
            protected Reader in;
    // Public Instance Methods
            public void close() throws IOException;  // Defines Reader
            public void mark(int readAheadLimit) throws IOException;  // Overrides
Reader
            public boolean markSupported();  // Overrides Reader
            public int read() throws IOException;  // Overrides Reader
            public int read(char[] cbuf, int off, int len) throws IOException;  //
Defines Reader
            public boolean ready() throws IOException;  // Overrides Reader
            public void reset() throws IOException;  // Overrides Reader
            public long skip(long n) throws IOException;  // Overrides Reader
}
```

Hierarchy:

Object->Reader->FilterReader

Extended By:

PushbackReader

PREVIOUS HOME NEXT
java.io.FilterOutputStream (JDK 1.0) BOOK INDEX java.io.FilterWriter (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

24.27 java.io.FilterWriter (JDK 1.1)

This abstract class is intended to act as a superclass for character output streams that filter the data written to them before writing it to some other character output stream.

`FilterWriter` is declared `abstract`, so that it cannot be instantiated. But none of its methods are themselves abstract: they all simply invoke the corresponding method on the output stream that was passed to the `FilterWriter` constructor. If you were allowed to instantiate a `FilterWriter` object, you'd find that it acts as a "null filter"--that it simply passes the characters written to it along, without any filtration.

Because `FilterWriter` implements a "null filter," it is an ideal superclass for classes that want to implement simple filters without having to override all of the methods of `Writer`. In order to create your own filtered character output stream, you should subclass `FilterWriter` and override all of its `write()` methods to perform the desired filtering operation. Note that you can implement two of the `write()` methods in terms of the third, and thus only implement your filtering algorithm once. In some cases, you may want to override other `Writer` methods as well, and you may often want to add other methods or constructors that are specific to your subclass.

`FilterWriter` is the character-stream analog of `FilterOutputStream`.

```
public abstract class FilterWriter extends Writer {
    // Protected Constructor
            protected FilterWriter(Writer out);
    // Protected Instance Variables
            protected Writer out;
    // Public Instance Methods
            public void close() throws IOException;  // Defines Writer
            public void flush() throws IOException;  // Defines Writer
            public void write(int c) throws IOException;  // Overrides Writer
            public void write(char[] cbuf, int off, int len) throws IOException;  //
Defines Writer
            public void write(String str, int off, int len) throws IOException;  //
Overrides Writer
}
```

Hierarchy:

Object->Writer->FilterWriter

PREVIOUS | HOME | NEXT
java.io.FilterReader (JDK 1.1) | BOOK INDEX | java.io.InputStream (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

31.9 java.util.zip.GZIPInputStream (JDK 1.1)

This class is a subclass of `InflaterInputStream` that reads and uncompresses data compressed in *gzip* format. To create a `GZIPInputStream`, you must simply specify the `InputStream` it is to read compressed data from, and optionally specify a buffer size for the internal decompression buffer. Once a `GZIPInputStream` is created, you can use the `read()` and `close()` methods as you would with any input stream.

```
public class GZIPInputStream extends InflaterInputStream {
    // Public Constructors
            public GZIPInputStream(InputStream in, int size) throws IOException;
            public GZIPInputStream(InputStream in) throws IOException;
    // Constants
            public static final int GZIP_MAGIC;
    // Protected Instance Variables
            protected CRC32 crc;
            protected boolean eos;
    // Public Instance Methods
            public void close() throws IOException;  // Overrides FilterInputStream
            public int read(byte[] buf, int off, int len) throws IOException;  //
Overrides InflaterInputStream
}
```

Hierarchy:

Object->InputStream->FilterInputStream->InflaterInputStream->GZIPInputStream

31.10 java.util.zip.GZIPOutputStream (JDK 1.1)

This class is a subclass of `DeflaterOutputStream` that compresses and writes data using the *gzip* file format. To create a `GZIPOutputStream`, you must specify the `OutputStream` that it is to write to, and may optionally specify a size for the internal compression buffer. Once the `GZIPOutputStream` is created, you can use the `write()` and `close()` methods as you would any other output stream.

```
public class GZIPOutputStream extends DeflaterOutputStream {
    // Public Constructors
            public GZIPOutputStream(OutputStream out, int size) throws IOException;
            public GZIPOutputStream(OutputStream out) throws IOException;
    // Protected Instance Variables
            protected CRC32 crc;
    // Public Instance Methods
            public void close() throws IOException;  // Overrides 
DeflaterOutputStream
            public void finish() throws IOException;  // Overrides 
DeflaterOutputStream
            public synchronized void write(byte[] buf, int off, int len) throws 
IOException;  // Overrides DeflaterOutputStream
}
```

Hierarchy:

Object->OutputStream->FilterOutputStream->DeflaterOutputStream->GZIPOutputStream

24.29 java.io.InputStreamReader (JDK 1.1)

This class is a character input stream that uses a byte input stream as its data source: it reads bytes from a specified `InputStream` and translates them into Unicode characters according to a particular platform- and locale-dependent character encoding. This is a very important internationalization feature in Java 1.1.

When you create an `InputStreamReader`, you specify an `InputStream` from which the `InputStreamReader` is to read bytes, and you also optionally specify the name of the character encoding used by those bytes. If you do not specify an encoding name, the `InputStreamReader` uses the default encoding for the default locale, which is usually the correct thing to do.

The `InputStreamReader` supports the standard `Reader` methods. It also has a `getEncoding()` method that returns the name of the encoding being used to convert bytes to characters.

```
public class InputStreamReader extends Reader {
    // Public Constructors
            public InputStreamReader(InputStream in);
            public InputStreamReader(InputStream in, String enc) throws 
UnsupportedEncodingException;
    // Public Instance Methods
            public void close() throws IOException;  // Defines Reader
            public String getEncoding();
            public int read() throws IOException;  // Overrides Reader
            public int read(char[] cbuf, int off, int len) throws IOException;  // 
Defines Reader
            public boolean ready() throws IOException;  // Overrides Reader
}
```

Hierarchy:

Object->Reader->InputStreamReader

Extended By:

FileReader

java.io.InputStream (JDK 1.0) BOOK INDEX java.io.InterruptedIOException (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

24.41 java.io.ObjectOutput (JDK 1.1)

This interface extends the `DataOutput` interface and adds methods for serializing objects and writing bytes and arrays of bytes.

```
public abstract interface ObjectOutput extends DataOutput {
    // Public Instance Methods
            public abstract void close() throws IOException;
            public abstract void flush() throws IOException;
            public abstract void write(int b) throws IOException;   // From 
DataOutput
            public abstract void write(byte[] b) throws IOException;  // From 
DataOutput
            public abstract void write(byte[] b, int off, int len) throws 
IOException;  // From DataOutput
            public abstract void writeObject(Object obj) throws IOException;
}
```

Implemented By:

ObjectOutputStream

Passed To:

Externalizable.writeExternal()

24.46 java.io.OutputStream (JDK 1.0)

This abstract class is the superclass of all output streams. It defines the basic output methods that all output stream classes provide.

`write()` writes a single byte or an array or subarray of bytes. `flush()` forces any buffered output to be written. `close()` closes the stream and frees up any system resources associated with it. The stream may not be used once `close()` has been called.

See also `Writer`.

```
public abstract class OutputStream extends Object {
    // Default Constructor: public OutputStream()
    // Public Instance Methods
            public void close() throws IOException;
            public void flush() throws IOException;
            public abstract void write(int b) throws IOException;
            public void write(byte[] b) throws IOException;
            public void write(byte[] b, int off, int len) throws IOException;
}
```

Extended By:

ByteArrayOutputStream, FileOutputStream, FilterOutputStream, ObjectOutputStream, PipedOutputStream

Passed To:

BufferedOutputStream(), ByteArrayOutputStream.writeTo(), CheckedOutputStream(), DataOutputStream(), DeflaterOutputStream(), FilterOutputStream(), GZIPOutputStream(), ObjectOutputStream(), OutputStreamWriter(), PrintStream(), PrintWriter(), Properties.save(), Runtime.getLocalizedOutputStream(), ZipOutputStream()

Returned By:

Process.getOutputStream(), Runtime.getLocalizedOutputStream(), Socket.getOutputStream(), SocketImpl.getOutputStream(), URLConnection.getOutputStream()

Type Of:

FilterOutputStream.out

PREVIOUS

NEXT

24.47 java.io.OutputStreamWriter (JDK 1.1)

This class is a character output stream that uses a byte output stream as the destination for its data: when characters are written to an `OutputStreamWriter`, it translates them into bytes according to a particular locale- and/or platform-specific character encoding, and writes those bytes to the specified `OutputStream`. This is a very important internationalization feature in Java 1.1.

When you create an `OutputStreamWriter`, you specify the `OutputStream` to which it writes bytes, and you optionally specify the name of the character encoding that should be used to convert characters to bytes. If you do not specify an encoding name, the `OutputStreamWriter` uses the default encoding of the default locale, which is usually the correct thing to do.

`OutputStreamWriter` supports the usual `Writer` methods. It also has a `getEncoding()` method which returns the name of the encoding being used to convert characters to bytes.

```
public class OutputStreamWriter extends Writer {
    // Public Constructors
            public OutputStreamWriter(OutputStream out, String enc) throws 
UnsupportedEncodingException;
            public OutputStreamWriter(OutputStream out);
    // Public Instance Methods
            public void close() throws IOException;  // Defines Writer
            public void flush() throws IOException;  // Defines Writer
            public String getEncoding();
            public void write(int c) throws IOException;  // Overrides Writer
            public void write(char[] cbuf, int off, int len) throws IOException;  // 
Defines Writer
            public void write(String str, int off, int len) throws IOException;  // 
Overrides Writer
}
```

Hierarchy:

Object->Writer->OutputStreamWriter

Extended By:

FileWriter

PREVIOUS HOME NEXT
java.io.OutputStream (JDK 1.0) BOOK INDEX java.io.PipedInputStream (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

24.49 java.io.PipedOutputStream (JDK 1.0)

This class is an `OutputStream` that implements one half of a pipe, and is useful for communication between threads. A `PipedOutputStream` must be connected to a `PipedInputStream`, which may be specified when the `PipedOutputStream` is created or with the `connect()` method. Data written to the `PipedOutputStream` are available for reading on the `PipedInputStream`.

See `OutputStream` for information on the low-level methods for writing data to a `PipedOutputStream`. A `FilterOutputStream` may be used to provide a higher-level interface for writing data to a `PipedOutputStream`.

```
public class PipedOutputStream extends OutputStream {
    // Public Constructors
            public PipedOutputStream(PipedInputStream snk) throws IOException;
            public PipedOutputStream();
    // Public Instance Methods
            public void close() throws IOException;  // Overrides OutputStream
            public void connect(PipedInputStream snk) throws IOException;
            public synchronized void flush() throws IOException;  // Overrides 
OutputStream
            public void write(int b) throws IOException;  // Defines OutputStream
            public void write(byte[] b, int off, int len) throws IOException;  // 
Overrides OutputStream
}
```

Hierarchy:

Object->OutputStream->PipedOutputStream

Passed To:

PipedInputStream(), PipedInputStream.connect()

24.50 java.io.PipedReader (JDK 1.1)

`PipedReader` is a character input stream that reads characters from a `PipedWriter` character output stream to which it is connected. `PipedReader` implements one-half of a pipe, and is useful for communication between two threads of an application.

A `PipedReader` cannot be used until it is connected to a `PipedWriter` object, which may be passed to the `PipedReader()` constructor or to the `connect()` method.

`PipedReader` inherits most of the methods of its superclass. See `Reader` for more information.

`PipedReader` is the character stream analog of `PipedInputStream`.

```
public class PipedReader extends Reader {
    // Public Constructors
            public PipedReader();
            public PipedReader(PipedWriter src) throws IOException;
    // Public Instance Methods
            public void close() throws IOException;  // Defines Reader
            public void connect(PipedWriter src) throws IOException;
            public int read(char[] cbuf, int off, int len) throws IOException;  //
Defines Reader
}
```

Hierarchy:

Object->Reader->PipedReader

Passed To:

PipedWriter(), PipedWriter.connect()

24.51 java.io.PipedWriter (JDK 1.1)

`PipedWriter` is a character output stream that writes characters to the `PipedReader` character input stream to which it is connected. `PipedWriter` implements one half of a pipe, and is useful for communication between two threads of an application.

A `PipedWriter` cannot be used until it is connected to a `PipedReader` object, which may be passed to the `PipedWriter()` constructor, or to the `connect()` method.

`PipedWriter` inherits most of the methods of its superclass. See `Writer` for more information.

`PipedWriter` is the character stream analog of `PipedOutputStream`.

```
public class PipedWriter extends Writer {
    // Public Constructors
            public PipedWriter();
            public PipedWriter(PipedReader sink) throws IOException;
    // Public Instance Methods
            public void close() throws IOException;  // Defines Writer
            public void connect(PipedReader sink) throws IOException;
            public void flush() throws IOException;  // Defines Writer
            public void write(char[] cbuf, int off, int len) throws IOException;  //
Defines Writer
}
```

Hierarchy:

Object->Writer->PipedWriter

Passed To:

PipedReader(), PipedReader.connect()

24.55 java.io.PushbackReader (JDK 1.1)

This class is a character input stream that uses another input stream as its input source, and adds the ability to push characters back onto the stream. This feature is often useful when writing parsers.

When you create a `PushbackReader` stream, you specify the stream to be read from, and may optionally specify the size of the pushback buffer--i.e., the number of characters that may be pushed back onto the stream or "unread." If you do not specify a size for this buffer, the default size is one character.

`PushbackReader` inherits or overrides all the standard `Reader` methods, and also adds three `unread()` methods that are used to push a single character, an array of characters, or a portion of an array of characters back onto the stream.

This class is the character stream analog of `PushbackInputStream`.

```
public class PushbackReader extends FilterReader {
    // Public Constructors
            public PushbackReader(Reader in, int size);
            public PushbackReader(Reader in);
    // Public Instance Methods
            public void close() throws IOException;  // Overrides FilterReader
            public boolean markSupported();  // Overrides FilterReader
            public int read() throws IOException;  // Overrides FilterReader
            public int read(char[] cbuf, int off, int len) throws IOException;  //
Overrides FilterReader
            public boolean ready() throws IOException;  // Overrides FilterReader
            public void unread(int c) throws IOException;
            public void unread(char[] cbuf, int off, int len) throws IOException;
            public void unread(char[] cbuf) throws IOException;
}
```

Hierarchy:

Object->Reader->FilterReader->PushbackReader

24.56 java.io.RandomAccessFile (JDK 1.0)

This class allows reading and writing of arbitrary bytes, text, and primitive Java data types from or to any specified location in a file. Because this class provides random, rather than sequential, access to files, it is neither a subclass of `InputStream` nor of `OutputStream`, but provides an entirely independent method for reading and writing data from or to files. `RandomAccessFile` implements the same interfaces as `DataInputStream` and `DataOutputStream`, and thus defines the same methods for reading and writing data as those classes do.

The `seek()` method provides random access to the file--it is used to select the position in the file from which, or to which, data should be read or written. The *`mode`* argument to the constructor methods should be "r" for a file that is to be read-only, and "rw" for a file that is to be written (and perhaps read as well).

```
public class RandomAccessFile extends Object implements DataOutput, DataInput {
    // Public Constructors
            public RandomAccessFile(String name, String mode) throws IOException;
            public RandomAccessFile(File file, String mode) throws IOException;
    // Public Instance Methods
            public native void close() throws IOException;
            public final FileDescriptor getFD() throws IOException;
            public native long getFilePointer() throws IOException;
            public native long length() throws IOException;
            public native int read() throws IOException;
            public int read(byte[] b, int off, int len) throws IOException;
            public int read(byte[] b) throws IOException;
            public final boolean readBoolean() throws IOException;  // From DataInput
            public final byte readByte() throws IOException;  // From DataInput
            public final char readChar() throws IOException;  // From DataInput
            public final double readDouble() throws IOException;  // From DataInput
            public final float readFloat() throws IOException;  // From DataInput
            public final void readFully(byte[] b) throws IOException;  // From
DataInput
            public final void readFully(byte[] b, int off, int len) throws
IOException;  // From DataInput
            public final int readInt() throws IOException;  // From DataInput
            public final String readLine() throws IOException;  // From DataInput
            public final long readLong() throws IOException;  // From DataInput
            public final short readShort() throws IOException;  // From DataInput
            public final String readUTF() throws IOException;  // From DataInput
            public final int readUnsignedByte() throws IOException;  // From
DataInput
            public final int readUnsignedShort() throws IOException;  // From
DataInput
            public native void seek(long pos) throws IOException;
            public int skipBytes(int n) throws IOException;  // From DataInput
            public native void write(int b) throws IOException;  // From DataOutput
            public void write(byte[] b) throws IOException;  // From DataOutput
            public void write(byte[] b, int off, int len) throws IOException;  // 
From DataOutput
            public final void writeBoolean(boolean v) throws IOException;  // From 
DataOutput
            public final void writeByte(int v) throws IOException;  // From 
DataOutput
            public final void writeBytes(String s) throws IOException;  // From 
DataOutput
            public final void writeChar(int v) throws IOException;  // From 
DataOutput
            public final void writeChars(String s) throws IOException;  // From 
DataOutput
            public final void writeDouble(double v) throws IOException;  // From 
DataOutput
            public final void writeFloat(float v) throws IOException;  // From 
DataOutput
            public final void writeInt(int v) throws IOException;  // From DataOutput
            public final void writeLong(long v) throws IOException;  // From 
DataOutput
            public final void writeShort(int v) throws IOException;  // From 
DataOutput
            public final void writeUTF(String str) throws IOException;  // From 
DataOutput
}
```

PREVIOUS | HOME | NEXT

java.io.PushbackReader (JDK 1.1) | BOOK INDEX | java.io.Reader (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

24.57 java.io.Reader (JDK 1.1)

This abstract class is the superclass of all character input streams. It is an analog to `InputStream`, which is the superclass of all byte input streams. `Reader` defines the basic methods that all character output streams provide.

`read()` returns a single character or an array (or subarray) of characters, blocking if necessary. It returns `-1` if the end of the stream has been reached. `ready()` returns `true` if there are characters available for reading. If `ready()` returns `true`, the next call to `read()` is guaranteed not to block. `close()` closes the character input stream. `skip()` skips a specified number of characters in the input stream. If `markSupported()` returns `true`, then `mark()` "marks" a position in the stream and, if necessary, creates a look-ahead buffer of the specified size. Future calls to `reset()` restore the stream to the marked position, if they occur within the specified look-ahead limit. Note that not all stream types support this mark-and-reset functionality.

To create a subclass of `Reader`, you need only implement the three-argument version of `read()` and `close()`. Most subclasses implement additional methods as well, however.

```
public abstract class Reader extends Object {
    // Protected Constructors
            protected Reader();
            protected Reader(Object lock);
    // Protected Instance Variables
            protected Object lock;
    // Public Instance Methods
            public abstract void close() throws IOException;
            public void mark(int readAheadLimit) throws IOException;
            public boolean markSupported();
            public int read() throws IOException;
            public int read(char[] cbuf) throws IOException;
            public abstract int read(char[] cbuf, int off, int len) throws 
IOException;
            public boolean ready() throws IOException;
            public void reset() throws IOException;
            public long skip(long n) throws IOException;
}
```

Extended By:

BufferedReader, CharArrayReader, FilterReader, InputStreamReader, PipedReader, StringReader

Passed To:

BufferedReader(), FilterReader(), LineNumberReader(), PushbackReader(), StreamTokenizer()

Type Of:

FilterReader.in

28.16 java.net.Socket (JDK 1.0)

This class implements a socket for interprocess communication over the network. The constructor methods create the socket and connect it to the specified host and port. You may also optionally specify whether communication through the socket should be based on an underlying reliable connection-based stream protocol, or on an underlying unreliable (but faster) datagram protocol. A stream protocol is the default.

Once the socket is created, `getInputStream()` and `getOutputStream()` return `InputStream` and `OutputStream` objects that you can use just as you would for file input and output. `getInetAddress()` and `getPort()` return the address and port that the socket is connected to. `getLocalPort()` returns the local port that the socket is using.

```
public class Socket extends Object {
    // Public Constructors
            public Socket(String host, int port) throws UnknownHostException, 
IOException;
            public Socket(InetAddress address, int port) throws IOException;
        1.1public Socket(String host, int port, InetAddress localAddr, int localPort) 
throws IOException;
        1.1public Socket(InetAddress address, int port, InetAddress localAddr, int 
localPort) throws IOException;
        #   public Socket(String host, int port, boolean stream) throws IOException;
        #   public Socket(InetAddress host, int port, boolean stream) throws 
IOException;
    // Protected Constructors
        1.1protected Socket();
        1.1protected Socket(SocketImpl impl) throws SocketException;
    // Class Methods
            public static synchronized void setSocketImplFactory(SocketImplFactory 
fac) throws IOException;
    // Public Instance Methods
            public synchronized void close() throws IOException;
            public InetAddress getInetAddress();
            public InputStream getInputStream() throws IOException;
        1.1public InetAddress getLocalAddress();
            public int getLocalPort();
            public OutputStream getOutputStream() throws IOException;
            public int getPort();
        1.1public int getSoLinger() throws SocketException;
        1.1public synchronized int getSoTimeout() throws SocketException;
        1.1public boolean getTcpNoDelay() throws SocketException;
        1.1public void setSoLinger(boolean on, int val) throws SocketException;
        1.1public synchronized void setSoTimeout(int timeout) throws SocketException;
        1.1public void setTcpNoDelay(boolean on) throws SocketException;
            public String toString();  // Overrides Object
}
```

Passed To:

ServerSocket.implAccept()

Returned By:

ServerSocket.accept()

PREVIOUS | HOME | NEXT

java.net.ServerSocket (JDK 1.0) | BOOK INDEX | java.net.SocketException (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

24.63 java.io.StringReader (JDK 1.1)

This class is a character input stream that uses a `String` object as the source of the characters it returns. When you create a `StringReader`, you must specify the `String` that it is to read from.

`StringReader` defines the normal `Reader` methods, and supports `mark()` and `reset()`. If `reset()` is called before `mark()` has been called, the stream is reset to the beginning of the specified string.

`StringReader` is a character stream analog to `StringBufferInputStream`, which is deprecated in Java 1.1. `StringReader` is also similar to `CharArrayReader`.

```
public class StringReader extends Reader {
    // Public Constructor
            public StringReader(String s);
    // Public Instance Methods
            public void close();  // Defines Reader
            public void mark(int readAheadLimit) throws IOException;  // Overrides 
Reader
            public boolean markSupported();  // Overrides Reader
            public int read() throws IOException;  // Overrides Reader
            public int read(char[] cbuf, int off, int len) throws IOException;  // 
Defines Reader
            public boolean ready();  // Overrides Reader
            public void reset() throws IOException;  // Overrides Reader
            public long skip(long ns) throws IOException;  // Overrides Reader
}
```

Hierarchy:

Object->Reader->StringReader

24.64 java.io.StringWriter (JDK 1.1)

This class is a character output stream that uses an internal `StringBuffer` object as the destination of the characters written to the stream. When you create a `StringWriter`, you may optionally specify an initial size for the `StringBuffer`, but you do not specify the `StringBuffer` itself--it is managed internally by the `StringWriter`, and grows as necessary to accommodate the characters written to it.

`StringWriter` defines the standard `write()`, `flush()`, and `close()` methods that all `Writer` subclasses do, and also defines two methods to obtain the characters that have been written into the stream's internal buffer. `toString()` returns the contents of the internal buffer as a `String`, and `getBuffer()` returns the buffer itself. Note that `getBuffer()` returns a reference to the actual internal buffer, not a copy of it, so any changes you make to the buffer are reflected in subsequent calls to `toString()`.

`StringWriter` is quite similar to `CharArrayWriter`, but does not have a byte stream analog.

```
public class StringWriter extends Writer {
    // Public Constructor
            public StringWriter();
    // Protected Constructor
            protected StringWriter(int initialSize);
    // Public Instance Methods
            public void close();  // Defines Writer
            public void flush();  // Defines Writer
            public StringBuffer getBuffer();
            public String toString();  // Overrides Object
            public void write(int c);  // Overrides Writer
            public void write(char[] cbuf, int off, int len);  // Defines Writer
            public void write(String str);  // Overrides Writer
            public void write(String str, int off, int len);  // Overrides Writer
}
```

Hierarchy:

Object->Writer->StringWriter

java.io.StringReader (JDK 1.1) BOOK INDEX java.io.SyncFailedException (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

24.69 java.io.Writer (JDK 1.1)

This abstract class is the superclass of all character output streams. It is an analog to `OutputStream`, which is the superclass of all byte output streams. `Writer` defines the basic `write()`, `flush()`, and `close()` methods that all character output streams provide.

The five versions of the `write()` method write a single character, a character array or subarray, or a string or substring to the destination of the stream. The most general version of this method--the one that writes a specified portion of a character array--is abstract and must be implemented by all subclasses. By default, the other `write()` methods are implemented in terms of this abstract one.

The `flush()` method is another abstract method that all subclasses must implement. It should force any output buffered by the stream to be written to its destination. If that destination is itself a character or byte output stream, it should invoke the `flush()` method of the destination stream as well.

The `close()` method is also abstract. Subclasses must implement this method so that it flushes and then closes the current stream, and also closes whatever destination stream it is connected to. Once the stream has been closed, any future calls to `write()` or `flush()` should throw an `IOException`.

```
public abstract class Writer extends Object {
    // Protected Constructors
            protected Writer();
            protected Writer(Object lock);
    // Protected Instance Variables
            protected Object lock;
    // Public Instance Methods
            public abstract void close() throws IOException;
            public abstract void flush() throws IOException;
            public void write(int c) throws IOException;
            public void write(char[] cbuf) throws IOException;
            public abstract void write(char[] cbuf, int off, int len) throws
IOException;
            public void write(String str) throws IOException;
            public void write(String str, int off, int len) throws IOException;
}
```

Extended By:

BufferedWriter, CharArrayWriter, FilterWriter, OutputStreamWriter, PipedWriter, PrintWriter, StringWriter

Passed To:

BufferedWriter(), CharArrayWriter.writeTo(), FilterWriter(), PrintWriter()

Type Of:

FilterWriter.out

PREVIOUS	HOME	NEXT
java.io.WriteAbortedException (JDK 1.1)	BOOK INDEX	The java.lang Package

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

31.15 java.util.zip.ZipFile (JDK 1.1)

This class reads the contents of ZIP files. It uses a random access file internally so that the entries of the ZIP file do not have to be read sequentially as they do with the `ZipInputStream` class.

A `ZipFile` object can be created by specifying the ZIP file to be read either as a `String` filename or as a `File` object. Once created, the `getEntry()` method returns a `ZipEntry` object for a named entry, and the `entries()` method returns an `Enumeration` object that allows you to loop through all the `ZipEntry` objects for the file. To read the contents of a specific `ZipEntry` within the ZIP file, pass the `ZipEntry` to `getInputStream()`--this returns an `InputStream` object from which you can read the entry's contents.

```
public class ZipFile extends Object {
    // Public Constructors
            public ZipFile(String name) throws IOException;
            public ZipFile(File file) throws ZipException, IOException;
    // Public Instance Methods
            public void close() throws IOException;
            public Enumeration entries();
            public ZipEntry getEntry(String name);
            public InputStream getInputStream(ZipEntry ze) throws IOException;
            public String getName();
}
```

31.16 java.util.zip.ZipInputStream (JDK 1.1)

This class is a subclass of `InflaterInputStream` that reads the entries of a ZIP file in sequential order. A `ZipInputStream` is created by specifying the `InputStream` from which it is to read the contents of the ZIP file. Once the `ZipInputStream` is created, the `getNextEntry()` method is used to begin reading of data from the next entry in the ZIP file. This method must be called before `read()` is called to begin reading of the first entry. Note that `getNextEntry()` returns a `ZipEntry` object that describes the entry being read. Also note that `getNextEntry()` returns `null` when there are no more entries to be read from the ZIP file.

The `read()` methods of `ZipInputStream` read until the end of the current entry and then return `-1` indicating that there are no more data to read. To continue with the next entry in the ZIP file, you must call `getNextEntry()` again. Similarly, the `skip()` method only skips bytes within the current entry. `closeEntry()` can be called to skip the remaining data in the current entry, but it is usually easier simply to call `getNextEntry()` to begin the next entry.

```
public class ZipInputStream extends InflaterInputStream {
    // Public Constructor
            public ZipInputStream(InputStream in);
    // Public Instance Methods
            public void close() throws IOException;  // Overrides FilterInputStream
            public void closeEntry() throws IOException;
            public ZipEntry getNextEntry() throws IOException;
            public int read(byte[] b, int off, int len) throws IOException;  //
Overrides InflaterInputStream
            public long skip(long n) throws IOException;  // Overrides
InflaterInputStream
}
```

Hierarchy:

Object->InputStream->FilterInputStream->InflaterInputStream->ZipInputStream

25.16 java.lang.Compiler (JDK 1.0)

The static methods of this class provide an interface to the just-in-time (JIT) byte-code to native code compiler in use by the Java interpreter. If no JIT compiler is in use by the VM, these methods do nothing.

`compileClass()` asks the JIT compiler to compile the specified class. `compileClasses()` asks the JIT compiler to compile all classes that match the specified name. These methods return `true` if the compilation was successful, or `false` if it failed or if there is no JIT compiler on the system. `enable()` and `disable()` turn just-in-time compilation on and off. `command()` asks the JIT compiler to perform some compiler-specific operation. This is a hook for vendor extensions. No standard operations have been defined.

```
public final class Compiler extends Object {
    // No Constructor
    // Class Methods
            public static native Object command(Object any);
            public static native boolean compileClass(Class clazz);
            public static native boolean compileClasses(String string);
            public static native void disable();
            public static native void enable();
}
```

24.61 java.io.StreamTokenizer (JDK 1.0)

This class performs lexical analysis of a specified input stream and breaks the input up into tokens. It can be extremely useful when writing simple parsers.

`nextToken()` returns the next token in the stream--this is either one of the constants defined by the class (which represent end-of-file, end-of-line, a parsed floating-point number, and a parsed word) or a character value. `pushBack()` "pushes" the token back onto the stream, so that it is returned by the next call to `nextToken()`. The public variables `sval` and `nval` contain the string and numeric values (if applicable) of the most recently read token. They are applicable when the returned token is `TT_WORD` and `TT_NUMBER`. `lineno()` returns the current line number.

The remaining methods allow you to specify how tokens are recognized. `wordChars()` specifies a range of characters that should be treated as parts of words. `whitespaceChars()` specifies a range of characters that serve to delimit tokens. `ordinaryChars()` and `ordinaryChar()` specify characters that are never part of tokens and should be returned as-is. `resetSyntax()` makes all characters "ordinary." `eolIsSignificant()` specifies whether end-of-line is significant. If so, the `TT_EOL` constant is returned for end-of-lines. Otherwise they are treated as whitespace.

`commentChar()` specifies a character that begins a comment that lasts until the end of the line. No characters in the comment are returned. `slashStarComments()` and `slashSlashComments()` specify whether the `StreamTokenizer` should recognize C and C++-style comments. If so, no parts of the comments are returned as tokens. `quoteChar()` specifies a character used to delimit strings. When a string token is parsed, the quote character is returned as the token value, and the body of the string is stored in the `sval` variable. `lowerCaseMode()` specifies whether `TT_WORD` tokens should be converted to all lowercase characters before being stored in `sval`. `parseNumbers()` specifies that the `StreamTokenizer` should recognize and return double-precision floating-point number tokens.

```
public class StreamTokenizer extends Object {
    // Public Constructors
        #   public StreamTokenizer(InputStream is);
        1.1  public StreamTokenizer(Reader r);
    // Constants
            public static final int TT_EOF;
            public static final int TT_EOL;
            public static final int TT_NUMBER;
            public static final int TT_WORD;
    // Public Instance Variables
            public double nval;
            public String sval;
            public int ttype;
    // Public Instance Methods
            public void commentChar(int ch);
            public void eolIsSignificant(boolean flag);
            public int lineno();
            public void lowerCaseMode(boolean fl);
            public int nextToken() throws IOException;
            public void ordinaryChar(int ch);
            public void ordinaryChars(int low, int hi);
            public void parseNumbers();
            public void pushBack();
            public void quoteChar(int ch);
            public void resetSyntax();
            public void slashSlashComments(boolean flag);
            public void slashStarComments(boolean flag);
            public String toString();  // Overrides Object
            public void whitespaceChars(int low, int hi);
            public void wordChars(int low, int hi);
}
```

PREVIOUS | HOME | NEXT

java.io.StreamCorruptedException (JDK 1.1) | BOOK INDEX | java.io.StringBufferInputStream (JDK 1.0; Deprecated.)

29.5 java.text.CollationKey (JDK 1.1)

`CollationKey` objects are used to compare strings more quickly than is possible with `Collation.compare()`. Objects of this class are returned by `Collation.getCollationKey()`. To compare two `CollationKey` objects, invoke the `compareTo()` method of key `A`, passing the key `B` as an argument (both `CollationKey` objects must be created through the same `Collation` object). The return value of this method is less than zero if the key `A` is collated before the key `B`. It is equal to zero if they are equivalent for the purposes of collation, and it is greater than zero if the key `A` is collated after the key `B`.

Use `getSourceString()` to obtain the string represented by a `CollationKey`.

```
public final class CollationKey extends Object {
    // No Constructor
    // Public Instance Methods
            public int compareTo(CollationKey target);
            public boolean equals(Object target);  // Overrides Object
            public String getSourceString();
            public int hashCode();  // Overrides Object
            public byte[] toByteArray();
}
```

Passed To:

CollationKey.compareTo()

Returned By:

Collator.getCollationKey(), RuleBasedCollator.getCollationKey()

PREVIOUS | HOME | NEXT
java.text.CollationElementIterator (JDK 1.1) | BOOK INDEX | java.text.Collator (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

21.6 java.awt.image.ImageConsumer (JDK 1.0)

This interface defines the methods necessary for a class that consumes image data to communicate with a class that produces image data. The methods defined by this interface should never be called by a program directly; instead, they are invoked by an `ImageProducer` to pass the image data and other information about the image to the `ImageConsumer`. The constants defined by this interface are values passed to the `setHints()` and `imageComplete()` methods.

Unless you want to do low-level manipulation of image data, you never need to use or implement an `ImageConsumer`.

```
public abstract interface ImageConsumer {
    // Constants
            public static final int COMPLETESCANLINES;
            public static final int IMAGEABORTED;
            public static final int IMAGEERROR;
            public static final int RANDOMPIXELORDER;
            public static final int SINGLEFRAME;
            public static final int SINGLEFRAMEDONE;
            public static final int SINGLEPASS;
            public static final int STATICIMAGEDONE;
            public static final int TOPDOWNLEFTRIGHT;
    // Public Instance Methods
            public abstract void imageComplete(int status);
            public abstract void setColorModel(ColorModel model);
            public abstract void setDimensions(int width, int height);
            public abstract void setHints(int hintflags);
            public abstract void setPixels(int x, int y, int w, int h, ColorModel
model, byte[] pixels, int off, int scansize);
            public abstract void setPixels(int x, int y, int w, int h, ColorModel
model, int[] pixels, int off, int scansize);
            public abstract void setProperties(Hashtable props);
}
```

Implemented By:

`ImageFilter, PixelGrabber`

Passed To:

`FilteredImageSource.addConsumer(), FilteredImageSource.isConsumer(), FilteredImageSource.removeConsumer(), FilteredImageSource.requestTopDownLeftRightResend(), FilteredImageSource.startProduction(), ImageFilter.getFilterInstance(), ImageProducer.addConsumer(), ImageProducer.isConsumer(), ImageProducer.removeConsumer(), ImageProducer.requestTopDownLeftRightResend(), ImageProducer.startProduction(), MemoryImageSource.addConsumer(), MemoryImageSource.isConsumer(), MemoryImageSource.removeConsumer(), MemoryImageSource.requestTopDownLeftRightResend(), MemoryImageSource.startProduction()`

Type Of:

`ImageFilter.consumer`

20.9 java.awt.event.ContainerEvent (JDK 1.1)

An event of this type serves as notification that the source `Container` has had a child added to it or removed from it. Note that this event is a notification only; the AWT adds or removes the child internally, and the recipient of this event need take no action itself.

`getChild()` returns the child `Component` that was added or removed, and `getContainer()` returns the `Container` that it was added to or removed from. `getContainer()` is simply a convenient alternative to `getSource()`.

`getID()` returns the constant `COMPONENT_ADDED` or `COMPONENT_REMOVED` to indicate whether the specified child was added or removed.

```
public class ContainerEvent extends ComponentEvent {
    // Public Constructor
            public ContainerEvent(Component source, int id, Component child);
    // Constants
            public static final int COMPONENT_ADDED;
            public static final int COMPONENT_REMOVED;
            public static final int CONTAINER_FIRST;
            public static final int CONTAINER_LAST;
    // Public Instance Methods
            public Component getChild();
            public Container getContainer();
            public String paramString();  // Overrides ComponentEvent
}
```

Hierarchy:

`Object->EventObject(Serializable)->AWTEvent->ComponentEvent->ContainerEvent`

Passed To:

`AWTEventMulticaster.componentAdded(), AWTEventMulticaster.componentRemoved(), Container.processContainerEvent(), ContainerAdapter.componentAdded(),` `ContainerAdapter.componentRemoved(), ContainerListener.componentAdded(), ContainerListener.componentRemoved()`

20.6 java.awt.event.ComponentEvent (JDK 1.1)

An event of this type serves as notification that the source `Component` has been moved, resized, shown, or hidden. Note that this event is a notification only: the AWT handles these `Component` operations internally, and the recipient of the event need take no action itself.

`getComponent()` returns the component that was moved, resized, shown or hidden. It is simply a convenient alternative to `getSource()`.

`getID()` returns one of four `COMPONENT_` constants to indicate what operation was performed on the `Component`.

```
public class ComponentEvent extends AWTEvent {
    // Public Constructor
            public ComponentEvent(Component source, int id);
    // Constants
            public static final int COMPONENT_FIRST;
            public static final int COMPONENT_HIDDEN;
            public static final int COMPONENT_LAST;
            public static final int COMPONENT_MOVED;
            public static final int COMPONENT_RESIZED;
            public static final int COMPONENT_SHOWN;
    // Public Instance Methods
            public Component getComponent();
            public String paramString();  // Overrides AWTEvent
}
```

Hierarchy:

```
Object->EventObject(Serializable)->AWTEvent->ComponentEvent
```

Extended By:

ContainerEvent, FocusEvent, InputEvent, PaintEvent, WindowEvent

Passed To:

AWTEventMulticaster.componentHidden(), AWTEventMulticaster.componentMoved(), AWTEventMulticaster.componentResized(), AWTEventMulticaster.componentShown(), Component.processComponentEvent(), ComponentAdapter.componentHidden(), ComponentAdapter.componentMoved(), ComponentAdapter.componentResized(), ComponentAdapter.componentShown(), ComponentListener.componentHidden(), ComponentListener.componentMoved(), ComponentListener.componentResized(), ComponentListener.componentShown()

20.8 java.awt.event.ContainerAdapter (JDK 1.1)

This class is a trivial implementation of `ContainerListener`; it provides empty bodies for each of the methods of that interface. When you are not interested in all of these methods, it is often easier to subclass `ContainerAdapter` than it is to implement `ContainerListener` from scratch.

```
public abstract class ContainerAdapter extends Object implements ContainerListener {
    // Default Constructor: public ContainerAdapter()
    // Public Instance Methods
            public void componentAdded(ContainerEvent e);  // From ContainerListener
            public void componentRemoved(ContainerEvent e);  // From 
ContainerListener
}
```

Hierarchy:

```
Object->ContainerAdapter(ContainerListener(EventListener))
```

20.10 java.awt.event.ContainerListener (JDK 1.1)

This interface defines the methods that an object must implement to "listen" for container events on AWT components. When a `ContainerEvent` occurs, an AWT component notifies its registered `ContainerListener` objects by invoking one of their methods.

An easy way to implement this interface is by subclassing the `ContainerAdapter` class.

```
public abstract interface ContainerListener extends EventListener {
    // Public Instance Methods
            public abstract void componentAdded(ContainerEvent e);
            public abstract void componentRemoved(ContainerEvent e);
}
```

Implemented By:

AWTEventMulticaster, ContainerAdapter

Passed To:

AWTEventMulticaster.add(), AWTEventMulticaster.remove(), Container.addContainerListener(), Container.removeContainerListener()

Returned By:

AWTEventMulticaster.add(), AWTEventMulticaster.remove()

java.awt.event.ContainerEvent (JDK 1.1) BOOK INDEX java.awt.event.FocusAdapter (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

20.5 java.awt.event.ComponentAdapter (JDK 1.1)

This class is a trivial implementation of `ComponentListener`; it provides empty bodies for each of the methods of that interface. When you are not interested in all of these methods, it is often easier to subclass `ComponentAdapter` than it is to implement `ComponentListener` from scratch.

```
public abstract class ComponentAdapter extends Object implements ComponentListener {
    // Default Constructor: public ComponentAdapter()
    // Public Instance Methods
            public void componentHidden(ComponentEvent e);  // From ComponentListener
            public void componentMoved(ComponentEvent e);  // From ComponentListener
            public void componentResized(ComponentEvent e);  // From
ComponentListener
            public void componentShown(ComponentEvent e);  // From ComponentListener
}
```

Hierarchy:

```
Object->ComponentAdapter(ComponentListener(EventListener))
```

20.7 java.awt.event.ComponentListener (JDK 1.1)

This interface defines the methods that an object must implement to "listen" for component events on AWT components. When a `ComponentEvent` occurs, an AWT component notifies its registered `ComponentListener` objects by invoking one of their methods.

An easy way to implement this interface is by subclassing the `ComponentAdapter` class.

```
public abstract interface ComponentListener extends EventListener {
    // Public Instance Methods
            public abstract void componentHidden(ComponentEvent e);
            public abstract void componentMoved(ComponentEvent e);
            public abstract void componentResized(ComponentEvent e);
            public abstract void componentShown(ComponentEvent e);
}
```

Implemented By:

`AWTEventMulticaster, ComponentAdapter`

Passed To:

`AWTEventMulticaster.add(), AWTEventMulticaster.remove(), Component.addComponentListener(), Component.removeComponentListener()`

Returned By:

`AWTEventMulticaster.add(), AWTEventMulticaster.remove()`

PREVIOUS HOME NEXT
java.awt.event.ComponentEvent (JDK 1.1) BOOK INDEX java.awt.event.ContainerAdapter (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

30.23 java.util.StringTokenizer (JDK 1.0)

This class, when instantiated with a String, breaks the string up into tokens separated by any of the characters in the specified string of delimiters. (For example, words separated by space and tab characters are tokens.) The hasMoreTokens() and nextToken() methods can be used to obtain the tokens in order. countTokens() returns the number of tokens in the string. Note that StringTokenizer implements the Enumeration interface, so you may also access the tokens with the familiar hasMoreElements() and nextElement() methods.

When you create a StringTokenizer you may specify a string of delimiter characters to use for the entire string, or you may rely on the default whitespace delimiters. You may also specify whether the delimiters themselves should be returned as tokens. You may optionally specify a new string of delimiter characters when you call nextToken().

```
public class StringTokenizer extends Object implements Enumeration {
    // Public Constructors
            public StringTokenizer(String str, String delim, boolean returnTokens);
            public StringTokenizer(String str, String delim);
            public StringTokenizer(String str);
    // Public Instance Methods
            public int countTokens();
            public boolean hasMoreElements();  // From Enumeration
            public boolean hasMoreTokens();
            public Object nextElement();  // From Enumeration
            public String nextToken();
            public String nextToken(String delim);
}
```

28.4 java.net.ContentHandlerFactory (JDK 1.0)

This interface defines a method that creates and returns an appropriate `ContentHandler` object for a specified MIME type. A system-wide `ContentHandlerFactory` interface may be specified by using the `URLConnection.setContentHandlerFactory()` method.

Normal applications never need to use or implement this interface.

```
public abstract interface ContentHandlerFactory {
    // Public Instance Methods
            public abstract ContentHandler createContentHandler(String mimetype);
}
```

Passed To:

URLConnection.setContentHandlerFactory()

28.19 java.net.SocketImplFactory (JDK 1.0)

This interface defines a method that creates `SocketImpl` objects. `SocketImplFactory` objects may be registered to create `SocketImpl` objects for the `Socket` and `ServerSocket` classes.

Normal applications never need to use or implement this interface.

```
public abstract interface SocketImplFactory {
    // Public Instance Methods
            public abstract SocketImpl createSocketImpl();
}
```

Passed To:

ServerSocket.setSocketFactory(), Socket.setSocketImplFactory()

18.17 java.awt.Cursor (JDK 1.1)

This class represents a mouse cursor. It defines a number of constants, each of which specify one of the supported cursor images. Pass one of these constants to the constructor to create a cursor of the specified type, and call `getType()` to determine the type of an existing `Cursor` object.

Since there are only a fixed number of available cursors, the static method `getPredefinedCursor()` is more efficient than the `Cursor()` constructor--it maintains a cache of `Cursor` objects that can be reused. The static `getDefaultCursor()` method returns the system default cursor.

```
public class Cursor extends Object implements Serializable {
    // Public Constructor
            public Cursor(int type);
    // Cursor Constants
            public static final int DEFAULT_CURSOR;
            public static final int CROSSHAIR_CURSOR, HAND_CURSOR, MOVE_CURSOR;
            public static final int TEXT_CURSOR, WAIT_CURSOR;
            public static final int N_RESIZE_CURSOR, S_RESIZE_CURSOR;
            public static final int E_RESIZE_CURSOR, W_RESIZE_CURSOR;
            public static final int NE_RESIZE_CURSOR, NW_RESIZE_CURSOR;
            public static final int SE_RESIZE_CURSOR, SW_RESIZE_CURSOR;
    // Class Variables
            protected static Cursor[] predefined;
    // Class Methods
            public static Cursor getDefaultCursor();
            public static Cursor getPredefinedCursor(int type);
    // Public Instance Methods
            public int getType();
}
```

Passed To:

`Component.setCursor(), ComponentPeer.setCursor()`

Returned By:

`Component.getCursor(), Cursor.getDefaultCursor(), Cursor.getPredefinedCursor()`

Type Of:

`Cursor.predefined`

23.9 java.beans.Introspector (JDK 1.1)

The `Introspector` is a class that is never instantiated. Its static `getBeanInfo()` methods provide a way to obtain information about a Java bean, and are typically only invoked by application builders or similar tools. `getBeanInfo()` first looks for a `BeanInfo` class for the specified Java bean class. For a class named `x`, it looks for a `BeanInfo` class named `xBeanInfo`, first in the current package, and then in each of the packages in the `BeanInfo` search path.

If no `BeanInfo` class is found, or if the `BeanInfo` class found does not provide complete information about the bean properties, events, and methods, `getBeanInfo()` "introspects" on the bean class by using the `java.lang.reflect` package to fill in the missing information. When explicit information is provided by a `BeanInfo` class, `getBeanInfo()` treats it as definitive. When determining information through introspection, however, it examines each of the bean's superclasses in turn, looking for a `BeanInfo` class at that level or using introspection. When calling `getBeanInfo()`, you may optionally specify a second class argument that specifies a superclass for which, and above which, `getBeanInfo()` does not introspect.

```
public class Introspector extends Object {
    // No Constructor
    // Class Methods
            public static String decapitalize(String name);
            public static BeanInfo getBeanInfo(Class beanClass) throws 
IntrospectionException;
            public static BeanInfo getBeanInfo(Class beanClass, Class stopClass) 
throws IntrospectionException;
            public static String[] getBeanInfoSearchPath();
            public static void setBeanInfoSearchPath(String[] path);
}
```

26.5 java.lang.reflect.Member (JDK 1.1)

This interface defines the methods shared by all members (fields, methods, and constructors) of a class. `getName()` returns the name of the member, `getModifiers()` returns its modifiers, and `getDeclaringClass()` returns the `Class` object that represents the class of which the member is a part.

```
public abstract interface Member {
    // Constants
            public static final int DECLARED;
            public static final int PUBLIC;
    // Public Instance Methods
            public abstract Class getDeclaringClass();
            public abstract int getModifiers();
            public abstract String getName();
}
```

Implemented By:

Constructor, Field, Method

30.17 java.util.Properties (JDK 1.0)

This class is an extension of `Hashtable` that allows key/value pairs to be read from and written to a stream. The `Properties` class is used to implement the system properties list, which supports user customization by allowing programs to look up the value of named resources.

When you create a `Properties` object, you may specify another `Properties` object that contains default values. Keys (property names) and values are associated in a `Properties` object with the `Hashtable` method `put()`. Values are looked up with `getProperty()`--if this method does not find the key in the current `Properties` object, it looks in the default `Properties` object that was passed to the constructor method. A default value may also be specified in case the key is not found at all.

`propertyNames()` returns an enumeration of all property names (keys) stored in the `Properties` object and (recursively) also all property names stored in the default `Properties` object associated with it. `list()` prints the properties stored in a `Properties` object. It is useful for debugging. `save()` writes a `Properties` object to a stream. `load()` reads key/value pairs from a stream and stores them in a `Properties` object.

```
public class Properties extends Hashtable {
    // Public Constructors
            public Properties();
            public Properties(Properties defaults);
    // Protected Instance Variables
            protected Properties defaults;
    // Public Instance Methods
            public String getProperty(String key);
            public String getProperty(String key, String defaultValue);
            public void list(PrintStream out);
        1.1public void list(PrintWriter out);
            public synchronized void load(InputStream in) throws IOException;
            public Enumeration propertyNames();
            public synchronized void save(OutputStream out, String header);
}
```

Hierarchy:

Object->Dictionary->Hashtable(Cloneable, Serializable)->Properties

Passed To:

Properties(), System.setProperties(), Toolkit.getPrintJob()

Returned By:

System.getProperties()

Type Of:

Properties.defaults

25.12 java.lang.ClassLoader (JDK 1.0)

This abstract class defines the necessary hook for Java to load classes over the network, or from other sources. Normal applications do not need to use or subclass this class.

```
public abstract class ClassLoader extends Object {
    // Protected Constructor
            protected ClassLoader();
    // Class Methods
        1.1public static final URL getSystemResource(String name);
        1.1public static final InputStream getSystemResourceAsStream(String name);
    // Public Instance Methods
        1.1public URL getResource(String name);
        1.1public InputStream getResourceAsStream(String name);
        1.1public Class loadClass(String name) throws ClassNotFoundException;
    // Protected Instance Methods
        #   protected final Class defineClass(byte[] data, int offset, int length);
        1.1protected final Class defineClass(String name, byte[] data, int offset, 
int length);
        1.1protected final Class findLoadedClass(String name);
            protected final Class findSystemClass(String name) throws 
ClassNotFoundException;
            protected abstract Class loadClass(String name, boolean resolve) throws 
ClassNotFoundException;
            protected final void resolveClass(Class c);
        1.1protected final void setSigners(Class cl, Object[] signers);
}
```

Passed To:

Beans.instantiate()

Returned By:

Class.getClassLoader(), SecurityManager.currentClassLoader()

java.lang.ClassFormatError (JDK 1.0) BOOK INDEX java.lang.ClassNotFoundException (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

31.13 java.util.zip.ZipEntry (JDK 1.1)

This class describes a single entry (typically a compressed file) stored within a ZIP file. The various methods get and set various pieces of information about the entry. The `ZipEntry` class is used by the `ZipFile` and `ZipInputStream`, which read ZIP files, and by the `ZipOutputStream`, which writes ZIP files.

When reading a ZIP file, the `ZipEntry` objects returned by the `ZipFile` or `ZipInputStream` contain the name, size, modification time, and other information about each entry in the file. When writing a ZIP file, on the other hand, you must create your own `ZipEntry` objects, and initialize them to contain the entry name and other appropriate information before writing the contents of the entry.

```
public class ZipEntry extends Object {
    // Public Constructor
            public ZipEntry(String name);
    // Constants
            public static final int DEFLATED;
            public static final int STORED;
    // Public Instance Methods
            public String getComment();
            public long getCompressedSize();
            public long getCrc();
            public byte[] getExtra();
            public int getMethod();
            public String getName();
            public long getSize();
            public long getTime();
            public boolean isDirectory();
            public void setComment(String comment);
            public void setCrc(long crc);
            public void setExtra(byte[] extra);
            public void setMethod(int method);
            public void setSize(long size);
            public void setTime(long time);
            public String toString();  // Overrides Object
}
```

Passed To:

ZipFile.getInputStream(), ZipOutputStream.putNextEntry()

Returned By:

ZipFile.getEntry(), ZipInputStream.getNextEntry()

PREVIOUS | HOME | NEXT

java.util.zip.InflaterInputStream (JDK 1.1) | BOOK INDEX | java.util.zip.ZipException (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

20.15 java.awt.event.ItemEvent (JDK 1.1)

An event of this type indicates that an item within an `ItemSelectable` component has had its selection state changed.

`getItemSelectable()` is a convenient alternative to `getSource()` that returns the `ItemSelectable` object that originated the event. `getItem()` returns an object that represents the item that was selected or deselected.

`getID()` returns the type of the `ItemEvent`. The standard AWT components always generate item events of type `ITEM_STATE_CHANGED`. The `getStateChange()` method returns the new selection state of the item: it returns one of the constants `SELECTED` or `DESELECTED`. (This value can be misleading for `Checkbox` components that are part of a `CheckboxGroup`. If the user attempts to deselect a selected component, a `DESELECTED` event is delivered, but the `CheckboxGroup` immediately re-selects the component to enforce its requirement that at least one `Checkbox` be selected at all times.)

```
public class ItemEvent extends AWTEvent {
    // Public Constructor
            public ItemEvent(ItemSelectable source, int id, Object item, int stateChange);
    // Constants
            public static final int DESELECTED;
            public static final int ITEM_FIRST;
            public static final int ITEM_LAST;
            public static final int ITEM_STATE_CHANGED;
            public static final int SELECTED;
    // Public Instance Methods
            public Object getItem();
            public ItemSelectable getItemSelectable();
            public int getStateChange();
            public String paramString();  // Overrides AWTEvent
}
```

Hierarchy:

Object->EventObject(Serializable)->AWTEvent->ItemEvent

Passed To:

AWTEventMulticaster.itemStateChanged(), Checkbox.processItemEvent(), CheckboxMenuItem.processItemEvent(), Choice.processItemEvent(), ItemListener.itemStateChanged(), List.processItemEvent()

PREVIOUS | HOME | NEXT

java.awt.event.InputEvent (JDK 1.1) | BOOK INDEX | java.awt.event.ItemListener (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

21.14 java.awt.image.ReplicateScaleFilter (JDK 1.1)

This class implements an `ImageFilter` that scales an image to a specified pixel size. It uses a very simple scaling algorithm in which rows and columns of image pixels are duplicated or omitted as necessary to achieve the desired size. See `AreaAveragingScaleFilter` for a scaling filter that results in smoother images.

The methods of this class are `ImageConsumer` methods used for communication between the image filter and the `FilteredImageSource` that uses it. Applications do not usually call these methods directly.

The easiest way to use this filter is to call `Image.getScaledInstance()`, specifying an appropriate hint constant.

```
public class ReplicateScaleFilter extends ImageFilter {
    // Public Constructor
            public ReplicateScaleFilter(int width, int height);
    // Protected Instance Variables
            protected int destHeight;
            protected int destWidth;
            protected Object outpixbuf;
            protected int srcHeight;
            protected int srcWidth;
            protected int[] srccols;
            protected int[] srcrows;
    // Public Instance Methods
            public void setDimensions(int w, int h);  // Overrides ImageFilter
            public void setPixels(int x, int y, int w, int h, ColorModel model,
            public void setPixels'u'byte[] pixels, int off, int scansize);  //
Overrides ImageFilter
            public void setPixels(int x, int y, int w, int h, ColorModel model,
            public void setPixels'u'int[] pixels, int off, int scansize);  //
Overrides ImageFilter
            public void setProperties(Hashtable props);  // Overrides ImageFilter
}
```

Hierarchy:

`Object->ImageFilter(ImageConsumer, Cloneable)->ReplicateScaleFilter`

Extended By:

`AreaAveragingScaleFilter`

PREVIOUS | HOME | NEXT
java.awt.image.RGBImageFilter (JDK 1.0) | BOOK INDEX | The java.awt.peer Package

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

25.49 java.lang.Process (JDK 1.0)

This class describes a process that is running externally to the Java interpreter. Note that a `Process` is a very different thing than a `Thread`. The `Process` class is abstract and may not be instantiated. Call one of the `Runtime.exec()` methods to start a process and return a corresponding `Process` object.

`waitFor()` blocks until the `Process` exits. `exitValue()` returns the exit code of the process. `destroy()` kills the process. `getInputStream()` returns an `InputStream` from which you can read any bytes the process sends to its standard error stream. `getErrorStream()` returns an `InputStream` from which you can read any bytes the process sends to its standard output stream. `getOutputStream()` returns an `OutputStream` that you can use to send bytes to the standard input stream of the process.

```
public abstract class Process extends Object {
    // Default Constructor: public Process()
    // Public Instance Methods
            public abstract void destroy();
            public abstract int exitValue();
            public abstract InputStream getErrorStream();
            public abstract InputStream getInputStream();
            public abstract OutputStream getOutputStream();
            public abstract int waitFor() throws InterruptedException;
}
```

Returned By:

Runtime.exec()

java.lang.OutOfMemoryError (JDK 1.0) BOOK INDEX java.lang.Runnable (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

24.68 java.io.WriteAbortedException (JDK 1.1)

This exception is thrown when reading a stream of data that is incomplete because an exception was thrown while it was being written. The `detail` field may contain the exception that terminated the output stream. The `getMessage()` method has been overridden to include the message of this `detail` exception, if any.

```
public class WriteAbortedException extends ObjectStreamException {
    // Public Constructor
            public WriteAbortedException(String s, Exception ex);
    // Public Instance Variables
            public Exception detail;
    // Public Instance Methods
            public String getMessage();  // Overrides Throwable
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->IOException->ObjectStreamException->WriteAbortedException

22.17 java.awt.peer.MenuItemPeer (JDK 1.0)

```
public abstract interface MenuItemPeer extends MenuComponentPeer {
    // Public Instance Methods
            public abstract void disable();
            public abstract void enable();
        1.1  public abstract void setEnabled(boolean b);
            public abstract void setLabel(String label);
}
```

Extended By:

CheckboxMenuItemPeer, MenuPeer

Returned By:

Toolkit.createMenuItem()

28.9 java.net.HttpURLConnection (JDK 1.1)

This class is a specialization of `URLConnection`. An instance of this class is returned when the `openConnection()` method is called for a `URL` object that uses the HTTP protocol.

The many constants defined by this class are the status codes returned by HTTP servers. `setRequestMethod()` specifies what kind of HTTP request is made. The contents of this request must be sent through the `OutputStream` returned by the `getOutputStream()` method of the superclass. Once an HTTP request has been sent, `getResponseCode()` returns the HTTP server's response code as an integer, and `getResponseMessage()` returns the server's response message. The `disconnect()` method closes the connection, and the static `setFollowRedirects()` specifies whether URL connections that use the HTTP protocol should automatically follow redirect responses sent by HTTP servers.

In order to successfully use this class, you need to understand the details of the HTTP protocol.

```
public abstract class HttpURLConnection extends URLConnection {
    // Protected Constructor
            protected HttpURLConnection(URL u);
    // Response Code Constants
            public static final int HTTP_ACCEPTED, HTTP_BAD_GATEWAY, HTTP_BAD_METHOD;
            public static final int HTTP_BAD_REQUEST, HTTP_CLIENT_TIMEOUT, 
HTTP_CONFLICT;
            public static final int HTTP_CREATED, HTTP_ENTITY_TOO_LARGE, 
HTTP_FORBIDDEN;
            public static final int HTTP_GATEWAY_TIMEOUT, HTTP_GONE, 
HTTP_INTERNAL_ERROR;
            public static final int HTTP_LENGTH_REQUIRED, HTTP_MOVED_PERM, 
HTTP_MOVED_TEMP;
            public static final int HTTP_MULT_CHOICE, HTTP_NOT_ACCEPTABLE, 
HTTP_NOT_AUTHORITATIVE;
            public static final int HTTP_NOT_FOUND, HTTP_NOT_MODIFIED, 
HTTP_NO_CONTENT;
            public static final int HTTP_OK, HTTP_PARTIAL, HTTP_PAYMENT_REQUIRED;
            public static final int HTTP_PRECON_FAILED, HTTP_PROXY_AUTH, 
HTTP_REQ_TOO_LONG;
            public static final int HTTP_RESET, HTTP_SEE_OTHER, HTTP_SERVER_ERROR;
            public static final int HTTP_UNAUTHORIZED, HTTP_UNAVAILABLE, 
HTTP_UNSUPPORTED_TYPE;
            public static final int HTTP_USE_PROXY, HTTP_VERSION;
    // Protected Instance Variables
            protected String method;
            protected int responseCode;
            protected String responseMessage;
    // Class Methods
            public static boolean getFollowRedirects();
            public static void setFollowRedirects(boolean set);
    // Public Instance Methods
            public abstract void disconnect();
            public String getRequestMethod();
            public int getResponseCode() throws IOException;
            public String getResponseMessage() throws IOException;
            public void setRequestMethod(String method) throws ProtocolException;
            public abstract boolean usingProxy();
}
```

Hierarchy:

Object->URLConnection->HttpURLConnection

PREVIOUS | HOME | NEXT
java.net.FileNameMap (JDK 1.1) | BOOK INDEX | java.net.InetAddress (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

18.42 java.awt.MenuComponent (JDK 1.0)

This class is the superclass of all menu-related classes: You never need to instantiate a `MenuComponent` directly. `setFont()` specifies the font to be used for all text within the menu component.

```
public abstract class MenuComponent extends Object implements Serializable {
    // Default Constructor: public MenuComponent()
    // Public Instance Methods
        1.1  public final void dispatchEvent(AWTEvent e);
            public Font getFont();
        1.1  public String getName();
            public MenuContainer getParent();
        #   public MenuComponentPeer getPeer();
            public boolean postEvent(Event evt);
            public void removeNotify();
            public void setFont(Font f);
        1.1  public void setName(String name);
            public String toString();  // Overrides Object
    // Protected Instance Methods
            protected String paramString();
        1.1  protected void processEvent(AWTEvent e);
}
```

Extended By:

`MenuBar, MenuItem`

Passed To:

`Component.remove(), Frame.remove(), Menu.remove(), MenuBar.remove(), MenuContainer.remove()`

java.awt.MenuBar (JDK 1.0) BOOK INDEX java.awt.MenuContainer (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

22.16 java.awt.peer.MenuComponentPeer (JDK 1.0)

```
public abstract interface MenuComponentPeer {
    // Public Instance Methods
            public abstract void dispose();
}
```

Extended By:

MenuBarPeer, MenuItemPeer

Returned By:

MenuComponent.getPeer()

30.4 java.util.Dictionary (JDK 1.0)

This abstract class is the superclass of `Hashtable`. Other hashtable-like data structures might also extend this class. See `Hashtable` for more information.

```
public abstract class Dictionary extends Object {
    // Default Constructor: public Dictionary()
    // Public Instance Methods
            public abstract Enumeration elements();
            public abstract Object get(Object key);
            public abstract boolean isEmpty();
            public abstract Enumeration keys();
            public abstract Object put(Object key, Object value);
            public abstract Object remove(Object key);
            public abstract int size();
}
```

Extended By:

Hashtable

30.22 java.util.Stack (JDK 1.0)

This class implements a last-in-first-out stack of objects. `push()` puts an object on the top of the stack. `pop()` removes and returns the top object from the stack. `peek()` returns the top object without removing it.

```
public class Stack extends Vector {
    // Default Constructor: public Stack()
    // Public Instance Methods
            public boolean empty();
            public synchronized Object peek();
            public synchronized Object pop();
            public Object push(Object item);
            public synchronized int search(Object o);
}
```

Hierarchy:

Object->Vector(Cloneable, Serializable)->Stack

31.11 java.util.zip.Inflater (JDK 1.1)

This class implements the general ZLIB data decompression algorithm used by *gzip*, *PKZip*, and other data compression applications. It decompresses or "inflates" data compressed through the `Deflater` class.

The important methods of this class are `setInput()`, which specifies input data to be decompressed, and `inflate()`, which decompresses the input data into an output buffer. A number of other methods exist so that this class can be used for stream-based decompression, as it is in the higher-level classes such as `GZIPInputStream` and `ZipInputStream`. These stream-based classes are sufficient in most cases. Most applications do not need to use `Inflater` directly.

```
public class Inflater extends Object {
    // Public Constructors
            public Inflater(boolean nowrap);
            public Inflater();
    // Public Instance Methods
            public synchronized native void end();
            public synchronized boolean finished();
            public synchronized native int getAdler();
            public synchronized int getRemaining();
            public synchronized native int getTotalIn();
            public synchronized native int getTotalOut();
            public synchronized native int inflate(byte[] b, int off, int len) throws
DataFormatException;
            public int inflate(byte[] b) throws DataFormatException;
            public synchronized boolean needsDictionary();
            public synchronized boolean needsInput();
            public synchronized native void reset();
            public synchronized native void setDictionary(byte[] b, int off, int
len);
            public void setDictionary(byte[] b);
            public synchronized void setInput(byte[] b, int off, int len);
            public void setInput(byte[] b);
    // Protected Instance Methods
            protected void finalize();  // Overrides Object
}
```

Passed To:

InflaterInputStream()

Type Of:

InflaterInputStream.inf

18.51 java.awt.PrintJob (JDK 1.1)

A PrintJob object represents a single printing session or "job." The job may consist of one or more individual pages.

PrintJob is abstract so it cannot be instantiated directly. Instead, you must call the getPrintJob() method of the Toolkit object. Calling this method posts an appropriate print dialog box to request information from the user, such as which printer should be used. An application has no control over this process, but may pass a Properties object in which the dialog stores the user's printing preferences. This Properties object can then be reused when initiating subsequent print jobs.

Once a PrintJob object has been obtained from the Toolkit object, you call the getGraphics() method of PrintJob to obtain a Graphics object. Any drawing done with this Graphics object is printed rather than being displayed on-screen. The object returned by getGraphics() implements the PrintGraphics interface. Do not make any assumptions about the initial state of the Graphics object; in particular note that you must specify a font before you can draw any text.

When you are done drawing all the desired output on a page, call the dispose() method of the Graphics object to force the current page to be printed. You can call PrintJob.getGraphics() and Graphics.dispose() repeatedly to print any number of pages required by your application. Note, however, that if the lastPageFirst() method returns true, the user has requested that pages be printed in reverse order. It is up to your application to implement this feature.

The getPageDimension() method returns the size of the page in pixels. getPageResolution() returns the resolution of the page in pixels per inch. This resolution is closer to a screen resolution (70 to 100 pixels per inch) rather than a typical printer resolution (300 to 600 pixels per inch). This means that on-screen drawings can be drawn directly to the printer without scaling. It also means, however, that you cannot take full advantage of the extra resolution provided by printers.

When you are done with a PrintJob, and you have called dispose() on the Graphics object returned by getGraphics(), you should call end() to terminate the job.

```
public abstract class PrintJob extends Object {
    // Default Constructor: public PrintJob()
    // Public Instance Methods
            public abstract void end();
            public void finalize();  // Overrides Object
            public abstract Graphics getGraphics();
            public abstract Dimension getPageDimension();
            public abstract int getPageResolution();
            public abstract boolean lastPageFirst();
}
```

Returned By:

`PrintGraphics.getPrintJob(), Toolkit.getPrintJob()`

PREVIOUS | HOME | NEXT

java.awt.PrintGraphics (JDK 1.1) | BOOK INDEX | java.awt.Rectangle (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

24.45 java.io.OptionalDataException (JDK 1.1)

This exception is thrown by the `readObject()` method of an `ObjectInputStream` when it encounters primitive type data where it expects object data. Despite the exception name, this data is not "optional," and object deserialization is aborted.

```
public class OptionalDataException extends ObjectStreamException {
    // No Constructor
    // Public Instance Variables
            public boolean eof;
            public int length;
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->IOException->ObjectStreamException->OptionalDataException

Thrown By:

ObjectInputStream.readObject()

26.2 java.lang.reflect.Constructor (JDK 1.1)

This class represents a constructor method of a class. Instances of `Constructor` are obtained by calling `getConstructor()` and related methods of `java.lang.Class`. `Constructor` implements the `Member` interface, so you can use the methods of that interface to obtain the constructor name, modifiers, and declaring class. In addition, `getParameterTypes()` and `getExceptionTypes()` also return important information about the represented constructor.

In addition to these methods that return information about the constructor, the `newInstance()` method allows the constructor to be invoked with an array of arguments in order to create a new instance of the class that declares the constructor. If any of the arguments to the constructor are of primitive types, they must be converted to their corresponding wrapper object types in order to be passed to `newInstance()`. If the constructor causes an exception, the `Throwable` object it throws is wrapped within the `InvocationTargetException` that is thrown by `newInstance()`. Note that `newInstance()` is much more useful than the `newInstance()` method of `java.lang.Class` because it can pass arguments to the constructor.

```
public final class Constructor extends Object implements Member {
    // No Constructor
    // Public Instance Methods
            public boolean equals(Object obj);  // Overrides Object
            public Class getDeclaringClass();  // From Member
            public Class[] getExceptionTypes();
            public native int getModifiers();  // From Member
            public String getName();  // From Member
            public Class[] getParameterTypes();
            public int hashCode();  // Overrides Object
            public native Object newInstance(Object[] initargs) throws
InstantiationException, IllegalAccessException, IllegalArgumentException,
InvocationTargetException;
            public String toString();  // Overrides Object
}
```

Returned By:

Class.getConstructor(), Class.getConstructors(), Class.getDeclaredConstructor(), Class.getDeclaredConstructors()

PREVIOUS | HOME | NEXT
java.lang.reflect.Array (JDK 1.1) | BOOK INDEX | java.lang.reflect.Field (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

PREVIOUS

NEXT

19.3 java.awt.datatransfer.DataFlavor (JDK 1.1)

Objects of this type define a data type or format for the purposes of data transfer. A `DataFlavor` is characterized by two values. The first is a descriptive human-readable name, passed to a `DataFlavor()` constructor or set with `setHumanPresentableName()`.

The second component of a `DataFlavor` specifies the type of data to be transferred in a more machine-readable way. The two `DataFlavor()` constructors allow two distinct ways to specify this data type. One way is by directly specifying the representation class of the data. This is a `Class` object that defines the Java type that the recipient of a data transfer is passed. The other way to specify the data type represented by a `DataFlavor` is to pass a string that specifies the MIME type of the data to be transferred.

When you construct a `DataFlavor` object by specifying the representation type of the data, the MIME type of the `DataFlavor` is automatically set to:

```
application/x-java-serialized-object class=classname
```

This indicates that the object is to be transferred using the data format of the Java object serialization protocol. When you pass a MIME type string to the `DataFlavor()` constructor, on the other hand, the representation class of the `DataFlavor` is automatically set to the `Class` object for `java.io.InputStream`. This means that the recipient of the data transfer is given an `InputStream` object from which it can read and parse data in the specified MIME format.

Because the same MIME type can be specified with a number of slightly different strings, the `DataFlavor` class converts the MIME type to a canonical form so that it can be uniquely identified and compared. Use `isMimeTypeEqual()` to compare the MIME type of a `DataFlavor` object with another MIME type, or with the MIME type of another `DataFlavor`.

Because textual data is so often transferred, the `DataFlavor` class defines constants for two commonly-used data flavors. `stringFlavor` represents text transferred as a `String` object, while `plainTextFlavor` represents text transferred through an `InputStream`, using the `text/plain` MIME type.

```
public class DataFlavor extends Object {
    // Public Constructors
            public DataFlavor(Class representationClass, String 
humanPresentableName);
            public DataFlavor(String mimeType, String humanPresentableName);
    // Class Variables
            public static DataFlavor plainTextFlavor;
            public static DataFlavor stringFlavor;
    // Public Instance Methods
            public boolean equals(DataFlavor dataFlavor);
            public String getHumanPresentableName();
            public String getMimeType();
            public Class getRepresentationClass();
            public boolean isMimeTypeEqual(String mimeType);
            public final boolean isMimeTypeEqual(DataFlavor dataFlavor);
            public void setHumanPresentableName(String humanPresentableName);
    // Protected Instance Methods
            protected String normalizeMimeType(String mimeType);
            protected String normalizeMimeTypeParameter(String parameterName, String 
parameterValue);
}
```

Passed To:

`DataFlavor.equals(), DataFlavor.isMimeTypeEqual(), StringSelection.getTransferData(), StringSelection.isDataFlavorSupported(), Transferable.getTransferData(), Transferable.isDataFlavorSupported(), UnsupportedFlavorException()`

Returned By:

`StringSelection.getTransferDataFlavors(), Transferable.getTransferDataFlavors()`

Type Of:

`DataFlavor.plainTextFlavor, DataFlavor.stringFlavor`

PREVIOUS	HOME	NEXT
java.awt.datatransfer.ClipboardOwner (JDK 1.1)	BOOK INDEX	java.awt.datatransfer.StringSelection (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

18.19 java.awt.Dimension (JDK 1.0)

This class has two public instance variables that describe the width and height of something. The `width` and `height` fields are public and may be manipulated directly.

```
public class Dimension extends Object implements Serializable {
    // Public Constructors
            public Dimension();
            public Dimension(Dimension d);
            public Dimension(int width, int height);
    // Public Instance Variables
            public int height;
            public int width;
    // Public Instance Methods
        1.1  public boolean equals(Object obj);  // Overrides Object
        1.1  public Dimension getSize();
        1.1  public void setSize(Dimension d);
        1.1  public void setSize(int width, int height);
            public String toString();  // Overrides Object
}
```

Passed To:

`Applet.resize(), Component.resize(), Component.setSize(), Dimension(), Dimension.setSize(), Rectangle(), Rectangle.setSize()`

Returned By:

`Many methods`

26.3 java.lang.reflect.Field (JDK 1.1)

This class represents a field of a class. Instances of `Field` are obtained by calling the `getField()` and related methods of `java.lang.Class`. `Field` implements the `Member` interface, so once you have obtained a `Field` object, you can use `getName()`, `getModifiers()`, and `getDeclaringClass()` to determine the name, modifiers, and class of the field. Additionally, `getType()` returns the type of the field.

The `set()` method sets the value of the represented field for a specified object to a given value. (If the represented field is `static`, then no object need be specified for it to be set upon, of course.) If the field is of a primitive type, its value can be specified using a wrapper object of type `Boolean`, `Integer`, and so on, or it can be set using the `setBoolean()`, `setInt()`, and related methods.

Similarly, the `get()` method queries the value of the represented field for a specified object and returns the field value as an `Object`. Various other methods query the field value and return it as various primitive types.

```
public final class Field extends Object implements Member {
    // No Constructor
    // Public Instance Methods
            public boolean equals(Object obj);  // Overrides Object
            public native Object get(Object obj) throws IllegalArgumentException, 
IllegalAccessException;
            public native boolean getBoolean(Object obj) throws 
IllegalArgumentException, IllegalAccessException;
            public native byte getByte(Object obj) throws IllegalArgumentException, 
IllegalAccessException;
            public native char getChar(Object obj) throws IllegalArgumentException, 
IllegalAccessException;
            public Class getDeclaringClass();  // From Member
            public native double getDouble(Object obj) throws 
IllegalArgumentException, IllegalAccessException;
            public native float getFloat(Object obj) throws IllegalArgumentException, 
IllegalAccessException;
            public native int getInt(Object obj) throws IllegalArgumentException, 
IllegalAccessException;
            public native long getLong(Object obj) throws IllegalArgumentException, 
IllegalAccessException;
            public native int getModifiers();  // From Member
            public String getName();  // From Member
            public native short getShort(Object obj) throws IllegalArgumentException, 
IllegalAccessException;
            public Class getType();
            public int hashCode();  // Overrides Object
            public native void set(Object obj, Object value) throws 
IllegalArgumentException, IllegalAccessException;
            public native void setBoolean(Object obj, boolean z) throws 
IllegalArgumentException, IllegalAccessException;
            public native void setByte(Object obj, byte b) throws 
IllegalArgumentException, IllegalAccessException;
            public native void setChar(Object obj, char c) throws 
IllegalArgumentException, IllegalAccessException;
            public native void setDouble(Object obj, double d) throws 
IllegalArgumentException, IllegalAccessException;
            public native void setFloat(Object obj, float f) throws 
IllegalArgumentException, IllegalAccessException;
            public native void setInt(Object obj, int i) throws 
IllegalArgumentException, IllegalAccessException;
            public native void setLong(Object obj, long l) throws 
IllegalArgumentException, IllegalAccessException;
            public native void setShort(Object obj, short s) throws 
IllegalArgumentException, IllegalAccessException;
            public String toString();  // Overrides Object
}
```

Returned By:

Class.getDeclaredField(), Class.getDeclaredFields(), Class.getField(), Class.getFields()

PREVIOUS | HOME | NEXT
java.lang.reflect.Constructor (JDK 1.1) | BOOK INDEX | java.lang.reflect.InvocationTargetException (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

28.10 java.net.InetAddress (JDK 1.0)

This class represents an Internet address, and is used when creating `DatagramPacket` or `Socket` objects. The class does not have a public constructor function, but instead supports three static methods which return one or more instances of `InetAddress`. `getLocalHost()` returns an `InetAddress` for the local host. `getByName()` returns the `InetAddress` of a host specified by name. `getAllByName()` returns an array of `InetAddress` that represents all of the available addresses for a host specified by name. Instance methods are `getHostName()`, which returns the hostname of an `InetAddress`, and `getAddress()`, which returns the Internet IP address as an array of bytes, with the highest-order byte as the first element of the array.

```
public final class InetAddress extends Object implements Serializable {
    // No Constructor
    // Class Methods
            public static InetAddress[] getAllByName(String host) throws
UnknownHostException;
            public static InetAddress getByName(String host) throws
UnknownHostException;
            public static InetAddress getLocalHost() throws UnknownHostException;
    // Public Instance Methods
            public boolean equals(Object obj);  // Overrides Object
            public byte[] getAddress();
            public String getHostAddress();
            public String getHostName();
            public int hashCode();  // Overrides Object
        1.1public boolean isMulticastAddress();
            public String toString();  // Overrides Object
}
```

Passed To:

DatagramPacket(), DatagramPacket.setAddress(), DatagramSocket(), DatagramSocketImpl.bind(), DatagramSocketImpl.join(), DatagramSocketImpl.leave(), DatagramSocketImpl.peek(), MulticastSocket.joinGroup(), MulticastSocket.leaveGroup(), MulticastSocket.setInterface(), SecurityManager.checkMulticast(), ServerSocket(), Socket(), SocketImpl.bind(), SocketImpl.connect()

Returned By:

DatagramPacket.getAddress(), DatagramSocket.getLocalAddress(), InetAddress.getAllByName(), InetAddress.getByName(), InetAddress.getLocalHost(), MulticastSocket.getInterface(), ServerSocket.getInetAddress(), Socket.getInetAddress(), Socket.getLocalAddress(), SocketImpl.getInetAddress()

Type Of:

SocketImpl.address

18.45 java.awt.MenuShortcut (JDK 1.1)

This class represents a keystroke used to select a `MenuItem` without actually pulling down the menu. A `MenuShortcut` object can be specified for a `MenuItem` when the `MenuItem` is created or by calling the item's `setShortcut()` method. The keystroke sequence for the menu shortcut automatically appears in the label for the menu item, so you need not add this information yourself.

When you create a `MenuShortcut`, you specify the key code of the shortcut--this is one of the `VK_` constants defined by `java.awt.event.KeyEvent`, and is not always the same as a corresponding character code. You may also optionally specify a `boolean` value that, if `true`, indicates that the `MenuShortcut` requires the **Shift** key to be held down.

Note that menu shortcuts are triggered in a platform-dependent way. When you create a shortcut, you specify only the keycode and an optional **Shift** modifier. The shortcut is not triggered, however, unless an additional modifier key is held down. On Windows platforms, for example, the **Ctrl** key is used for menu shortcuts. You can query the platform-specific menu shortcut key with `Toolkit.getMenuShortcutKeyMask()`.

```
public class MenuShortcut extends Object implements Serializable {
    // Public Constructors
            public MenuShortcut(int key);
            public MenuShortcut(int key, boolean useShiftModifier);
    // Public Instance Methods
            public boolean equals(MenuShortcut s);
            public int getKey();
            public String toString();  // Overrides Object
            public boolean usesShiftModifier();
    // Protected Instance Methods
            protected String paramString();
}
```

Passed To:

`MenuBar.deleteShortcut(), MenuBar.getShortcutMenuItem(), MenuItem(), MenuItem.setShortcut(), MenuShortcut.equals()`

Returned By:

`MenuItem.getShortcut()`

26.6 java.lang.reflect.Method (JDK 1.1)

This class represents a method. Instances of `Method` are obtained by calling the `getMethod()` and related methods of `java.lang.Class`. `Method` implements the `Member` interface, so you can use the methods of that interface to obtain the method name, modifiers, and declaring class. In addition, `getReturnType()`, `getParameterTypes()`, and `getExceptionTypes()` also return important information about the represented method.

Perhaps most important, the `invoke()` method allows the method represented by the `Method` object to be invoked with a specified array of argument values. If any of the arguments are of primitive types, they must be converted to their corresponding wrapper object types in order to be passed to `invoke()`. If the represented method is an instance method (i.e., if it is not `static`), the instance on which it should be invoked must also be passed to `invoke()`. The return value of the represented method is returned by `invoke()`. If the return value is a primitive value, it is first converted to the corresponding wrapper type. If the invoked method causes an exception, the `Throwable` object it throws is wrapped within the `InvocationTargetException` that is thrown by `invoke()`.

```
public final class Method extends Object implements Member {
    // No Constructor
    // Public Instance Methods
            public boolean equals(Object obj);  // Overrides Object
            public Class getDeclaringClass();  // From Member
            public Class[] getExceptionTypes();
            public native int getModifiers();  // From Member
            public String getName();  // From Member
            public Class[] getParameterTypes();
            public Class getReturnType();
            public int hashCode();  // Overrides Object
            public native Object invoke(Object obj, Object[] args) throws 
IllegalAccessException, IllegalArgumentException, InvocationTargetException;
            public String toString();  // Overrides Object
}
```

Passed To:

EventSetDescriptor(), IndexedPropertyDescriptor(), MethodDescriptor(), PropertyDescriptor()

Returned By:

Class.getDeclaredMethod(), Class.getDeclaredMethods(), Class.getMethod(), Class.getMethods(), EventSetDescriptor.getAddListenerMethod(), EventSetDescriptor.getListenerMethods(), EventSetDescriptor.getRemoveListenerMethod(), IndexedPropertyDescriptor.getIndexedReadMethod(), IndexedPropertyDescriptor.getIndexedWriteMethod(), MethodDescriptor.getMethod(), PropertyDescriptor.getReadMethod(), PropertyDescriptor.getWriteMethod()

PREVIOUS | HOME | NEXT
java.lang.reflect.Member (JDK 1.1) | BOOK INDEX | java.lang.reflect.Modifier (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

18.47 java.awt.Point (JDK 1.0)

This class holds the X and Y coordinates of a two-dimensional point. The `move()` method sets the coordinates, and the `translate()` method adds the specified values to the coordinates. The `x` and `y` fields are public and may be manipulated directly.

```
public class Point extends Object implements Serializable {
    // Public Constructors
        1.1  public Point();
        1.1  public Point(Point p);
            public Point(int x, int y);
    // Public Instance Variables
            public int x;
            public int y;
    // Public Instance Methods
            public boolean equals(Object obj);  // Overrides Object
        1.1  public Point getLocation();
            public int hashCode();  // Overrides Object
            public void move(int x, int y);
        1.1  public void setLocation(Point p);
        1.1  public void setLocation(int x, int y);
            public String toString();  // Overrides Object
            public void translate(int x, int y);
}
```

Passed To:

`Component.contains(), Component.getComponentAt(), Component.setLocation(), Container.getComponentAt(), Point(), Point.setLocation(), Polygon.contains(), Rectangle(), Rectangle.add(), Rectangle.contains(), Rectangle.setLocation(),` ScrollPane.setScrollPosition()

Returned By:

Component.getLocation(), Component.getLocationOnScreen(), Component.location(), ComponentPeer.getLocationOnScreen(), GridBagLayout.getLayoutOrigin(), GridBagLayout.location(), MouseEvent.getPoint(), Point.getLocation(), Rectangle.getLocation(), ScrollPane.getScrollPosition()

PREVIOUS | HOME | NEXT
java.awt.Panel (JDK 1.0) | BOOK INDEX | java.awt.Polygon (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

24.17 java.io.FileDescriptor (JDK 1.0)

This class is a platform-independent representation of a low-level handle to an open file or an open socket. The static `in`, `out`, and `err` variables are `FileDescriptor` objects that represent the system standard input, output, and error streams, respectively. There is no public constructor method to create a `FileDescriptor` object. You can obtain one with the `getFD()` method of `FileInputStream`, `FileOutputStream`, and `RandomAccessFile`.

```
public final class FileDescriptor extends Object {
    // Default Constructor: public FileDescriptor()
    // Constants
            public static final FileDescriptor err;
            public static final FileDescriptor in;
            public static final FileDescriptor out;
    // Public Instance Methods
        1.1  public native void sync() throws SyncFailedException;
            public native boolean valid();
}
```

Passed To:

FileInputStream(), FileOutputStream(), FileReader(), FileWriter(), SecurityManager.checkRead(), SecurityManager.checkWrite()

Returned By:

DatagramSocketImpl.getFileDescriptor(), FileInputStream.getFD(), FileOutputStream.getFD(), RandomAccessFile.getFD(), SocketImpl.getFileDescriptor()

Type Of:

DatagramSocketImpl.fd, FileDescriptor.err, FileDescriptor.in, FileDescriptor.out, SocketImpl.fd

25.51 java.lang.Runtime (JDK 1.0)

This class encapsulates a number of platform-dependent system functions. The static method `getRuntime()` returns the `Runtime` object for the current platform, and this object can be used to perform system functions in a platform-independent way.

`exec()` starts a new process running externally to the interpreter. Note that any processes run outside of Java may be system-dependent.

`exit()` causes the Java interpreter to exit and return a specified return code.

`freeMemory()` returns the approximate amount of free memory. `totalMemory()` returns the total amount of memory available to the Java interpreter. `gc()` forces the garbage collector to run synchronously, which may free up more memory. Similarly, `runFinalization()` forces the `finalize()` methods of unreferenced objects to be run immediately. This may free up system resources that those objects were holding.

`load()` loads a dynamic library with a fully specified path name. `loadLibrary()` loads a dynamic library with only the library name specified; it looks in platform-dependent locations for the specified library. These libraries generally contain native code definitions for native methods.

`traceInstructions()` and `traceMethodCalls()` enable and disable tracing by the interpreter.

Note that some of the `Runtime` methods are more commonly called via the static methods of the `System` class.

```
public class Runtime extends Object {
    // No Constructor
    // Class Methods
            public static Runtime getRuntime();
        1.1public static void runFinalizersOnExit(boolean value);
    // Public Instance Methods
            public Process exec(String command) throws IOException;
            public Process exec(String command, String[] envp) throws IOException;
            public Process exec(String[] cmdarray) throws IOException;
            public Process exec(String[] cmdarray, String[] envp) throws IOException;
            public void exit(int status);
            public native long freeMemory();
            public native void gc();
        #   public InputStream getLocalizedInputStream(InputStream in);
        #   public OutputStream getLocalizedOutputStream(OutputStream out);
            public synchronized void load(String filename);
            public synchronized void loadLibrary(String libname);
            public native void runFinalization();
            public native long totalMemory();
            public native void traceInstructions(boolean on);
            public native void traceMethodCalls(boolean on);
}
```

Returned By:

Runtime.getRuntime()

PREVIOUS | HOME | NEXT
java.lang.Runnable (JDK 1.0) | BOOK INDEX | java.lang.RuntimeException (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

25.64 java.lang.Throwable (JDK 1.0)

This is the root class of the Java exception and error hierarchy. All exceptions and errors are subclasses of `Throwable`. The `getMessage()` method retrieves any error message associated with the exception or error. `printStackTrace()` prints a stack trace that shows where the exception occurred. `fillInStackTrace()` extends the stack trace when the exception is partially handled, and then re-thrown.

```
public class Throwable extends Object implements Serializable {
    // Public Constructors
            public Throwable();
            public Throwable(String message);
    // Public Instance Methods
            public native Throwable fillInStackTrace();
        1.1public String getLocalizedMessage();
            public String getMessage();
            public void printStackTrace();
            public void printStackTrace(PrintStream s);
        1.1public void printStackTrace(PrintWriter s);
            public String toString();  // Overrides Object
}
```

Extended By:

Error, Exception

Passed To:

ExceptionInInitializerError(), InvocationTargetException(), Thread.stop(), ThreadGroup.uncaughtException()

Returned By:

ExceptionInInitializerError.getException(), InvocationTargetException.getTargetException(), Throwable.fillInStackTrace()

Thrown By:

Object.finalize()

PREVIOUS | HOME | NEXT

java.lang.ThreadGroup (JDK 1.0) | BOOK INDEX | java.lang.UnknownError (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

21.2 java.awt.image.ColorModel (JDK 1.0)

This abstract class defines a color model--i.e., a scheme for representing a color. Subclasses implement the methods of this interface to convert their particular representation of a pixel value into the default RGB color model. The static method `getRGBDefault()` returns a `ColorModel` object that implements the default color model--RGB plus alpha transparency. You generally never need to call the instance methods of a `ColorModel`--they are called internally by other image manipulation classes.

See also `DirectColorModel` and `IndexColorModel`.

```
public abstract class ColorModel extends Object {
    // Public Constructor
            public ColorModel(int bits);
    // Protected Instance Variables
            protected int pixel_bits;
    // Class Methods
            public static ColorModel getRGBdefault();
    // Public Instance Methods
            public void finalize();  // Overrides Object
            public abstract int getAlpha(int pixel);
            public abstract int getBlue(int pixel);
            public abstract int getGreen(int pixel);
            public int getPixelSize();
            public int getRGB(int pixel);
            public abstract int getRed(int pixel);
}
```

Extended By:

`DirectColorModel, IndexColorModel`

Passed To:

AreaAveragingScaleFilter.setPixels(), CropImageFilter.setPixels(),
ImageConsumer.setColorModel(), ImageConsumer.setPixels(),
ImageFilter.setColorModel(), ImageFilter.setPixels(),
MemoryImageSource(), MemoryImageSource.newPixels(),
PixelGrabber.setColorModel(), PixelGrabber.setPixels(),
ReplicateScaleFilter.setPixels(), RGBImageFilter.setColorModel(),
RGBImageFilter.setPixels(), RGBImageFilter.substituteColorModel()

Returned By:

ColorModel.getRGBdefault(), Component.getColorModel(),
ComponentPeer.getColorModel(), PixelGrabber.getColorModel(),
Toolkit.getColorModel()

Type Of:

RGBImageFilter.newmodel, RGBImageFilter.origmodel

PREVIOUS	HOME	NEXT
java.awt.image.AreaAveragingScaleFilter (JDK 1.1)	BOOK INDEX	java.awt.image.CropImageFilter (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

23.17 java.beans.PropertyEditorManager (JDK 1.1)

The PropertyEditorManager class is never meant to be instantiated; it defines static methods for registering and looking up PropertyEditor classes for a specified property type.

A Java bean may specify a particular PropertyEditor class for a given property by specifying it in a PropertyDescriptor object for the property. If it does not do this, the PropertyEditorManager is used to register and look up editors. A bean or an application builder tool may call the registerEditor() method to register a PropertyEditor for properties of a specified type. Application builders and bean Customizer classes may call the findEditor() method to obtain a PropertyEditor for a given property type.

If no editor has been registered for a given type, the PropertyEditorManager attempts to locate one. For a type *x*, it looks for a class *x*Editor first in the same package as *x*, and then in each package listed in the property editor search path.

```
public class PropertyEditorManager extends Object {
    // Default Constructor: public PropertyEditorManager()
    // Class Methods
            public static PropertyEditor findEditor(Class targetType);
            public static String[] getEditorSearchPath();
            public static void registerEditor(Class targetType, Class editorClass);
            public static void setEditorSearchPath(String[] path);
}
```

24.13 java.io.DataOutputStream (JDK 1.0)

This class is a subclass of `FilterOutputStream` that allows you to write Java primitive data types in a portable binary format. Create a `DataOutputStream` by specifying the `OutputStream` that is to be filtered in the call to the constructor.

Many of this class's methods write a single Java primitive type, in binary format to the output stream. `write()` writes a single byte, an array, or a subarray of bytes. `flush()` forces any buffered data to be output. `size()` returns the number of bytes written so far.

`writeUTF()` outputs a Java string of Unicode characters using a slightly modified version of the UTF-8 "transformation format." UTF-8 is an ASCII-compatible encoding of Unicode characters that is often used for the transmission and storage of Unicode text. Except for the `writeUTF()` method, this class is used for binary output of data. Textual output should be done with `PrintWriter`, or `PrintStream` in Java 1.0.

`DataOutputStream` only has methods to output primitive types. Use `ObjectOutputStream` to output object values.

```
public class DataOutputStream extends FilterOutputStream implements DataOutput {
    // Public Constructor
            public DataOutputStream(OutputStream out);
    // Protected Instance Variables
            protected int written;
    // Public Instance Methods
            public void flush() throws IOException;  // Overrides FilterOutputStream
            public final int size();
            public synchronized void write(int b) throws IOException;  // Overrides
FilterOutputStream
            public synchronized void write(byte[] b, int off, int len) throws
IOException;  // Overrides FilterOutputStream
            public final void writeBoolean(boolean v) throws IOException;  // From
DataOutput
            public final void writeByte(int v) throws IOException;  // From
DataOutput
            public final void writeBytes(String s) throws IOException;  // From
DataOutput
            public final void writeChar(int v) throws IOException;  // From
DataOutput
            public final void writeChars(String s) throws IOException;  // From
DataOutput
            public final void writeDouble(double v) throws IOException;  // From
DataOutput
            public final void writeFloat(float v) throws IOException;  // From
DataOutput
            public final void writeInt(int v) throws IOException;  // From DataOutput
            public final void writeLong(long v) throws IOException;  // From
DataOutput
            public final void writeShort(int v) throws IOException;  // From
DataOutput
            public final void writeUTF(String str) throws IOException;  // From
DataOutput
}
```

Hierarchy:

Object->OutputStream->FilterOutputStream->DataOutputStream(DataOutput)

PREVIOUS | HOME | NEXT
java.io.DataOutput (JDK 1.0) | BOOK INDEX | java.io.EOFException (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

18.32 java.awt.Image (JDK 1.0)

This abstract class represents a displayable image in a platform-independent way. An `Image` object may not be instantiated directly through a constructor; it must be obtained through a call like `Applet.getImage()` or `Component.createImage()`. `getSource()` method returns the `ImageProducer` object that produces the image data. `getGraphics()` returns a `Graphics` object that can be used for drawing into offscreen images (but not images that are downloaded or generated by an `ImageProducer`).

```
public abstract class Image extends Object {
    // Default Constructor: public Image()
    // Constants
        1.1  public static final int SCALE_AREA_AVERAGING;
        1.1  public static final int SCALE_DEFAULT;
        1.1  public static final int SCALE_FAST;
        1.1  public static final int SCALE_REPLICATE;
        1.1  public static final int SCALE_SMOOTH;
            public static final Object UndefinedProperty;
    // Public Instance Methods
            public abstract void flush();
            public abstract Graphics getGraphics();
            public abstract int getHeight(ImageObserver observer);
            public abstract Object getProperty(String name, ImageObserver observer);
        1.1  public Image getScaledInstance(int width, int height, int hints);
            public abstract ImageProducer getSource();
            public abstract int getWidth(ImageObserver observer);
}
```

Passed To:

`Component.checkImage(), Component.imageUpdate(), Component.prepareImage(), ComponentPeer.checkImage(), ComponentPeer.prepareImage(), Frame.setIconImage(), FramePeer.setIconImage(), Graphics.drawImage(), ImageObserver.imageUpdate(), MediaTracker.addImage(), MediaTracker.removeImage(), PixelGrabber(), Toolkit.checkImage(), Toolkit.prepareImage()`

Returned By:

`Applet.getImage(), AppletContext.getImage(), BeanInfo.getIcon(), Component.createImage(), ComponentPeer.createImage(), Frame.getIconImage(), Image.getScaledInstance(), SimpleBeanInfo.getIcon(), SimpleBeanInfo.loadImage(),` `Toolkit.createImage(), Toolkit.getImage()`

20.12 java.awt.event.FocusEvent (JDK 1.1)

An event of this type indicates that a `Component` has gained or lost focus on a temporary or permanent basis.

Use the inherited `getComponent()` method to determine which component has gained or lost focus. Use `getID()` to determine the type of focus event; it returns `FOCUS_GAINED` or `FOCUS_LOST`.

When focus is lost, you can call `isTemporary()` to determine whether it is a temporary loss of focus. Temporary focus loss occurs when the window that contains the component loses focus, for example, or when focus is temporarily diverted to a popup menu or a scrollbar. Similarly, you can also use `isTemporary()` to determine whether focus is being granted to a component on a temporary basis.

```
public class FocusEvent extends ComponentEvent {
    // Public Constructors
            public FocusEvent(Component source, int id, boolean temporary);
            public FocusEvent(Component source, int id);
    // Constants
            public static final int FOCUS_FIRST;
            public static final int FOCUS_GAINED;
            public static final int FOCUS_LAST;
            public static final int FOCUS_LOST;
    // Public Instance Methods
            public boolean isTemporary();
            public String paramString();  // Overrides ComponentEvent
}
```

Hierarchy:

`Object->EventObject(Serializable)->AWTEvent->ComponentEvent->FocusEvent`

Passed To:

`AWTEventMulticaster.focusGained(), AWTEventMulticaster.focusLost(),` `Component.processFocusEvent()`, `FocusAdapter.focusGained()`, `FocusAdapter.focusLost()`, `FocusListener.focusGained()`, `FocusListener.focusLost()`

20.11 java.awt.event.FocusAdapter (JDK 1.1)

This class is a trivial implementation of `FocusListener`; it provides empty bodies for each of the methods of that interface. When you are not interested in all of these methods, it is often easier to subclass `FocusAdapter` than it is to implement `FocusListener` from scratch.

```
public abstract class FocusAdapter extends Object implements FocusListener {
    // Default Constructor: public FocusAdapter()
    // Public Instance Methods
            public void focusGained(FocusEvent e);  // From FocusListener
            public void focusLost(FocusEvent e);  // From FocusListener
}
```

Hierarchy:

```
Object->FocusAdapter(FocusListener(EventListener))
```

20.13 java.awt.event.FocusListener (JDK 1.1)

This interface defines the methods that an object must implement to "listen" for focus events on AWT components. When a `FocusEvent` occurs, an AWT component notifies its registered `FocusListener` objects by invoking one of their methods.

An easy way to implement this interface is by subclassing the `FocusAdapter` class.

```
public abstract interface FocusListener extends EventListener {
    // Public Instance Methods
            public abstract void focusGained(FocusEvent e);
            public abstract void focusLost(FocusEvent e);
}
```

Implemented By:

`AWTEventMulticaster, FocusAdapter`

Passed To:

`AWTEventMulticaster.add(), AWTEventMulticaster.remove(), Component.addFocusListener(), Component.removeFocusListener()`

Returned By:

`AWTEventMulticaster.add(), AWTEventMulticaster.remove()`

24.43 java.io.ObjectStreamClass (JDK 1.1)

This class is used to represent a class that is being serialized. An `ObjectStreamClass` object contains the name of a class and its unique version identifier. This class does not have a constructor; you should use the static `lookup()` method to obtain an `ObjectStreamClass` object for a given `Class` object. The `forClass()` instance method performs the opposite operation--it returns the `Class` object that corresponds to a given `ObjectStreamClass`.

Most applications never need to use this class.

```
public class ObjectStreamClass extends Object implements Serializable {
    // No Constructor
    // Class Methods
            public static ObjectStreamClass lookup(Class cl);
    // Public Instance Methods
            public Class forClass();
            public String getName();
            public long getSerialVersionUID();
            public String toString();  // Overrides Object
}
```

Passed To:

ObjectInputStream.resolveClass()

Returned By:

ObjectStreamClass.lookup()

java.io.ObjectOutputStream (JDK 1.1) BOOK INDEX java.io.ObjectStreamException (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

25.8 java.lang.Class (JDK 1.0)

This class represents a Java class or interface, or, in Java 1.1, any Java type. There is one `Class` object for each class that is loaded into the Java Virtual Machine, and in Java 1.1 there are special `Class` objects that represent the Java primitive types. The `TYPE` constants defined by `Boolean`, `Integer`, and the other primitive "wrapper classes" hold these special `Class` objects. Array types are also represented by `Class` objects in Java 1.1.

There is no constructor for this class. You can obtain a `Class` object by calling the `getClass()` method of any instance of the desired class. In Java 1.1, a `Class` object may also be referred to by appending `.class` to the name of a class. Finally, and most interestingly, a class can be dynamically loaded by passing its fully-qualified name (i.e., package name plus class name) to the static `Class.forName()` method. This method loads the named class, if it is not already loaded, into the Java interpreter, and returns a `Class` object for it.

The `newInstance()` method creates an instance of a given class--this allows you to create instances of dynamically loaded classes for which you cannot use the `new` keyword. Note that this method only works when the target class has a no-argument constructor. See `newInstance()` in `java.lang.reflect.Constructor` for a more powerful way to instantiate dynamically loaded classes.

`getName()` returns the name of the class. `getSuperclass()` returns its superclass. `isInterface()` tests whether the `Class` object represents an interface, and `getInterfaces()` returns an array of the interfaces that this class implements. The various other `get` and `is` methods return other information about the represented class, and form part of the Java Reflection API, along with the classes in `java.lang.reflect`.

```
public final class Class extends Object implements Serializable {
    // No Constructor
    // Class Methods
            public static native Class forName(String className) throws 
ClassNotFoundException;
    // Public Instance Methods
            public native ClassLoader getClassLoader();
        1.1public Class[] getClasses();
        1.1public native Class getComponentType();
        1.1public Constructor getConstructor(Class[] parameterTypes) throws 
NoSuchMethodException, SecurityException;
        1.1public Constructor[] getConstructors() throws SecurityException;
        1.1public Class[] getDeclaredClasses() throws SecurityException;
        1.1public Constructor getDeclaredConstructor(Class[] parameterTypes) throws 
NoSuchMethodException, SecurityException;
        1.1public Constructor[] getDeclaredConstructors() throws SecurityException;
        1.1public Field getDeclaredField(String name) throws NoSuchFieldException, 
SecurityException;
        1.1public Field[] getDeclaredFields() throws SecurityException;
        1.1public Method getDeclaredMethod(String name, Class[] parameterTypes)
throws NoSuchMethodException, SecurityException;
        1.1public Method[] getDeclaredMethods() throws SecurityException;
        1.1public Class getDeclaringClass();
        1.1public Field getField(String name) throws NoSuchFieldException,
SecurityException;
        1.1public Field[] getFields() throws SecurityException;
            public native Class[] getInterfaces();
        1.1public Method getMethod(String name, Class[] parameterTypes) throws
NoSuchMethodException, SecurityException;
        1.1public Method[] getMethods() throws SecurityException;
        1.1public native int getModifiers();
            public native String getName();
        1.1public URL getResource(String name);
        1.1public InputStream getResourceAsStream(String name);
        1.1public native Object[] getSigners();
            public native Class getSuperclass();
        1.1public native boolean isArray();
        1.1public native boolean isAssignableFrom(Class cls);
        1.1public native boolean isInstance(Object obj);
            public native boolean isInterface();
        1.1public native boolean isPrimitive();
            public native Object newInstance() throws InstantiationException,
IllegalAccessException;
            public String toString();  // Overrides Object
}
```

Passed To:

Array.newInstance(), BeanDescriptor(), Beans.getInstanceOf(), Beans.isInstanceOf(), Class.getConstructor(), Class.getDeclaredConstructor(), Class.getDeclaredMethod(), Class.getMethod(), Class.isAssignableFrom(), ClassLoader.resolveClass(), ClassLoader.setSigners(), Compiler.compileClass(), DataFlavor(), EventSetDescriptor(), IndexedPropertyDescriptor(), Introspector.getBeanInfo(), ObjectOutputStream.annotateClass(), ObjectStreamClass.lookup(), PropertyDescriptor(), PropertyDescriptor.setPropertyEditorClass(), PropertyEditorManager.findEditor(), PropertyEditorManager.registerEditor(), SecurityManager.checkMemberAccess()

Returned By:

BeanDescriptor.getBeanClass(), BeanDescriptor.getCustomizerClass(), Class.forName(), Class.getClasses(), Class.getComponentType(), Class.getDeclaredClasses(), Class.getDeclaringClass(), Class.getInterfaces(), Class.getSuperclass(), ClassLoader.defineClass(), ClassLoader.findLoadedClass(), ClassLoader.findSystemClass(), ClassLoader.loadClass(), Constructor.getDeclaringClass(), Constructor.getExceptionTypes(), Constructor.getParameterTypes(), DataFlavor.getRepresentationClass(), EventSetDescriptor.getListenerType(), Field.getDeclaringClass(), Field.getType(), IndexedPropertyDescriptor.getIndexedPropertyType(), Member.getDeclaringClass(), Method.getDeclaringClass(), Method.getExceptionTypes(), Method.getParameterTypes(), Method.getReturnType(), Object.getClass(), ObjectInputStream.resolveClass(), ObjectStreamClass.forClass(), PropertyDescriptor.getPropertyEditorClass(), PropertyDescriptor.getPropertyType(), SecurityManager.currentLoadedClass(), SecurityManager.getClassContext()

Type Of:

Boolean.TYPE, Byte.TYPE, Character.TYPE, Double.TYPE, Float.TYPE, Integer.TYPE, Long.TYPE, Short.TYPE, Void.TYPE

PREVIOUS java.lang.Character (JDK 1.0)

HOME BOOK INDEX

NEXT java.lang.ClassCastException (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

23.2 java.beans.BeanInfo (JDK 1.1)

The `BeanInfo` interface defines the methods that a class must implement in order to export information about a Java bean. The `Introspector` class knows how to obtain all the basic information required about a bean. A bean that wants to be more "programmer-friendly" may provide a class that implements this interface, however, in order to provide additional information about itself (such as an icon and description strings for each of its properties, events, and methods). Note that a bean developer defines a class that implements the methods of this interface. Typically only builder applications and similar tools actually invoke the methods defined here.

The methods `getBeanDescriptor()`, `getEventSetDescriptors()`, `getPropertyDescriptors()`, and `getMethodDescriptors()` should return appropriate descriptor objects for the bean, or `null`, if the bean does not provide explicit bean, event set, property, or method descriptor objects. The `getDefaultEventIndex()` and `getDefaultPropertyIndex()` methods return values that specify the "default" event and property--i.e., that are most likely to be of interest to a programmer using the bean. These methods should return -1 if there are no defaults.

The `getIcon()` method should return an image object suitable for representing the bean in a palette or menu of available beans. The argument passed to this method is one of the four constants defined by the class; it specifies the type and size of icon requested. If the requested icon cannot be provided, `getIcon()` should return `null`.

A `BeanInfo` class is allowed to return `null` or -1 if it cannot provide the requested information. In this case, the `Introspector` class provides basic values for the omitted information from its own introspection of the bean. See `SimpleBeanInfo` for a trivial implementation of this interface, suitable for convenient subclassing.

```
public abstract interface BeanInfo {
    // Constants
            public static final int ICON_COLOR_16x16;
            public static final int ICON_COLOR_32x32;
            public static final int ICON_MONO_16x16;
            public static final int ICON_MONO_32x32;
    // Public Instance Methods
```

```
            public abstract BeanInfo[] getAdditionalBeanInfo();
            public abstract BeanDescriptor getBeanDescriptor();
            public abstract int getDefaultEventIndex();
            public abstract int getDefaultPropertyIndex();
            public abstract EventSetDescriptor[] getEventSetDescriptors();
            public abstract Image getIcon(int iconKind);
            public abstract MethodDescriptor[] getMethodDescriptors();
            public abstract PropertyDescriptor[] getPropertyDescriptors();
}
```

Implemented By:

SimpleBeanInfo

Returned By:

BeanInfo.getAdditionalBeanInfo(), Introspector.getBeanInfo(), SimpleBeanInfo.getAdditionalBeanInfo()

PREVIOUS | HOME | NEXT
java.beans.BeanDescriptor (JDK 1.1) | BOOK INDEX | java.beans.Beans (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

23.20 java.beans.SimpleBeanInfo (JDK 1.1)

The `SimpleBeanInfo` class is a trivial implementation of the `BeanInfo` interface. The methods of this class all return `null` or `-1`, indicating that no bean information is available. To use this class, you need only to override the method or methods that return the particular type of bean information you want to provide.

In addition, `SimpleBeanInfo` provides a convenience method, `loadImage()`, that takes a resource name as an argument and returns an `Image object`. This method is useful when defining the `getIcon()` method.

```
public class SimpleBeanInfo extends Object implements BeanInfo {
    // Default Constructor: public SimpleBeanInfo()
    // Public Instance Methods
            public BeanInfo[] getAdditionalBeanInfo();  // From BeanInfo
            public BeanDescriptor getBeanDescriptor();  // From BeanInfo
            public int getDefaultEventIndex();  // From BeanInfo
            public int getDefaultPropertyIndex();  // From BeanInfo
            public EventSetDescriptor[] getEventSetDescriptors();  // From BeanInfo
            public Image getIcon(int iconKind);  // From BeanInfo
            public MethodDescriptor[] getMethodDescriptors();  // From BeanInfo
            public PropertyDescriptor[] getPropertyDescriptors();  // From BeanInfo
            public Image loadImage(String resourceName);
}
```

23.5 java.beans.EventSetDescriptor (JDK 1.1)

An EventSetDescriptor object is a type of FeatureDescriptor that describes a single set of events supported by a Java bean. A "set" of events corresponds to the one or more methods supported by a single EventListener interface. The BeanInfo class for a Java bean optionally creates EventSetDescriptor objects to describe the event sets the bean supports. Typically, only application builders and similar tools use the get and is methods of EventSetDescriptor objects to obtain the event-set description information.

To create an EventSetDescriptor object, you must specify the class of the bean that supports the event set, the base name of the event set, the class of the EventListener interface that corresponds to the event set, and the methods within this interface that are invoked when particular events within the set occur. Optionally, you may also specify the methods of the bean class that are used to add and remove EventListener objects. The various constructors allow you to specify methods by name, as java.lang.reflect.Method objects, or as MethodDescriptor objects.

Once you have created an EventSetDescriptor, you can use setUnicast() to specify whether it represents a unicast event and setInDefaultEventSet() to specify whether the event set should be treated as the default event set by builder applications. The methods of the FeatureDescriptor superclass allow additional information about the property to be specified.

```
public class EventSetDescriptor extends FeatureDescriptor {
    // Public Constructors
            public EventSetDescriptor(Class sourceClass, String eventSetName, Class 
listenerType, 
            public EventSetDescriptor'u'String listenerMethodName) throws 
IntrospectionException;
            public EventSetDescriptor(Class sourceClass, String eventSetName, Class 
listenerType, 
            public EventSetDescriptor'u'String[] listenerMethodNames, String 
addListenerMethodName 
            public EventSetDescriptor'u'String removeListenerMethodName) throws 
IntrospectionException;
            public EventSetDescriptor(String eventSetName, Class listenerType, 
Method[] listenerMethods, 
            public EventSetDescriptor'u'Method addListenerMethod, Method 
removeListenerMethod) 
            public EventSetDescriptor'u'throws IntrospectionException;
            public EventSetDescriptor(String eventSetName, Class listenerType, 
            public EventSetDescriptor'u'MethodDescriptor[] listenerMethodDescriptors, 
Method addListenerMethod, 
            public EventSetDescriptor'u'Method removeListenerMethod) throws 
IntrospectionException;
```

```
    // Public Instance Methods
            public Method getAddListenerMethod();
            public MethodDescriptor[] getListenerMethodDescriptors();
            public Method[] getListenerMethods();
            public Class getListenerType();
            public Method getRemoveListenerMethod();
            public boolean isInDefaultEventSet();
            public boolean isUnicast();
            public void setInDefaultEventSet(boolean inDefaultEventSet);
            public void setUnicast(boolean unicast);
}
```

Hierarchy:

Object->FeatureDescriptor->EventSetDescriptor

Returned By:

BeanInfo.getEventSetDescriptors(), SimpleBeanInfo.getEventSetDescriptors()

PREVIOUS | HOME | NEXT
java.beans.Customizer (JDK 1.1) | BOOK INDEX | java.beans.FeatureDescriptor (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

28.5 java.net.DatagramPacket (JDK 1.0)

This class implements a "packet" of data that may be sent or received over the network through a `DatagramSocket`. One of the `DatagramPacket` constructors specifies an array of binary data to be sent with its destination address and port. A packet created with this constructor may then be sent with the `send()` method of a `DatagramSocket`. The other `DatagramPacket` constructor specifies an array of bytes into which data should be received. The `receive()` method of `DatagramSocket` waits for data and stores it in a `DatagramPacket` created in this way. The contents and sender of a received packet may be queried with the `DatagramPacket` instance methods.

```
public final class DatagramPacket extends Object {
    // Public Constructors
            public DatagramPacket(byte[] ibuf, int ilength);
            public DatagramPacket(byte[] ibuf, int ilength, InetAddress iaddr, int 
iport);
    // Public Instance Methods
            public synchronized InetAddress getAddress();
            public synchronized byte[] getData();
            public synchronized int getLength();
            public synchronized int getPort();
        1.1public synchronized void setAddress(InetAddress iaddr);
        1.1public synchronized void setData(byte[] ibuf);
        1.1public synchronized void setLength(int ilength);
        1.1public synchronized void setPort(int iport);
}
```

Passed To:

DatagramSocket.receive(), DatagramSocket.send(), DatagramSocketImpl.receive(), DatagramSocketImpl.send(), MulticastSocket.send()

21.4 java.awt.image.DirectColorModel (JDK 1.0)

This class is a `ColorModel` that extracts the red, green, blue, and alpha values directly from the bits of the pixel. The arguments to the constructor methods specify the number of significant bits in the pixel and the mask used to extract each of the color components from the pixel. The default RGB color model is a `DirectColorModel`.

You should not need to instantiate any kind of `ColorModel` object unless you are processing image data that does not use the standard RGB color format.

```
public class DirectColorModel extends ColorModel {
    // Public Constructors
            public DirectColorModel(int bits, int rmask, int gmask, int bmask);
            public DirectColorModel(int bits, int rmask, int gmask, int bmask, int 
amask);
    // Public Instance Methods
            public final int getAlpha(int pixel);  // Defines ColorModel
            public final int getAlphaMask();
            public final int getBlue(int pixel);  // Defines ColorModel
            public final int getBlueMask();
            public final int getGreen(int pixel);  // Defines ColorModel
            public final int getGreenMask();
            public final int getRGB(int pixel);  // Overrides ColorModel
            public final int getRed(int pixel);  // Defines ColorModel
            public final int getRedMask();
}
```

Hierarchy:

```
Object->ColorModel->DirectColorModel
```

21.10 java.awt.image.IndexColorModel (JDK 1.0)

This class is a `ColorModel` that determines the red, green, blue, and alpha values for a pixel by using the pixel value as an index into colormap arrays. If no array of alpha values is specified, all pixels are considered fully opaque, except for one optionally specified reserved value that is fully transparent. This color model is useful when working with image data that is defined in terms of a color map.

You should not need to instantiate any kind of `ColorModel` object unless you are processing image data that does not use the standard RGB color format.

```
public class IndexColorModel extends ColorModel {
    // Public Constructors
            public IndexColorModel(int bits, int size, byte[] r, byte[] g, byte[] b);
            public IndexColorModel(int bits, int size, byte[] r, byte[] g, byte[] b, 
int trans);
            public IndexColorModel(int bits, int size, byte[] r, byte[] g, byte[] b, 
byte[] a);
            public IndexColorModel(int bits, int size, byte[] cmap, int start, 
boolean hasalpha);
            public IndexColorModel(int bits, int size, byte[] cmap, int start, 
boolean hasalpha, int trans);
    // Public Instance Methods
            public final int getAlpha(int pixel);  // Defines ColorModel
            public final void getAlphas(byte[] a);
            public final int getBlue(int pixel);  // Defines ColorModel
            public final void getBlues(byte[] b);
            public final int getGreen(int pixel);  // Defines ColorModel
            public final void getGreens(byte[] g);
            public final int getMapSize();
            public final int getRGB(int pixel);  // Overrides ColorModel
            public final int getRed(int pixel);  // Defines ColorModel
            public final void getReds(byte[] r);
            public final int getTransparentPixel();
}
```

Hierarchy:

`Object->ColorModel->IndexColorModel`

Passed To:

```
RGBImageFilter.filterIndexColorModel()
```

Returned By:

```
RGBImageFilter.filterIndexColorModel()
```

29.11 java.text.FieldPosition (JDK 1.1)

FieldPosition objects are optionally passed to the format() methods of the Format class and its subclasses to return additional information about the formatting that has been performed. The getBeginIndex() and getEndIndex() methods of this class are used to return the starting and ending character positions of some field of the formatted string. The integer value passed to the FieldPosition() constructor specifies what field of the returned string should have its bounds returned. The NumberFormat and DateFormat classes define various constants (which end with the string _FIELD) that can be used here. Typically, this bounds information is useful for aligning formatted strings in columns--for example, aligning the decimal points in a column of numbers.

```
public class FieldPosition extends Object {
    // Public Constructor
            public FieldPosition(int field);
    // Public Instance Methods
            public int getBeginIndex();
            public int getEndIndex();
            public int getField();
}
```

Passed To:

ChoiceFormat.format(), DateFormat.format(), DecimalFormat.format(), Format.format(), MessageFormat.format(), NumberFormat.format(), SimpleDateFormat.format()

18.55 java.awt.Shape (JDK 1.1)

This interface encapsulates some very generic information about geometric shapes. All `Shape` objects must have a bounding box--i.e., a rectangle that completely encloses the represented shape.

When the forthcoming Java2D API is integrated with the AWT, this interface may be changed. For this reason, Java 1.1 applications can use this interface, but should not implement it in their own classes.

```
public abstract interface Shape {
    // Public Instance Methods
            public abstract Rectangle getBounds();
}
```

Implemented By:

`Polygon, Rectangle`

Passed To:

`Graphics.setClip()`

Returned By:

`Graphics.getClip()`

30.20 java.util.ResourceBundle (JDK 1.1)

This abstract class allows subclasses to define sets of localized resources which can then be dynamically loaded as needed by internationalized programs. Such resources may include user-visible text and images that appear in an application, and even more complex things such as `Menu` objects.

Use `getBundle()` to load a `ResourceBundle` subclass that is appropriate for the default or the specified locale. Use `getObject()`, `getString()`, and `getStringArray()` to look up a named resource in a bundle. To define a bundle, provide implementations of `handleGetObject()` and `getKeys()`. It is often easier, however, to subclass `ListResourceBundle`, or to provide a property file that is used by `PropertyResourceBundle`. The name of any localized `ResourceBundle` class you define should include the locale language code, and, optionally, the locale country code.

```
public abstract class ResourceBundle extends Object {
    // Default Constructor: public ResourceBundle()
    // Protected Instance Variables
            protected ResourceBundle parent;
    // Class Methods
            public static final ResourceBundle getBundle(String baseName) throws 
MissingResourceException;
            public static final ResourceBundle getBundle(String baseName, Locale 
locale);
    // Public Instance Methods
            public abstract Enumeration getKeys();
            public final Object getObject(String key) throws 
MissingResourceException;
            public final String getString(String key) throws 
MissingResourceException;
            public final String[] getStringArray(String key) throws 
MissingResourceException;
    // Protected Instance Methods
            protected abstract Object handleGetObject(String key) throws 
MissingResourceException;
            protected void setParent(ResourceBundle parent);
}
```

Extended By:

ListResourceBundle, PropertyResourceBundle

Passed To:

ResourceBundle.setParent()

Returned By:

ResourceBundle.getBundle()

Type Of:

ResourceBundle.parent

PREVIOUS	HOME	NEXT
java.util.Random (JDK 1.0)	BOOK INDEX	java.util.SimpleTimeZone (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

22.24 java.awt.peer.TextComponentPeer (JDK 1.0)

```
public abstract interface TextComponentPeer extends ComponentPeer {
    // Public Instance Methods
        1.1  public abstract int getCaretPosition();
            public abstract int getSelectionEnd();
            public abstract int getSelectionStart();
            public abstract String getText();
            public abstract void select(int selStart, int selEnd);
        1.1  public abstract void setCaretPosition(int pos);
            public abstract void setEditable(boolean editable);
            public abstract void setText(String l);
}
```

Extended By:

TextAreaPeer, TextFieldPeer

31.2 java.util.zip.CheckedInputStream (JDK 1.1)

This class is a subclass of `java.io.FilterInputStream`; it allows a stream to be read and a checksum computed on its contents at the same time. This is useful when you want to check the integrity of a stream of data against a published checksum value.

To create a `CheckedInputStream`, you must specify both the stream that it should read and also a `Checksum` object, such as `CRC32`, that implements the particular checksum algorithm you desire. The `read()` and `skip()` methods are the same as those of other input streams. As bytes are read, they are incorporated into the checksum that is being computed. Note that the `getChecksum()` method does not return the checksum value itself, but rather the `Checksum` object. You must call the `getValue()` method of this object to obtain the checksum value.

```
public class CheckedInputStream extends FilterInputStream {
    // Public Constructor
            public CheckedInputStream(InputStream in, Checksum cksum);
    // Public Instance Methods
            public Checksum getChecksum();
            public int read() throws IOException;  // Overrides FilterInputStream
            public int read(byte[] buf, int off, int len) throws IOException;  //
Overrides FilterInputStream
            public long skip(long n) throws IOException;  // Overrides
FilterInputStream
}
```

Hierarchy:

Object->InputStream->FilterInputStream->CheckedInputStream

31.3 java.util.zip.CheckedOutputStream (JDK 1.1)

This class is a subclass of `java.io.FilterOutputStream` that allows data to be written to a stream and a checksum computed on that data at the same time.

To create a `CheckedOutputStream` you must specify the output stream that it is to write its data to, and you must also specify a `Checksum` object, such as an instance of `Adler32`, that implements the particular checksum algorithm you desire. The `write()` methods are similar to those of other `OutputStream` classes. The `getChecksum()` method returns the `Checksum` object. Note that you must call `getValue()` on this object in order to obtain the actual checksum value.

```
public class CheckedOutputStream extends FilterOutputStream {
    // Public Constructor
            public CheckedOutputStream(OutputStream out, Checksum cksum);
    // Public Instance Methods
            public Checksum getChecksum();
            public void write(int b) throws IOException;  // Overrides
FilterOutputStream
            public void write(byte[] b, int off, int len) throws IOException;  //
Overrides FilterOutputStream
}
```

Hierarchy:

Object->OutputStream->FilterOutputStream->CheckedOutputStream

30.13 java.util.MissingResourceException (JDK 1.1)

This exception signals that no `ResourceBundle` could be located for the desired locale, or that a named resource could not be found within a given `ResourceBundle`.

`getClassName()` returns the name of the `ResourceBundle` class in question, and `getKey()` returns the name of the resource that could not be located.

```
public class MissingResourceException extends RuntimeException {
    // Public Constructor
            public MissingResourceException(String s, String className, String key);
    // Public Instance Methods
            public String getClassName();
            public String getKey();
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->RuntimeException->MissingResourceException

Thrown By:

Locale.getISO3Country(), Locale.getISO3Language(), ResourceBundle.getBundle(), ResourceBundle.getObject(), ResourceBundle.getString(), ResourceBundle.getStringArray(), ResourceBundle.handleGetObject()

20.21 java.awt.event.MouseEvent (JDK 1.1)

An event of this type indicates that the user has moved the mouse or pressed one of the mouse buttons.

Call `getID()` to determine the specific type of mouse event that has occurred. This method returns one of the following seven constants, which corresponds to a method in either the `MouseListener` or `MouseMotionListener` interface.

`MOUSE_PRESSED`

The user has pressed a mouse button.

`MOUSE_RELEASED`

The user has released a mouse button.

`MOUSE_CLICKED`

The user has pressed and released a mouse button without any intervening mouse drag.

`MOUSE_DRAGGED`

The user has moved the mouse while holding a button down

`MOUSE_MOVED`

The user has moved the mouse without holding any buttons down.

`MOUSE_ENTERED`

The mouse pointer has entered the component. `MOUSE_EXITED`

The mouse pointer has left the component

Use `getX()` and `getY()` or `getPoint()` to obtain the coordinates of the mouse event. Use `translatePoint()` to modify these coordinates by a specified amount.

Use `getModifiers()` and other methods and constants inherited from `InputEvent` to determine the mouse button or keyboard modifiers that were down when the event occurred. See `InputEvent` for details. Note that mouse button modifiers are not reported for `MOUSE_RELEASED` events, since, technically, the mouse button in question is no longer pressed. This can be surprising.

Use `getComponent()`, inherited from `ComponentEvent`, to determine over which component the event occurred.

For mouse events of type `MOUSE_CLICKED`, `MOUSE_PRESSED`, or `MOUSE_RELEASED`, call `getClickCount()` to determine how many consecutive clicks have occurred.

If you are using popup menus, use `isPopupTrigger()` to test whether the current event represents the standard platform-dependent popup menu trigger event.

```
public class MouseEvent extends InputEvent {
    // Public Constructor
            public MouseEvent(Component source, int id, long when, int modifiers, int
x, int y,
            public MouseEvent'u'int clickCount, boolean popupTrigger);
    // Constants
            public static final int MOUSE_CLICKED;
            public static final int MOUSE_DRAGGED;
            public static final int MOUSE_ENTERED;
            public static final int MOUSE_EXITED;
            public static final int MOUSE_FIRST;
            public static final int MOUSE_LAST;
            public static final int MOUSE_MOVED;
            public static final int MOUSE_PRESSED;
            public static final int MOUSE_RELEASED;
    // Public Instance Methods
            public int getClickCount();
            public Point getPoint();
            public int getX();
            public int getY();
            public boolean isPopupTrigger();
            public String paramString();  // Overrides ComponentEvent
            public synchronized void translatePoint(int x, int y);
}
```

Hierarchy:

```
Object->EventObject(Serializable)->AWTEvent->ComponentEvent->InputEvent->MouseEvent
```

Passed To:

```
AWTEventMulticaster.mouseClicked(), AWTEventMulticaster.mouseDragged(),
AWTEventMulticaster.mouseEntered(), AWTEventMulticaster.mouseExited(),
AWTEventMulticaster.mouseMoved(), AWTEventMulticaster.mousePressed(),
AWTEventMulticaster.mouseReleased(), Component.processMouseEvent(),
Component.processMouseMotionEvent(), MouseAdapter.mouseClicked(),
MouseAdapter.mouseEntered(), MouseAdapter.mouseExited(), MouseAdapter.mousePressed(),
MouseAdapter.mouseReleased(), MouseListener.mouseClicked(),
MouseListener.mouseEntered(), MouseListener.mouseExited(),
MouseListener.mousePressed(), MouseListener.mouseReleased(),
MouseMotionAdapter.mouseDragged(), MouseMotionAdapter.mouseMoved(),
```

`MouseMotionListener.mouseDragged(), MouseMotionListener.mouseMoved()`

28.3 java.net.ContentHandler (JDK 1.0)

This abstract class defines a method that reads data from a `URLConnection` and returns an object representing that data. Each subclass that implements this method is responsible for handling a different type of content (i.e., a different MIME type). Applications never create `ContentHandler` objects directly--they are created, when necessary, by the registered `ContentHandlerFactory` object. Applications should also never call `ContentHandler` methods directly--they should call `URL.getContent()` or `URLConnection.getContent()` instead. You only need to subclass `ContentHandler` if you are writing a Web browser or similar application that needs to parse and understand some new content type.

```
public abstract class ContentHandler extends Object {
    // Default Constructor: public ContentHandler()
    // Public Instance Methods
            public abstract Object getContent(URLConnection urlc) throws IOException;
}
```

Returned By:

ContentHandlerFactory.createContentHandler()

30.11 java.util.ListResourceBundle (JDK 1.1)

This abstract class provides a simple way to define a `ResourceBundle`. You may find it easier to subclass `ListResourceBundle` than to subclass `ResourceBundle` directly.

`ListResourceBundle` provides implementations for the abstract `handleGetObject()` and `getKeys()` methods defined by `ResourceBundle`, and it adds its own abstract `getContents()` method that your subclasses must override. `getContents()` returns an `Object[][]`--an array of arrays of objects. This array can have any number of elements. Each element of this array must itself be an array with two elements: the first element of each subarray should be a `String` that specifies the name of a resource, and the corresponding second element should be the value of that resource--this value can be an `Object` of any desired type.

See also `ResourceBundle` and `PropertyResourceBundle`.

```
public abstract class ListResourceBundle extends ResourceBundle {
    // Default Constructor: public ListResourceBundle()
    // Public Instance Methods
            public Enumeration getKeys();  // Defines ResourceBundle
            public final Object handleGetObject(String key);  // Defines 
ResourceBundle
    // Protected Instance Methods
            protected abstract Object[][] getContents();
}
```

Hierarchy:

Object->ResourceBundle->ListResourceBundle

28.8 java.net.FileNameMap (JDK 1.1)

This interface defines a single method that is called to obtain the MIME type of a file based on the name of a file. The `fileNameMap` field of the `URLConnection` class refers to an object that implements this interface. The filename-to-file-type map it implements is used by the static `URLConnection.guessContentTypeFromName()` method.

```
public abstract interface FileNameMap {
    // Public Instance Methods
            public abstract String getContentTypeFor(String fileName);
}
```

Type Of:

URLConnection.fileNameMap

18.11 java.awt.CheckboxGroup (JDK 1.0)

A `CheckboxGroup` object enforces mutual exclusion (also known as "radio button" behavior) among any number of `Checkbox` buttons. A `Checkbox` component can specify a `CheckboxGroup` object when created or with `Checkbox.setCheckboxGroup()`. If a `Checkbox` with a given `CheckboxGroup` object is selected, then the `CheckboxGroup` ensures that the previously selected `Checkbox` becomes unselected.

```
public class CheckboxGroup extends Object implements Serializable {
    // Public Constructor
            public CheckboxGroup();
    // Public Instance Methods
        #   public Checkbox getCurrent();
        1.1 public Checkbox getSelectedCheckbox();
        #   public synchronized void setCurrent(Checkbox box);
        1.1 public synchronized void setSelectedCheckbox(Checkbox box);
            public String toString();  // Overrides Object
}
```

Passed To:

`Checkbox(), Checkbox.setCheckboxGroup(), CheckboxPeer.setCheckboxGroup()`

Returned By:

`Checkbox.getCheckboxGroup()`

java.awt.Checkbox (JDK 1.0) BOOK INDEX java.awt.CheckboxMenuItem (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

29.15 java.text.ParseException (JDK 1.1)

This exception signals that a string had an incorrect format and could not be parsed. It is typically thrown by the `parse()` or `parseObject()` methods of `Format` and its subclasses, but is also thrown by certain methods in the `java.text` package that are passed patterns or other rules in string form. The `getErrorOffset()` method of this class returns the character position at which the parsing error occurred in the offending string.

```
public class ParseException extends Exception {
    // Public Constructor
            public ParseException(String s, int errorOffset);
    // Public Instance Methods
            public int getErrorOffset();
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->ParseException

Thrown By:

DateFormat.parse(), Format.parseObject(), MessageFormat.parse(), NumberFormat.parse(), RuleBasedCollator()

25.20 java.lang.ExceptionInInitializerError (JDK 1.1)

This error is thrown by the Java Virtual Machine when an exception occurs in the static initializer of a class. You can use the `getException()` method to obtain the `Throwable` object that was thrown from the initializer.

```
public class ExceptionInInitializerError extends LinkageError {
    // Public Constructors
            public ExceptionInInitializerError();
            public ExceptionInInitializerError(Throwable thrown);
            public ExceptionInInitializerError(String s);
    // Public Instance Methods
            public Throwable getException();
}
```

Hierarchy:

Object->Throwable(Serializable)->Error->LinkageError->ExceptionInInitializerError

18.43 java.awt.MenuContainer (JDK 1.0)

This interface defines the methods necessary for `MenuContainer` types such as the `Menu`, `Frame`, and `MenuBar` objects. Unless you implement new menu-like components, you never need to use it.

```
public abstract interface MenuContainer {
    // Public Instance Methods
            public abstract Font getFont();
            public abstract boolean postEvent(Event evt);
            public abstract void remove(MenuComponent comp);
}
```

Implemented By:

`Component, Frame, Menu, MenuBar`

Returned By:

`MenuComponent.getParent()`

29.16 java.text.ParsePosition (JDK 1.1)

`ParsePosition` objects are passed to the `parse()` and `parseObject()` methods of `Format` and its subclasses. The `ParsePosition` class represents the position in a string at which parsing should begin or at which parsing stopped. Before calling a `parse()` method, you can specify the starting position of parsing by passing the desired index to the `ParsePosition()` constructor, or by calling the `setIndex()` of an existing `ParsePosition` object. When `parse()` returns, you can determine where parsing ended by calling `getIndex()`. When parsing multiple objects or values from a string, a single `ParsePosition` object can be used sequentially.

```
public class ParsePosition extends Object {
    // Public Constructor
            public ParsePosition(int index);
    // Public Instance Methods
            public int getIndex();
            public void setIndex(int index);
}
```

Passed To:

ChoiceFormat.parse(), DateFormat.parse(), DateFormat.parseObject(), DecimalFormat.parse(), Format.parseObject(), MessageFormat.parse(), MessageFormat.parseObject(), NumberFormat.parse(), NumberFormat.parseObject(), SimpleDateFormat.parse()

23.7 java.beans.IndexedPropertyDescriptor (JDK 1.1)

An `IndexedPropertyDescriptor` object is a type of `PropertyDescriptor` that describes a Java bean property that is (or behaves like) an array. The `BeanInfo` class for a Java bean optionally creates and initializes `IndexedPropertyDescriptor` objects to describe the indexed properties that the bean supports. Typically, only application builders and similar tools use the descriptor objects to obtain indexed property description information.

You create an `IndexedPropertyDescriptor` by specifying the name of the indexed property and the `Class` object for the bean. If you have not followed the standard "design patterns" for accessor method naming, you may also specify the accessor methods for the property, either as method names or as `java.lang.reflect.Method` objects. Once you have created an `IndexedPropertyDescriptor` object, you can use the methods of `PropertyDescriptor` and `FeatureDescriptor` to provide additional information about the indexed property.

```
public class IndexedPropertyDescriptor extends PropertyDescriptor {
    // Public Constructors
            public IndexedPropertyDescriptor(String propertyName, Class beanClass)
throws IntrospectionException;
            public IndexedPropertyDescriptor(String propertyName, Class beanClass,
String getterName,
            public IndexedPropertyDescriptor'u'String setterName, String
indexedGetterName,
            public IndexedPropertyDescriptor'u'String indexedSetterName) throws
IntrospectionException;
            public IndexedPropertyDescriptor(String propertyName, Method getter,
Method setter, Method indexedGetter,
            public IndexedPropertyDescriptor'u'Method indexedSetter) throws
IntrospectionException;
    // Public Instance Methods
            public Class getIndexedPropertyType();
            public Method getIndexedReadMethod();
            public Method getIndexedWriteMethod();
}
```

Hierarchy:

Object->FeatureDescriptor->PropertyDescriptor->IndexedPropertyDescriptor

PREVIOUS | HOME | NEXT

java.beans.FeatureDescriptor (JDK 1.1) | BOOK INDEX | java.beans.IntrospectionException (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

23.3 java.beans.Beans (JDK 1.1)

The `Beans` class is never meant to be instantiated; its static methods provide miscellaneous JavaBeans features.

The `instantiate()` method creates an instance of a bean. The specified bean name represents either a serialized bean file or a bean class file; it is interpreted relative to the specified `ClassLoader` object.

The `setDesignTime()` and `isDesignTime()` methods are used to set and query a flag that indicates whether beans are being used in an application builder environment. Similarly, `setGuiAvailable()` and `isGuiAvailable()` set and query a flag that indicates whether the Java Virtual Machine is running in an environment in which a GUI is available. (Note that untrusted applet code cannot call `setDesignTime()` or `setGuiAvailable()`.)

The `isInstanceOf()` method is a replacement for the Java `instanceof` operator for use with beans. Currently, it behaves just like `instanceof`, but in the future it may work with beans that consist of a set of Java objects, each of which provides a different "view" of a bean. Similarly, the `getInstanceOf()` method is a replacement for the Java cast operator. It converts a bean to a superclass or interface type. Currently, it behaves just like a cast, but you should use it for future compatibility with multiclass beans.

```
public class Beans extends Object {
    // Default Constructor: public Beans()
    // Class Methods
            public static Object getInstanceOf(Object bean, Class targetType);
            public static Object instantiate(ClassLoader cls, String beanName)
            public static Object instantiate'u'throws IOException,
ClassNotFoundException;
            public static boolean isDesignTime();
            public static boolean isGuiAvailable();
            public static boolean isInstanceOf(Object bean, Class targetType);
            public static void setDesignTime(boolean isDesignTime) throws
SecurityException;
            public static void setGuiAvailable(boolean isGuiAvailable) throws
SecurityException;
}
```

28.12 java.net.MulticastSocket (JDK 1.1)

This subclass of `DatagramSocket` is used to send and receive multicast UDP packets. It extends `DatagramSocket` by adding `joinGroup()` and `leaveGroup()` methods to join and leave multicast groups. The IP address specified to these methods should be a valid multicast address in the range of 224.0.0.1 to 239.255.255.255. Note that you do not have to join a group to send a packet to a multicast address, but you must join the group to receive packets sent to that address.

`MulticastSocket` defines a variant `send()` method that allows you to specify a time-to-live (TTL) value for the packet you send. This value specifies the number of network "hops" the packet may travel before it expires. You can also set a default TTL for all packets sent though a `MulticastSocket` with `setTTL()`.

Note that untrusted applets are not allowed to use multicast sockets.

```
public class MulticastSocket extends DatagramSocket {
    // Public Constructors
            public MulticastSocket() throws IOException;
            public MulticastSocket(int port) throws IOException;
    // Public Instance Methods
            public InetAddress getInterface() throws SocketException;
            public byte getTTL() throws IOException;
            public void joinGroup(InetAddress mcastaddr) throws IOException;
            public void leaveGroup(InetAddress mcastaddr) throws IOException;
            public synchronized void send(DatagramPacket p, byte ttl) throws
IOException;
            public void setInterface(InetAddress inf) throws SocketException;
            public void setTTL(byte ttl) throws IOException;
}
```

Hierarchy:

Object->DatagramSocket->MulticastSocket

java.net.MalformedURLException (JDK 1.0) | BOOK INDEX | java.net.NoRouteToHostException (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

30.18 java.util.PropertyResourceBundle (JDK 1.1)

This class is a concrete subclass of `ResourceBundle`. It reads a `Properties` file from a specified `InputStream` and implements the `ResourceBundle` API for looking up named resources from the resulting `Properties` object.

A `Properties` file contains lines of the form:

`name=value`

Each such line defines a named property with the specified `String` value.

Although you can instantiate a `PropertyResourceBundle` yourself, it is more common to simply define a `Properties` file, and then allow `ResourceBundle.getBundle()` to look up that file and return the necessary `PropertyResourceBundle` object.

See also `Properties` and `ResourceBundle`.

```
public class PropertyResourceBundle extends ResourceBundle {
    // Public Constructor
            public PropertyResourceBundle(InputStream stream) throws IOException;
    // Public Instance Methods
            public Enumeration getKeys();  // Defines ResourceBundle
            public Object handleGetObject(String key);  // Defines ResourceBundle
}
```

Hierarchy:

Object->ResourceBundle->PropertyResourceBundle

24.35 java.io.LineNumberReader (JDK 1.1)

This class is a character input stream that keeps track of the number of lines of text that have been read from it. It supports the usual `Reader` methods, and also the `readLine()` method introduced by its superclass. In addition to these methods, you can call `getLineNumber()` to query the number of lines set so far. You can also call `setLineNumber()` to set the line number for the current line. Subsequent lines are numbered sequentially from this specified starting point.

This class is a character-stream analog to `LineNumberInputStream`, which has been deprecated in Java 1.1.

```
public class LineNumberReader extends BufferedReader {
    // Public Constructors
            public LineNumberReader(Reader in);
            public LineNumberReader(Reader in, int sz);
    // Public Instance Methods
            public int getLineNumber();
            public void mark(int readAheadLimit) throws IOException;  // Overrides 
BufferedReader
            public int read() throws IOException;  // Overrides BufferedReader
            public int read(char[] cbuf, int off, int len) throws IOException;  // 
Overrides BufferedReader
            public String readLine() throws IOException;  // Overrides BufferedReader
            public void reset() throws IOException;  // Overrides BufferedReader
            public void setLineNumber(int lineNumber);
            public long skip(long n) throws IOException;  // Overrides BufferedReader
}
```

Hierarchy:

Object->Reader->BufferedReader->LineNumberReader

23.10 java.beans.MethodDescriptor (JDK 1.1)

A `MethodDescriptor` object is a type of `FeatureDescriptor` that describes a method supported by a Java bean. The `BeanInfo` class for a Java bean optionally creates `MethodDescriptor` objects that describe the methods the bean exports. While a `BeanInfo` class creates and initializes `MethodDescriptor` objects, it is typically only application builders and similar tools that use these objects to obtain information about the methods supported by a bean.

To create a `MethodDescriptor`, you must specify the `java.lang.reflect.Method` object for the method, and optionally specify an array of `ParameterDescriptor` objects that describe the parameters of the method. Once you have created a `MethodDescriptor` object, you can use `FeatureDescriptor` methods to provide additional information about each method.

```
public class MethodDescriptor extends FeatureDescriptor {
    // Public Constructors
            public MethodDescriptor(Method method);
            public MethodDescriptor(Method method, ParameterDescriptor[] 
parameterDescriptors);
    // Public Instance Methods
            public Method getMethod();
            public ParameterDescriptor[] getParameterDescriptors();
}
```

Hierarchy:

Object->FeatureDescriptor->MethodDescriptor

Passed To:

EventSetDescriptor()

Returned By:

BeanInfo.getMethodDescriptors(), EventSetDescriptor.getListenerMethodDescriptors(), SimpleBeanInfo.getMethodDescriptors()

22.23 java.awt.peer.TextAreaPeer (JDK 1.0)

```
public abstract interface TextAreaPeer extends TextComponentPeer {
    // Public Instance Methods
        1.1  public abstract Dimension getMinimumSize(int rows, int columns);
        1.1  public abstract Dimension getPreferredSize(int rows, int columns);
        1.1  public abstract void insert(String text, int pos);
            public abstract void insertText(String txt, int pos);
            public abstract Dimension minimumSize(int rows, int cols);
            public abstract Dimension preferredSize(int rows, int cols);
        1.1  public abstract void replaceRange(String text, int start, int end);
            public abstract void replaceText(String txt, int start, int end);
}
```

Hierarchy:

(TextAreaPeer(TextComponentPeer(ComponentPeer)))

Returned By:

Toolkit.createTextArea()

22.25 java.awt.peer.TextFieldPeer (JDK 1.0)

```
public abstract interface TextFieldPeer extends TextComponentPeer {
    // Public Instance Methods
        1.1  public abstract Dimension getMinimumSize(int columns);
        1.1  public abstract Dimension getPreferredSize(int columns);
            public abstract Dimension minimumSize(int cols);
            public abstract Dimension preferredSize(int cols);
        1.1  public abstract void setEchoChar(char echoChar);
            public abstract void setEchoCharacter(char c);
}
```

Hierarchy:

(TextFieldPeer(TextComponentPeer(ComponentPeer)))

Returned By:

Toolkit.createTextField()

23.12 java.beans.PropertyChangeEvent (JDK 1.1)

The `PropertyChangeEvent` class is a subclass of `java.util.EventObject`. An event of this type is sent to interested `PropertyChangeListener` objects whenever a Java bean changes a "bound" property, or whenever a `PropertyEditor` or `Customizer` changes a property value. A `PropertyChangeEvent` is also sent to registered `VetoableChangeListener` objects when a bean attempts to change the value of a "constrained" property.

When creating a `PropertyChangeEvent`, you normally specify the bean that generated the event, the programmatic (locale-independent) name of the property that changed, and the old and new values of the property. If the values cannot be determined, `null` should be passed instead. If the event is a notification that more than one property value changed, the name should also be `null`.

While Java beans must generate and send `PropertyChangeEvent` objects, it is typically only application builders and similar tools that are interested in receiving them.

```
public class PropertyChangeEvent extends EventObject {
    // Public Constructor
            public PropertyChangeEvent(Object source, String propertyName, Object 
oldValue, Object newValue);
    // Public Instance Methods
            public Object getNewValue();
            public Object getOldValue();
            public Object getPropagationId();
            public String getPropertyName();
            public void setPropagationId(Object propagationId);
}
```

Hierarchy:

Object->EventObject(Serializable)->PropertyChangeEvent

Passed To:

PropertyChangeListener.propertyChange(), PropertyVetoException(), VetoableChangeListener.vetoableChange()

Returned By:

PropertyVetoException.getPropertyChangeEvent()

PREVIOUS | HOME | NEXT
java.beans.ParameterDescriptor (JDK 1.1) | BOOK INDEX | java.beans.PropertyChangeListener (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

18.21 java.awt.EventQueue (JDK 1.1)

This class implements an event queue for AWT events in Java 1.1. When an `EventQueue` is created, a new thread is automatically created and started to remove events from the front of the queue and dispatch them to the appropriate component. It is this thread, created by the `EventQueue`, that notifies event listeners and executes most of the code in a typical GUI-driven application.

An application can create and use its own private `EventQueue`, but all AWT events are placed on and dispatched from a single system `EventQueue`. Use the `getSystemEventQueue()` method of the `Toolkit` class to get the system `EventQueue` object.

`getNextEvent()` removes and returns the event at the front of the queue. It blocks if there are no events in the queue. `peekEvent()` returns the event at the front of the queue without removing it from the queue. Passed an optional `AWTEvent id` field, it returns the first event of the specified type. Finally, `postEvent()` places a new event on the end of the event queue.

Most applications do not need to use the `EventQueue` class at all; they can simply rely on the system to dispatch events automatically.

```
public class EventQueue extends Object {
    // Public Constructor
            public EventQueue();
    // Public Instance Methods
            public synchronized AWTEvent getNextEvent() throws InterruptedException;
            public synchronized AWTEvent peekEvent();
            public synchronized AWTEvent peekEvent(int id);
            public synchronized void postEvent(AWTEvent theEvent);
}
```

Returned By:

`Toolkit.getSystemEventQueue(), Toolkit.getSystemEventQueueImpl()`

18.50 java.awt.PrintGraphics (JDK 1.1)

The `Graphics` object returned by the `getGraphics()` method of `PrintJob` always implements this `PrintGraphics` interface. You can use this fact to distinguish a `Graphics` object that draws to the screen from one that generates hardcopy. This is a useful thing to do in a `paint()` method when you want to generate hardcopy that differs somewhat from what is displayed on-screen.

The `getPrintJob()` method defined by this interface can also be used, of course, to return the `PrintJob` with which the `PrintGraphics` object is associated.

```
public abstract interface PrintGraphics {
    // Public Instance Methods
            public abstract PrintJob getPrintJob();
}
```

23.19 java.beans.PropertyVetoException (JDK 1.1)

The `PropertyVetoException` signals that a `VetoableChangeListener` that received a `PropertyChangeEvent` for a "constrained" property of a bean has vetoed that proposed change.

When this exception is received, the property in question should revert back to its original value, and any `VetoableChangeListener` objects that have already been notified of the property change must be re-notified to indicate that the property has reverted to its old value. The `VetoableChangeSupport` class handles this re-notification automatically and re-throws the `PropertyVetoException` to notify its caller that the change was rejected.

```
public class PropertyVetoException extends Exception {
    // Public Constructor
            public PropertyVetoException(String mess, PropertyChangeEvent evt);
    // Public Instance Methods
            public PropertyChangeEvent getPropertyChangeEvent();
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->PropertyVetoException

Thrown By:

VetoableChangeListener.vetoableChange(), VetoableChangeSupport.fireVetoableChange()

23.15 java.beans.PropertyDescriptor (JDK 1.1)

A `PropertyDescriptor` object is a type of `FeatureDescriptor` that describes a single property of a Java bean. The `BeanInfo` class for a Java bean optionally creates and initializes `PropertyDescriptor` objects to describe the properties that the bean supports. Typically, only application builders and similar tools use the `get` and `is` methods to obtain this property description information.

You create a `PropertyDescriptor` by specifying the name of the property and the `Class` object for the bean. If you have not followed the standard "design patterns" for accessor method naming, you may also specify the accessor methods for the property. Once a `PropertyDescriptor` is created, the `setBound()` and `setConstrained()` methods allow you to specify whether the property is bound and/or constrained. `setPropertyEditorClass()` allows you to specify a specific property editor that should be used to edit the value of this property (this is useful, for example, when the property is an enumerated type with a specific list of supported values). The methods of the `FeatureDescriptor` superclass allow additional information about the property to be specified.

```
public class PropertyDescriptor extends FeatureDescriptor {
    // Public Constructors
            public PropertyDescriptor(String propertyName, Class beanClass) throws
IntrospectionException;
            public PropertyDescriptor(String propertyName, Class beanClass, String
getterName,
            public PropertyDescriptor'u'String setterName) throws
IntrospectionException;
            public PropertyDescriptor(String propertyName, Method getter, Method
setter) throws IntrospectionException;
    // Public Instance Methods
            public Class getPropertyEditorClass();
            public Class getPropertyType();
            public Method getReadMethod();
            public Method getWriteMethod();
            public boolean isBound();
            public boolean isConstrained();
            public void setBound(boolean bound);
            public void setConstrained(boolean constrained);
            public void setPropertyEditorClass(Class propertyEditorClass);
}
```

Hierarchy:

Object->FeatureDescriptor->PropertyDescriptor

Extended By:

IndexedPropertyDescriptor

Returned By:

BeanInfo.getPropertyDescriptors(), SimpleBeanInfo.getPropertyDescriptors()

30.8 java.util.EventObject (JDK 1.1)

`EventObject` serves as a superclass for all events objects used by the Java 1.1 AWT event model and the JavaBeans event model. This class defines a very generic type of event; it is extended by the more specific event classes in the `java.awt`, `java.awt.event`, and `java.beans` packages.

The only common feature shared by all events is a source object, which is the object that in some way "generated" the event. The source object is passed to the `EventObject()` constructor, and is returned by the `getSource()` method.

```
public class EventObject extends Object implements Serializable {
    // Public Constructor
            public EventObject(Object source);
    // Protected Instance Variables
            protected transient Object source;
    // Public Instance Methods
            public Object getSource();
            public String toString();  // Overrides Object
}
```

Extended By:

AWTEvent, PropertyChangeEvent

26.4 java.lang.reflect.InvocationTargetException (JDK 1.1)

An object of this class is thrown by `Method.invoke()` and `Constructor.newInstance()` when an exception is thrown by the method or constructor invoked through those methods. The `InvocationTargetException` class serves as a wrapper around the object that was thrown; that object can be retrieved with the `getTargetException()` method.

```
public class InvocationTargetException extends Exception {
    // Public Constructors
            public InvocationTargetException(Throwable target);
            public InvocationTargetException(Throwable target, String s);
    // Protected Constructor
            protected InvocationTargetException();
    // Public Instance Methods
            public Throwable getTargetException();
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->InvocationTargetException

Thrown By:

Constructor.newInstance(), Method.invoke()

19.4 java.awt.datatransfer.StringSelection (JDK 1.1)

This convenience class implements the `Transferable` and `ClipboardOwner` interfaces in order to make it very easy to transfer `String` values through the AWT data transfer mechanism. It is able to transfer `String` values using either the `DataFlavor.stringFlavor` or `DataFlavor.plainTextFlavor` data flavors.

To create a `StringSelection` object, simply pass the `String` you want to transfer to the `StringSelection()` constructor. You can then make the `StringSelection` available for transfer by passing it to the `setContents()` method of the `Clipboard`. You need never call the methods of `StringSelection` yourself.

```
public class StringSelection extends Object implements Transferable, ClipboardOwner {
    // Public Constructor
            public StringSelection(String data);
    // Public Instance Methods
            public synchronized Object getTransferData(DataFlavor flavor)
            public synchronized Object getTransferData'u'throws
UnsupportedFlavorException, IOException;
            public synchronized Object getTransferData'u'// From Transferable
            public synchronized DataFlavor[] getTransferDataFlavors();  // From
Transferable
            public boolean isDataFlavorSupported(DataFlavor flavor);  // From
Transferable
            public void lostOwnership(Clipboard clipboard, Transferable contents);
// From ClipboardOwner
}
```

19.5 java.awt.datatransfer.Transferable (JDK 1.1)

This interface defines the methods that a class must define if it is to act as the source object in a data transfer operation.

`getTransferDataFlavors()` should return an array of `DataFlavor` objects that specify the data types or formats in which the object can provide its data. The `DataFlavor` objects should be ordered from best format (most richly descriptive) to worst format.

`isDataFlavorSupported()` must return a boolean value indicating whether it can transfer data using a specified `DataFlavor`. Finally, `getTransferData()` must return an object that represents the data formatted as required by the specified `DataFlavor`.

`StringSelection` is a pre-defined class that implements the `Transferable` interface for the transfer of string data.

```
public abstract interface Transferable {
    // Public Instance Methods
            public abstract Object getTransferData(DataFlavor flavor) throws 
UnsupportedFlavorException, IOException;
            public abstract DataFlavor[] getTransferDataFlavors();
            public abstract boolean isDataFlavorSupported(DataFlavor flavor);
}
```

Implemented By:

`StringSelection`

Passed To:

`Clipboard.setContents(), ClipboardOwner.lostOwnership(), StringSelection.lostOwnership()`

Returned By:

`Clipboard.getContents()`

Type Of:

`Clipboard.contents`

20.25 java.awt.event.PaintEvent (JDK 1.1)

An event of this type indicates that a component should have its `update()` method invoked. (The `update()` method typically, by default, invokes the `paint()` method.)

`PaintEvent` differs from the other event types in `java.awt.event` in that it does not have a corresponding `EventListener` interface. `PaintEvent` is essentially for internal use by the AWT redisplay framework, so your programs should not try to handle it the way they handle other events. Instead, applets and custom components should simply override their `paint()` and/or `update()` methods to redraw themselves appropriately. AWT automatically invokes `update()` (which typically invokes `paint()`) when a `PaintEvent` arrives.

Although you do not typically use the `PaintEvent`, redraw events are implemented through this class for simplicity, so that they are on equal footing with other event types, and so that advanced programs can manipulate them through the `EventQueue`.

```
public class PaintEvent extends ComponentEvent {
    // Public Constructor
            public PaintEvent(Component source, int id, Rectangle updateRect);
    // Constants
            public static final int PAINT;
            public static final int PAINT_FIRST;
            public static final int PAINT_LAST;
            public static final int UPDATE;
    // Public Instance Methods
            public Rectangle getUpdateRect();
            public String paramString();  // Overrides ComponentEvent
            public void setUpdateRect(Rectangle updateRect);
}
```

Hierarchy:

```
Object->EventObject(Serializable)->AWTEvent->ComponentEvent->PaintEvent
```

java.awt.event.MouseMotionListener (JDK 1.1) | BOOK INDEX | java.awt.event.TextEvent (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

31.4 java.util.zip.Checksum (JDK 1.1)

This interface defines the methods required to compute a checksum on a stream of data. The checksum is computed based on the bytes of data supplied by the `update()` methods, and the current value of the checksum can be obtained at any time with the `getValue()` method. `reset()` resets the checksum to its default value--use this method before beginning a new stream of data.

Note that the checksum value computed by a `Checksum` object and returned through the `getValue()` method must fit into a `long` value. Therefore, this interface is not suitable for the cryptographic checksum algorithms used in cryptography and security.

The classes `CheckedInputStream` and `CheckedOutputStream` provide a higher-level API for computing a checksum on a stream of data.

```
public abstract interface Checksum {
    // Public Instance Methods
            public abstract long getValue();
            public abstract void reset();
            public abstract void update(int b);
            public abstract void update(byte[] b, int off, int len);
}
```

Implemented By:

Adler32, CRC32

Passed To:

CheckedInputStream(), CheckedOutputStream()

Returned By:

CheckedInputStream.getChecksum(), CheckedOutputStream.getChecksum()

PREVIOUS HOME NEXT
java.util.zip.CheckedOutputStream (JDK 1.1) BOOK INDEX java.util.zip.CRC32 (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

31.5 java.util.zip.CRC32 (JDK 1.1)

This class implements the `Checksum` interface and computes a checksum on a stream of data using the CRC-32 algorithm. The `CheckedInputStream` and `CheckedOutputStream` classes provide a higher-level interface to computing checksums on streams of data.

```
public class CRC32 extends Object implements Checksum {
    // Default Constructor: public CRC32()
    // Public Instance Methods
            public long getValue();  // From Checksum
            public void reset();  // From Checksum
            public void update(int b);  // From Checksum
            public native void update(byte[] b, int off, int len);  // From Checksum
            public void update(byte[] b);
}
```

Type Of:

GZIPInputStream.crc, GZIPOutputStream.crc

PREVIOUS

NEXT

20.29 java.awt.event.WindowEvent (JDK 1.1)

An event of this type indicates that an important action has occurred for a `Window` object.

Call `getWindow()` to determine the `Window` object that is the source of this event. Call `getID()` to determine the specific type of event that has occurred. Each of the following seven constants corresponds to one of the methods of the `WindowListener` interface:

`WINDOW_OPENED`

This type indicates that the window has been created and opened; it is only delivered the first time that a window is opened.

`WINDOW_CLOSING`

This type indicates that the user has requested that the window be closed through the system menu, or through a close button on the window's border, or by invoking a platform-defined keystroke, such as **Alt-F4** in Windows. The application should respond to this event by calling `hide()` or `destroy()` on the `Window` object.

`WINDOW_CLOSED`

This event type is delivered after a window is closed by a call to `hide()` or `destroy()`.

`WINDOW_ICONIFIED`

This event type is delivered when the user iconifies the window.

`WINDOW_DEICONIFIED`

This event type is delivered when the user de-iconifies the window.

WINDOW_ACTIVATED

This event type is delivered when the window is activated--that is, when it is given the keyboard focus and becomes the "active" window.

WINDOW_DEACTIVATED

This event type is delivered when the window ceases to be the active window, typically when the user activates some other window.

```
public class WindowEvent extends ComponentEvent {
    // Public Constructor
            public WindowEvent(Window source, int id);
    // Constants
            public static final int WINDOW_ACTIVATED;
            public static final int WINDOW_CLOSED;
            public static final int WINDOW_CLOSING;
            public static final int WINDOW_DEACTIVATED;
            public static final int WINDOW_DEICONIFIED;
            public static final int WINDOW_FIRST;
            public static final int WINDOW_ICONIFIED;
            public static final int WINDOW_LAST;
            public static final int WINDOW_OPENED;
    // Public Instance Methods
            public Window getWindow();
            public String paramString();  // Overrides ComponentEvent
}
```

Hierarchy:

Object->EventObject(Serializable)->AWTEvent->ComponentEvent->WindowEvent

Passed To:

AWTEventMulticaster.windowActivated(), AWTEventMulticaster.windowClosed(), AWTEventMulticaster.windowClosing(), AWTEventMulticaster.windowDeactivated(), AWTEventMulticaster.windowDeiconified(),

```
AWTEventMulticaster.windowIconified(),
AWTEventMulticaster.windowOpened(), Window.processWindowEvent(),
WindowAdapter.windowActivated(), WindowAdapter.windowClosed(),
WindowAdapter.windowClosing(), WindowAdapter.windowDeactivated(),
WindowAdapter.windowDeiconified(), WindowAdapter.windowIconified(),
WindowAdapter.windowOpened(), WindowListener.windowActivated(),
WindowListener.windowClosed(), WindowListener.windowClosing(),
WindowListener.windowDeactivated(),
WindowListener.windowDeiconified(), WindowListener.windowIconified(),
WindowListener.windowOpened()
```

30.6 java.util.Enumeration (JDK 1.0)

This interface defines the methods necessary to enumerate, or iterate through, a set of values, such as the set of values contained in a hashtable or binary tree. It is particularly useful for data structures, like hashtables, for which elements cannot simply be looked up by index, as they can in arrays. An `Enumeration` is usually not instantiated directly, but instead is created by the object that is to have its values enumerated. A number of classes, such as `Vector` and `Hashtable`, have methods that return `Enumeration` objects.

To use an `Enumeration` object, you use its two methods in a loop: `hasMoreElements()` returns `true` if there are more values to be enumerated, and can be used to determine whether a loop should continue. Within a loop, a call to `nextElement()` returns a value from the enumeration. An `Enumeration` makes no guarantees about the order in which the values are returned. The values in an `Enumeration` may be iterated through only once--there is no way to reset it to the beginning.

```
public abstract interface Enumeration {
    // Public Instance Methods
            public abstract boolean hasMoreElements();
            public abstract Object nextElement();
}
```

Implemented By:

StringTokenizer

Passed To:

SequenceInputStream()

Returned By:

AppletContext.getApplets(), Dictionary.elements(), Dictionary.keys(), FeatureDescriptor.attributeNames(), Hashtable.elements(), Hashtable.keys(), ListResourceBundle.getKeys(), MenuBar.shortcuts(), Properties.propertyNames(), PropertyResourceBundle.getKeys(), ResourceBundle.getKeys(), Vector.elements(), ZipFile.entries()

PREVIOUS | HOME | NEXT

java.util.EmptyStackException (JDK 1.0) | BOOK INDEX | java.util.EventListener (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

20.16 java.awt.event.ItemListener (JDK 1.1)

This interface defines the method that an object must implement to "listen" for item events on AWT components. When an `ItemEvent` occurs, an AWT component notifies its registered `ItemListener` objects by invoking their `itemStateChanged()` methods.

```
public abstract interface ItemListener extends EventListener {
    // Public Instance Methods
            public abstract void itemStateChanged(ItemEvent e);
}
```

Implemented By:

`AWTEventMulticaster`

Passed To:

`AWTEventMulticaster.add(), AWTEventMulticaster.remove(), Checkbox.addItemListener(), Checkbox.removeItemListener(), CheckboxMenuItem.addItemListener(), CheckboxMenuItem.removeItemListener(), Choice.addItemListener(), Choice.removeItemListener(), ItemSelectable.addItemListener(), ItemSelectable.removeItemListener(), List.addItemListener(), List.removeItemListener()`

Returned By:

`AWTEventMulticaster.add(), AWTEventMulticaster.remove()`

PREVIOUS java.awt.event.ItemEvent (JDK 1.1)

HOME
BOOK INDEX

NEXT java.awt.event.KeyAdapter (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

20.17 java.awt.event.KeyAdapter (JDK 1.1)

This class is a trivial implementation of `KeyListener`; it provides empty bodies for each of the methods of that interface. When you are not interested in all of these methods, it is often easier to subclass `KeyAdapter` than it is to implement `KeyListener` from scratch.

```
public abstract class KeyAdapter extends Object implements KeyListener {
    // Default Constructor: public KeyAdapter()
    // Public Instance Methods
            public void keyPressed(KeyEvent e);  // From KeyListener
            public void keyReleased(KeyEvent e);  // From KeyListener
            public void keyTyped(KeyEvent e);  // From KeyListener
}
```

Hierarchy:

```
Object->KeyAdapter(KeyListener(EventListener))
```

20.19 java.awt.event.KeyListener (JDK 1.1)

This interface defines the methods that an object must implement to "listen" for key events on AWT components. When a `KeyEvent` occurs, an AWT component notifies its registered `KeyListener` objects by invoking one of their methods.

An easy way to implement this interface is by subclassing the `KeyAdapter` class.

```
public abstract interface KeyListener extends EventListener {
    // Public Instance Methods
            public abstract void keyPressed(KeyEvent e);
            public abstract void keyReleased(KeyEvent e);
            public abstract void keyTyped(KeyEvent e);
}
```

Implemented By:

`AWTEventMulticaster, KeyAdapter`

Passed To:

`AWTEventMulticaster.add(), AWTEventMulticaster.remove(), Component.addKeyListener(), Component.removeKeyListener()`

Returned By:

`AWTEventMulticaster.add(), AWTEventMulticaster.remove()`

java.awt.event.KeyEvent (JDK 1.1) | BOOK INDEX | java.awt.event.MouseAdapter (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

19.2 java.awt.datatransfer.ClipboardOwner (JDK 1.1)

This interface defines the single method that an object that places data on a `Clipboard` must implement. This method is used to notify the object when its data on the clipboard is replaced by other, more recent, data. An object that places data on a clipboard must remain ready to satisfy requests for that data until `lostOwnership()` is called.

```
public abstract interface ClipboardOwner {
    // Public Instance Methods
            public abstract void lostOwnership(Clipboard clipboard, Transferable
contents);
}
```

Implemented By:

`StringSelection`

Passed To:

`Clipboard.setContents()`

Type Of:

`Clipboard.owner`

20.20 java.awt.event.MouseAdapter (JDK 1.1)

This class is a trivial implementation of `MouseListener`; it provides empty bodies for each of the methods of that interface. When you are not interested in all of these methods, it is often easier to subclass `MouseAdapter` than it is to implement `MouseListener` from scratch.

```
public abstract class MouseAdapter extends Object implements MouseListener {
    // Default Constructor: public MouseAdapter()
    // Public Instance Methods
            public void mouseClicked(MouseEvent e);  // From MouseListener
            public void mouseEntered(MouseEvent e);  // From MouseListener
            public void mouseExited(MouseEvent e);  // From MouseListener
            public void mousePressed(MouseEvent e);  // From MouseListener
            public void mouseReleased(MouseEvent e);  // From MouseListener
}
```

Hierarchy:

```
Object->MouseAdapter(MouseListener(EventListener))
```

20.22 java.awt.event.MouseListener (JDK 1.1)

This interface defines the methods that an object must implement to "listen" for mouse events on AWT components. When a `MouseEvent` occurs, an AWT component notifies its registered `MouseListener` objects (or `MouseMotionListener` objects if the event involves mouse motion) by invoking one of their methods.

An easy way to implement this interface is by subclassing the `MouseAdapter` class.

```
public abstract interface MouseListener extends EventListener {
    // Public Instance Methods
            public abstract void mouseClicked(MouseEvent e);
            public abstract void mouseEntered(MouseEvent e);
            public abstract void mouseExited(MouseEvent e);
            public abstract void mousePressed(MouseEvent e);
            public abstract void mouseReleased(MouseEvent e);
}
```

Implemented By:

`AWTEventMulticaster, MouseAdapter`

Passed To:

`AWTEventMulticaster.add(), AWTEventMulticaster.remove(), Component.addMouseListener(), Component.removeMouseListener()`

Returned By:

`AWTEventMulticaster.add(), AWTEventMulticaster.remove()`

20.23 java.awt.event.MouseMotionAdapter (JDK 1.1)

This class is a trivial implementation of `MouseMotionListener`; it provides empty bodies for each of the methods of that interface. When you are not interested in all of these methods, it is often easier to subclass `MouseMotionAdapter` than it is to implement `MouseMotionListener` from scratch.

```
public abstract class MouseMotionAdapter extends Object implements
MouseMotionListener {
    // Default Constructor: public MouseMotionAdapter()
    // Public Instance Methods
            public void mouseDragged(MouseEvent e);  // From MouseMotionListener
            public void mouseMoved(MouseEvent e);  // From MouseMotionListener
}
```

Hierarchy:

```
Object->MouseMotionAdapter(MouseMotionListener(EventListener))
```

20.24 java.awt.event.MouseMotionListener (JDK 1.1)

This interface defines the methods that an object must implement to "listen" for mouse motion events on AWT components. When a `MouseEvent` occurs involving a mouse drag or mouse motion with no buttons down, an AWT component notifies its registered `MouseMotionListener` objects by invoking one of their methods.

An easy way to implement this is by subclassing the `MouseMotionAdapter` class.

```
public abstract interface MouseMotionListener extends EventListener {
    // Public Instance Methods
            public abstract void mouseDragged(MouseEvent e);
            public abstract void mouseMoved(MouseEvent e);
}
```

Implemented By:

`AWTEventMulticaster, MouseMotionAdapter`

Passed To:

`AWTEventMulticaster.add(), AWTEventMulticaster.remove(), Component.addMouseMotionListener(), Component.removeMouseMotionListener()`

Returned By:

`AWTEventMulticaster.add(), AWTEventMulticaster.remove()`

PREVIOUS | HOME | NEXT

java.awt.event.MouseMotionAdapter (JDK 1.1) | BOOK INDEX | java.awt.event.PaintEvent (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

30.19 java.util.Random (JDK 1.0)

This class implements a pseudo-random number generator. `nextDouble()` and `nextFloat()` return a value between 0.0 and 1.0. `nextLong()` and `nextInt()` return `long` and `int` values distributed across the range of those data types. `nextGaussian()` returns pseudo-random values with a Gaussian distribution--the mean of the values is 0.0, and the standard deviation is 1.0. You can use the `setSeed()` method or the optional constructor argument to initialize the pseudo-random number generator with some variable seed value other than the current time (the default), or with a constant to ensure a repeatable sequence of pseudo-randomness.

```
public class Random extends Object implements Serializable {
    // Public Constructors
            public Random();
            public Random(long seed);
    // Public Instance Methods
        1.1public void nextBytes(byte[] bytes);
            public double nextDouble();
            public float nextFloat();
            public synchronized double nextGaussian();
            public int nextInt();
            public long nextLong();
            public synchronized void setSeed(long seed);
    // Protected Instance Methods
        1.1protected synchronized int next(int bits);
}
```

Passed To:

BigInteger()

29.4 java.text.CollationElementIterator (JDK 1.1)

A `CollationElementIterator` object is returned by the `getCollationElementIterator()` method of the `RuleBasedCollator` object. The purpose of this class is to allow a program to iterate (with the `next()` method) through the characters of a string, returning ordering values for each of the "collation keys" in the string. Note that collation keys are not exactly the same thing as characters. In the traditional Spanish collation order, for example, the two-character sequence "ch" is treated as a single collation key that comes alphabetically between the letters "c" and "d."

The value returned by the `next()` method is the collation order of the next collation key in the string. This numeric value can be directly compared to the value returned by `next()` for other `CollationElementIterator` objects. The value returned by `next()` can also be decomposed into primary, secondary, and tertiary ordering values with the static methods of this class.

This class is used by `RuleBasedCollator` to implement its `compare()` method, and to create `CollationKey` objects. Few applications ever need to use it directly.

```
public final class CollationElementIterator extends Object {
    // No Constructor
    // Constants
            public static final int NULLORDER;
    // Class Methods
            public static final int primaryOrder(int order);
            public static final short secondaryOrder(int order);
            public static final short tertiaryOrder(int order);
    // Public Instance Methods
            public int next();
            public void reset();
}
```

Returned By:

RuleBasedCollator.getCollationElementIterator()

20.26 java.awt.event.TextEvent (JDK 1.1)

An event of this type indicates that the user has edited the text value that appears in a `TextField`, `TextArea`, or other `TextComponent`. This event is triggered by any change to the displayed text. Note that this is not the same as the `ActionEvent` sent by the `TextField` object when the user edits the text and strikes the **Return** key.

Use the inherited `getSource()` to determine the object that was the source of this event. You have to cast that object to its `TextComponent` type. Call `getID()` to determine the type of a `TextEvent`. The standard AWT components always generate text events of type `TEXT_VALUE_CHANGED`.

```
public class TextEvent extends AWTEvent {
    // Public Constructor
            public TextEvent(Object source, int id);
    // Constants
            public static final int TEXT_FIRST;
            public static final int TEXT_LAST;
            public static final int TEXT_VALUE_CHANGED;
    // Public Instance Methods
            public String paramString();  // Overrides AWTEvent
}
```

Hierarchy:

`Object->EventObject(Serializable)->AWTEvent->TextEvent`

Passed To:

`AWTEventMulticaster.textValueChanged()`, `TextComponent.processTextEvent()`, `TextListener.textValueChanged()`

23.13 java.beans.PropertyChangeListener (JDK 1.1)

This interface is an extension of java.util.EventListener and defines the method that a class must implement in order to be notified when property changes occur. A PropertyChangeEvent is sent out to all registered PropertyChangeListener objects whenever a bean changes one of its "bound" properties, or whenever a PropertyEditor or Customizer changes the value of a property.

```
public abstract interface PropertyChangeListener extends EventListener {
    // Public Instance Methods
            public abstract void propertyChange(PropertyChangeEvent evt);
}
```

Passed To:

Customizer.addPropertyChangeListener(), Customizer.removePropertyChangeListener(), PropertyChangeSupport.addPropertyChangeListener(), PropertyChangeSupport.removePropertyChangeListener(), PropertyEditor.addPropertyChangeListener(), PropertyEditor.removePropertyChangeListener(), PropertyEditorSupport.addPropertyChangeListener(), PropertyEditorSupport.removePropertyChangeListener()

24.11 java.io.DataInputStream (JDK 1.0)

This class is a type of `FilterInputStream` that allows you to read binary representations of Java primitive data types in a portable way. Create a `DataInputStream` by specifying the `InputStream` that is to be filtered in the call to the constructor.

Many of this class's methods read and return a single Java primitive type, in binary format, from the stream. `readUnsignedByte()` and `readUnsignedShort()` read unsigned values and return them as `int` values, since unsigned `byte` and `short` types are not supported in Java. `read()` reads data into an array of bytes, blocking until at least some data are available. By contrast, `readFully()` reads data into an array of bytes, but blocks until all of the requested data become available. `skipBytes()` blocks until the specified number of bytes have been read and discarded.

`readLine()` reads characters from the stream until it encounters a newline, a carriage return, or a newline carriage return pair. The returned string is not terminated with a newline or carriage return. This method is deprecated in Java 1.1; see `BufferedReader` for an alternative. `readUTF()` reads a string of Unicode text encoded in a slightly modified version of the UTF-8 "transformation format." UTF-8 is an ASCII-compatible encoding of Unicode characters that is often used for the transmission and storage of Unicode text. This class uses a modified UTF-8 encoding that never contains embedded null characters.

`DataInputStream` only reads primitive Java types. Use `ObjectInputStream` to read object values.

```
public class DataInputStream extends FilterInputStream implements DataInput {
    // Public Constructor
            public DataInputStream(InputStream in);
    // Class Methods
            public static final String readUTF(DataInput in) throws IOException;
    // Public Instance Methods
            public final int read(byte[] b) throws IOException;  // Overrides
FilterInputStream
            public final int read(byte[] b, int off, int len) throws IOException;  //
Overrides FilterInputStream
            public final boolean readBoolean() throws IOException;  // From DataInput
            public final byte readByte() throws IOException;  // From DataInput
            public final char readChar() throws IOException;  // From DataInput
            public final double readDouble() throws IOException;  // From DataInput
            public final float readFloat() throws IOException;  // From DataInput
            public final void readFully(byte[] b) throws IOException;  // From
DataInput
            public final void readFully(byte[] b, int off, int len) throws
IOException;  // From DataInput
            public final int readInt() throws IOException;  // From DataInput
```

```
        #   public final String readLine() throws IOException;  // From DataInput
            public final long readLong() throws IOException;  // From DataInput
            public final short readShort() throws IOException;  // From DataInput
            public final String readUTF() throws IOException;  // From DataInput
            public final int readUnsignedByte() throws IOException;  // From 
DataInput
            public final int readUnsignedShort() throws IOException;  // From 
DataInput
            public final int skipBytes(int n) throws IOException;  // From DataInput
}
```

Hierarchy:

Object->InputStream->FilterInputStream->DataInputStream(DataInput)

24.10 java.io.DataInput (JDK 1.0)

This interface defines the methods required for streams that can read Java primitive data types in a machine-independent binary format. It is implemented by `DataInputStream` and `RandomAccessFile`. See `DataInputStream` for more information on the methods.

```
public abstract interface DataInput {
    // Public Instance Methods
            public abstract boolean readBoolean() throws IOException;
            public abstract byte readByte() throws IOException;
            public abstract char readChar() throws IOException;
            public abstract double readDouble() throws IOException;
            public abstract float readFloat() throws IOException;
            public abstract void readFully(byte[] b) throws IOException;
            public abstract void readFully(byte[] b, int off, int len) throws
IOException;
            public abstract int readInt() throws IOException;
            public abstract String readLine() throws IOException;
            public abstract long readLong() throws IOException;
            public abstract short readShort() throws IOException;
            public abstract String readUTF() throws IOException;
            public abstract int readUnsignedByte() throws IOException;
            public abstract int readUnsignedShort() throws IOException;
            public abstract int skipBytes(int n) throws IOException;
}
```

Extended By:

ObjectInput

Implemented By:

DataInputStream, RandomAccessFile

Passed To:

DataInputStream.readUTF()

24.15 java.io.Externalizable (JDK 1.1)

This interface defines the methods that must be implemented by an object that wants complete control over the way it is serialized. The `writeExternal()` and `readExternal()` methods should be implemented to write and read object data in some arbitrary format, using the methods of the `DataOutput` and `DataInput` objects. `Externalizable` objects must serialize their own fields, and are also responsible for serializing the fields of their superclasses.

Most objects do not need to define a custom output format and can use the `Serializable` interface instead of `Externalizable` for serialization.

```
public abstract interface Externalizable extends Serializable {
    // Public Instance Methods
            public abstract void readExternal(ObjectInput in) throws IOException, 
ClassNotFoundException;
            public abstract void writeExternal(ObjectOutput out) throws IOException;
}
```

25.50 java.lang.Runnable (JDK 1.0)

This interface specifies the `run()` method that is required for use with the `Thread` class. Any class that implements this interface can provide the "body" of a thread. See `Thread` for more information.

```
public abstract interface Runnable {
    // Public Instance Methods
            public abstract void run();
}
```

Implemented By:

Thread

Passed To:

Thread()

22.12 java.awt.peer.LabelPeer (JDK 1.0)

```
public abstract interface LabelPeer extends ComponentPeer {
    // Public Instance Methods
            public abstract void setAlignment(int alignment);
            public abstract void setText(String label);
}
```

Returned By:

Toolkit.createLabel()

22.4 java.awt.peer.CheckboxPeer (JDK 1.0)

```
public abstract interface CheckboxPeer extends ComponentPeer {
    // Public Instance Methods
            public abstract void setCheckboxGroup(CheckboxGroup g);
            public abstract void setLabel(String label);
            public abstract void setState(boolean state);
}
```

Returned By:

Toolkit.createCheckbox()

21.3 java.awt.image.CropImageFilter (JDK 1.0)

This class implements an `ImageFilter` that crops an image to a specified rectangle. The methods defined by this class are used for communication between the filter and its `FilteredImageSource` and should never be called directly.

```
public class CropImageFilter extends ImageFilter {
    // Public Constructor
            public CropImageFilter(int x, int y, int w, int h);
    // Public Instance Methods
            public void setDimensions(int w, int h);  // Overrides ImageFilter
            public void setPixels(int x, int y, int w, int h, ColorModel model,
            public void setPixels'u'byte[] pixels, int off, int scansize);  //
Overrides ImageFilter
            public void setPixels(int x, int y, int w, int h, ColorModel model,
            public void setPixels'u'int[] pixels, int off, int scansize);  //
Overrides ImageFilter
            public void setProperties(Hashtable props);  // Overrides ImageFilter
}
```

Hierarchy:

```
Object->ImageFilter(ImageConsumer, Cloneable)->CropImageFilter
```

22.9 java.awt.peer.FileDialogPeer (JDK 1.0)

```
public abstract interface FileDialogPeer extends DialogPeer {
    // Public Instance Methods
            public abstract void setDirectory(String dir);
            public abstract void setFile(String file);
            public abstract void setFilenameFilter(FilenameFilter filter);
}
```

Hierarchy:

(FileDialogPeer(DialogPeer(WindowPeer(ContainerPeer(ComponentPeer)))))

Returned By:

Toolkit.createFileDialog()

22.11 java.awt.peer.FramePeer (JDK 1.0)

```
public abstract interface FramePeer extends WindowPeer {
    // Public Instance Methods
            public abstract void setIconImage(Image im);
            public abstract void setMenuBar(MenuBar mb);
            public abstract void setResizable(boolean resizeable);
            public abstract void setTitle(String title);
}
```

Hierarchy:

(FramePeer(WindowPeer(ContainerPeer(ComponentPeer))))

Returned By:

Toolkit.createFrame()

22.22 java.awt.peer.ScrollbarPeer (JDK 1.0)

```
public abstract interface ScrollbarPeer extends ComponentPeer {
    // Public Instance Methods
            public abstract void setLineIncrement(int l);
            public abstract void setPageIncrement(int l);
            public abstract void setValues(int value, int visible, int minimum, int 
maximum);
}
```

Returned By:

Toolkit.createScrollbar()

22.8 java.awt.peer.DialogPeer (JDK 1.0)

```
public abstract interface DialogPeer extends WindowPeer {
    // Public Instance Methods
            public abstract void setResizable(boolean resizeable);
            public abstract void setTitle(String title);
}
```

Hierarchy:

(DialogPeer(WindowPeer(ContainerPeer(ComponentPeer))))

Extended By:

FileDialogPeer

Returned By:

Toolkit.createDialog()

22.3 java.awt.peer.CheckboxMenuItemPeer (JDK 1.0)

```
public abstract interface CheckboxMenuItemPeer extends MenuItemPeer {
    // Public Instance Methods
            public abstract void setState(boolean t);
}
```

Hierarchy:

(CheckboxMenuItemPeer(MenuItemPeer(MenuComponentPeer)))

Returned By:

Toolkit.createCheckboxMenuItem()

22.20 java.awt.peer.PopupMenuPeer (JDK 1.1)

```
public abstract interface PopupMenuPeer extends MenuPeer {
    // Public Instance Methods
            public abstract void show(Event e);
}
```

Hierarchy:

(PopupMenuPeer(MenuPeer(MenuItemPeer(MenuComponentPeer))))

Returned By:

Toolkit.createPopupMenu()

20.27 java.awt.event.TextListener (JDK 1.1)

This interface defines the method that an object must implement to "listen" for text events on AWT components. When a `TextEvent` occurs, an AWT component notifies its registered `TextListener` objects by invoking their `textValueChanged()` methods.

```
public abstract interface TextListener extends EventListener {
    // Public Instance Methods
            public abstract void textValueChanged(TextEvent e);
}
```

Implemented By:

`AWTEventMulticaster`

Passed To:

`AWTEventMulticaster.add(), AWTEventMulticaster.remove(), TextComponent.addTextListener(), TextComponent.removeTextListener()`

Returned By:

`AWTEventMulticaster.add(), AWTEventMulticaster.remove()`

Type Of:

`TextComponent.textListener`

22.26 java.awt.peer.WindowPeer (JDK 1.0)

```
public abstract interface WindowPeer extends ContainerPeer {
    // Public Instance Methods
            public abstract void toBack();
            public abstract void toFront();
}
```

Hierarchy:

(WindowPeer(ContainerPeer(ComponentPeer)))

Extended By:

DialogPeer, FramePeer

Returned By:

Toolkit.createWindow()

25.69 java.lang.Void (JDK 1.1)

The Void class may not be instantiated, and serves merely as a placeholder for its static TYPE field, which is a Class object constant that represents the void type.

```
public final class Void extends Object {
    // No Constructor
    // Constants
            public static final Class TYPE;
}
```

30.16 java.util.Observer (JDK 1.0)

This interface defines the `update()` method required for an object to "observe" subclasses of `Observable`. An `Observer` registers interest in an `Observable()` by calling the `Observable.addObserver()` method. `Observer` objects that have been registered in this way will have their `update()` method invoked by the `Observable` when that object has changed.

```
public abstract interface Observer {
    // Public Instance Methods
            public abstract void update(Observable o, Object arg);
}
```

Passed To:

Observable.addObserver(), Observable.deleteObserver()

24.40 java.io.ObjectInputValidation (JDK 1.1)

A class implements this interface and defines the `validateObject()` method in order to be able to validate itself when it, and all the objects it depends on, have been completely deserialized from an `ObjectInputStream`. The `validateObject()` method is only invoked, however, if the object is passed to `ObjectInputStream.registerValidation()`; this must be done from the `readObject()` method of the object. Note that if an object is deserialized as part of a larger object graph, its `validateObject()` method is not invoked until the entire graph is read, and the original call to `ObjectInputStream.readObject()` is about to return.

`validateObject()` should throw an `InvalidObjectException` if the object fails validation. This stops object serialization, and causes the original call to `ObjectInputStream.readObject()` to terminate with the `InvalidObjectException` exception.

```
public abstract interface ObjectInputValidation {
    // Public Instance Methods
            public abstract void validateObject() throws InvalidObjectException;
}
```

Passed To:

ObjectInputStream.registerValidation()

23.21 java.beans.VetoableChangeListener (JDK 1.1)

This interface is an extension of `java.util.EventListener` and defines the method that a class must implement in order to be notified when a Java bean makes a change to a "constrained" property. A `PropertyChangeEvent` is passed to the `vetoableChange()` method when such a change occurs, and if the `VetoableChangeListener` wants to prevent the change from occurring, this method should throw a `PropertyVetoException`.

```
public abstract interface VetoableChangeListener extends EventListener {
    // Public Instance Methods
            public abstract void vetoableChange(PropertyChangeEvent evt) throws
PropertyVetoException;
}
```

Passed To:

VetoableChangeSupport.addVetoableChangeListener(), VetoableChangeSupport.removeVetoableChangeListener()

20.28 java.awt.event.WindowAdapter (JDK 1.1)

This class is a trivial implementation of `WindowListener`; it provides empty bodies for each of the methods of that interface. When you are not interested in all of these methods, it is often easier to subclass `WindowAdapter` than it is to implement `WindowListener` from scratch.

```
public abstract class WindowAdapter extends Object implements WindowListener {
    // Default Constructor: public WindowAdapter()
    // Public Instance Methods
            public void windowActivated(WindowEvent e);  // From WindowListener
            public void windowClosed(WindowEvent e);  // From WindowListener
            public void windowClosing(WindowEvent e);  // From WindowListener
            public void windowDeactivated(WindowEvent e);  // From WindowListener
            public void windowDeiconified(WindowEvent e);  // From WindowListener
            public void windowIconified(WindowEvent e);  // From WindowListener
            public void windowOpened(WindowEvent e);  // From WindowListener
}
```

Hierarchy:

```
Object->WindowAdapter(WindowListener(EventListener))
```

20.30 java.awt.event.WindowListener (JDK 1.1)

This interface defines the methods that an object must implement to "listen" for window events on AWT components. When a `WindowEvent` occurs, an AWT component notifies its registered `WindowListener` objects by invoking one of their methods.

An easy way to implement this interface is by subclassing the `WindowAdapter` class.

```
public abstract interface WindowListener extends EventListener {
    // Public Instance Methods
            public abstract void windowActivated(WindowEvent e);
            public abstract void windowClosed(WindowEvent e);
            public abstract void windowClosing(WindowEvent e);
            public abstract void windowDeactivated(WindowEvent e);
            public abstract void windowDeiconified(WindowEvent e);
            public abstract void windowIconified(WindowEvent e);
            public abstract void windowOpened(WindowEvent e);
}
```

Implemented By:

`AWTEventMulticaster, WindowAdapter`

Passed To:

`AWTEventMulticaster.add(), AWTEventMulticaster.remove(),`
`Window.addWindowListener(), Window.removeWindowListener()`

Returned By:

`AWTEventMulticaster.add(), AWTEventMulticaster.remove()`

PREVIOUS | HOME | NEXT
java.awt.event.WindowEvent (JDK 1.1) | BOOK INDEX | The java.awt.image Package

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

24.12 java.io.DataOutput (JDK 1.0)

This interface defines the methods required for streams that can write character data or Java primitive data types in a machine-independent binary format. It is implemented by `DataOutputStream` and `RandomAccessFile`. See `DataOutputStream` for more information on the methods.

```
public abstract interface DataOutput {
    // Public Instance Methods
            public abstract void write(int b) throws IOException;
            public abstract void write(byte[] b) throws IOException;
            public abstract void write(byte[] b, int off, int len) throws 
IOException;
            public abstract void writeBoolean(boolean v) throws IOException;
            public abstract void writeByte(int v) throws IOException;
            public abstract void writeBytes(String s) throws IOException;
            public abstract void writeChar(int v) throws IOException;
            public abstract void writeChars(String s) throws IOException;
            public abstract void writeDouble(double v) throws IOException;
            public abstract void writeFloat(float v) throws IOException;
            public abstract void writeInt(int v) throws IOException;
            public abstract void writeLong(long v) throws IOException;
            public abstract void writeShort(int v) throws IOException;
            public abstract void writeUTF(String str) throws IOException;
}
```

Extended By:

ObjectOutput

Implemented By:

DataOutputStream, RandomAccessFile

PREVIOUS java.io.DataInputStream (JDK 1.0)

HOME

BOOK INDEX

NEXT java.io.DataOutputStream (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

18.4 java.awt.AWTException (JDK 1.0)

Signals that an exception has occurred in the `java.awt` package.

```
public class AWTException extends Exception {
    // Public Constructor
            public AWTException(String msg);
}
```

Hierarchy:

```
Object->Throwable(Serializable)->Exception->AWTException
```

18.31 java.awt.IllegalComponentStateException (JDK 1.1)

This exception signals that an AWT component is not in the appropriate state (for example, it hasn't been added to a container yet or is currently hidden) for some requested operation.

```
public class IllegalComponentStateException extends IllegalStateException {
    // Public Constructors
            public IllegalComponentStateException();
            public IllegalComponentStateException(String s);
}
```

Hierarchy:

```
Object->Throwable(Serializable)->Exception->RuntimeException->
IllegalStateException->IllegalComponentStateException
```

9.3 Serialization and Class Versioning

When an object is serialized, some information about its class must obviously be serialized with it, so that the correct class file can be loaded when the object is deserialized. This information about the class is represented by the `java.io.ObjectStreamClass` class. It contains the fully-qualified name of the class and a version number. The version number is very important because an early version of a class may not be able to deserialize a serialized instance created by a later version of the same class.

The version number of a class is stored in a constant `long` field named `serialVersionUID`. For example, a class might declare its version number with a field like this:

```
static final long serialVersionUID = 280432937854755317L;
```

If a class does not define a `serialVersionUID` constant, the `ObjectOutputStream` automatically computes a unique version number for it by applying the Secure Hash Algorithm (SHA) to the name of the class, its interfaces, fields, and methods. In this case, any change to a field in a class or to a non-private class method signature results in a change to the automatically-computed unique version number.

If you need to make minor changes to a class without breaking serialization compatibility, you should explicitly declare a `serialVersionUID` constant so that an updated and incompatible version number is not automatically generated. You can use the *serialver* program that is provided with the JDK to compute an initial value for this constant for the first version of your class. When you make minor, compatible changes to the class, leave the constant unchanged. Then, if you make larger changes that break serialization compatibility, run *serialver* again to generate an updated version number.

9.4 Serialized Applets

One of the uses of object serialization in Java 1.1 is for serialized applets. As discussed in Chapter 15, *Java-Related HTML Tags*, the `<APPLET>` tag has a new attribute, `OBJECT`, that can be used in place of the `CODE` attribute to specify a serialized object file instead of a class file. When such an `<APPLET>` tag is encountered, the applet viewer or Web browser creates the applet by deserializing it.

The reason that this is an interesting thing to do is that it allows an applet to be shipped in a pre-initialized state. The code for the applet need not even include the code that performed the initialization. For example, imagine a GUI builder tool that allows a programmer to build a GUI using a point-and-click interface. Such a tool could create a tree of AWT components within an `Applet` panel, and then serialize the applet, including all of the GUI components it contains. When deserialized, the applet would have a complete GUI, despite the fact that the applet's class file does not contain any code to create the GUI.

You can experiment with applet serialization with the JDK 1.1 *appletviewer* program. Start an applet running in *appletviewer* in the usual way. This loads the applet and runs its `init()` and `start()` methods. Next, select the **Stop** item from the menu to stop the applet. Now use the **Save** menu item to serialize the applet to a file. By convention, your serialized applet file should be given a *.ser* extension. If the applet refers to any non-serializable objects, you may not be able to serialize it. For example, you may encounter problems serializing applets that use threads or images.

Once you have serialized an applet, create an HTML file with an `<APPLET>` tag something like this:

```
<APPLET OBJECT="MyApplet.ser" WIDTH=400 HEIGHT=200></APPLET >
```

Finally, you can use *appletviewer* with this new HTML file. It should deserialize and display the applet. When created in this way, the applet's `init()` method is not called (since it was called before serialization), but its `start()` method is called (because the applet should have been stopped before serialization).

PREVIOUS | HOME | NEXT
Serialization and Class Versioning | BOOK INDEX | Advanced Serialization

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

9.5 Advanced Serialization

The Object Serialization API also has some advanced features beyond those demonstrated above. One such feature is the `Externalizable` interface. If a class implements this interface, the `ObjectInputStream` and `ObjectOutputStream` classes use an object's `readExternal()` and `writeExternal()` methods to read and write its state from a stream. Most classes that want custom serialization behavior simply implement `readObject()` and `writeObject()` methods to do pre- and post-processing. `Externalizable` objects, on the other hand, take complete control over reading and writing their state. This interface is intended for objects that do not even want to use the basic serialized object data format used for all `Serializable` objects. A word processor object, for example, might "externalize" itself using its own native file format.

Another advanced serialization feature is the ability to register an `ObjectInputValidation` object that can verify that an object graph has been deserialized in a consistent, valid state. Typically, the root of an object graph registers such a validation object by calling the `registerValidation()` method of the `ObjectInputStream` from its own custom `readObject()` method. Then, when the graph has been read completely, the `validateObject()` method of the `ObjectInputValidation` object is called to perform whatever validity checks are necessary.

Finally, you may notice that `ObjectOutputStream` and `ObjectInputStream` have methods like `annotateClass()`, `replaceObject()`, `resolveClass()`, and `resolveObject()`. These are intended for use by subclasses of the object streams that implement special kinds of object serialization behavior.

28.21 java.net.UnknownServiceException (JDK 1.0)

Signals an attempt to use an unsupported service of a network connection.

```
public class UnknownServiceException extends IOException {
    // Public Constructors
            public UnknownServiceException();
            public UnknownServiceException(String msg);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->IOException->UnknownServiceException

Cloneable

Name

Cloneable

Synopsis

Interface Name:

`java.lang.Cloneable`

Super-interface:

None

Immediate Sub-interfaces:

`java.text.CharacterIterator`

Implemented by:

`java.awt.GridBagConstraints`, `java.awt.Insets`,

`java.awt.image.ImageFilter`,

`java.text.BreakIterator`,

`java.text.Collator`, `java.text.DateFormat`,

`java.text.DateFormatSymbols`,

`java.text.DecimalFormatSymbols`,

`java.text.Format`, `java.text.NumberFormat`,

`java.util.BitSet`, `java.util.Calendar`,

`java.util.Date`, `java.util.Hashtable`,

`java.util.Locale`, `java.util.TimeZone`,

`java.util.Vector`

Availability:

JDK 1.0 or later

Description

The `Cloneable` interface provides no functionality; it declares no methods or variables. This interface is simply provided as a way of indicating that an object can be cloned (that is, copied). A class that is declared as implementing this interface is assumed to have overridden the `Object` class' implementation of `clone()` with an implementation that can successfully clone instances of the class. The implementation of `clone()` that is provided by the `Object` class simply throws a `CloneNotSupportedException`.

Interface Declaration

```
public interface java.lang.Cloneable {
}
```

See Also

Exceptions; Object

28.20 java.net.UnknownHostException (JDK 1.0)

Signals that the name of a specified host could not be resolved.

```
public class UnknownHostException extends IOException {
    // Public Constructors
            public UnknownHostException(String host);
            public UnknownHostException();
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->IOException->UnknownHostException

Thrown By:

InetAddress.getAllByName(), InetAddress.getByName(), InetAddress.getLocalHost(), Socket()

12. Reflection

Contents:
Obtaining Class and Member Information
Invoking a Named Method

As noted in Chapter 4, *What's New in Java 1.1*, the Reflection API allows a Java program to inspect and manipulate itself. The Reflection API comprises the much-expanded `Class` class in `java.lang` and the `java.lang.reflect` package, which represents the members of a class with `Method`, `Constructor`, and `Field` objects.

Reflection can be used to obtain information about a class and its members. This is the technique that the JavaBeans "introspection" mechanism uses to determine the properties, events, and methods that are supported by a bean, for example. Reflection can also be used to manipulate objects in Java. You can use the `Field` class to query and set the values of fields, the `Method` class to invoke methods, and the `Constructor` class to create new objects. The examples in this chapter demonstrate both techniques for using the Reflection API.

12.1 Obtaining Class and Member Information

Example 12.1 shows a program that uses the `Class`, `Constructor`, `Field`, and `Method` classes to display information about a specified class. The program's output is similar to the class synopses that appear in the reference section of this book. (You might notice that the names of method arguments are not shown; argument names are not stored in class files, so they are not available through the Reflection API.)

Here is the output from using `ShowClass` on itself:

```
% java ShowClass ShowClass
public class ShowClass extends java.lang.Object {
 // Constructors
  public ShowClass();
 // Fields
 // Methods
  public static void main(java.lang.String[]);
  public static void print_class(java.lang.Class);
  public static java.lang.String typename(java.lang.Class);
  public static java.lang.String modifiers(int);
  public static void print_field(java.lang.reflect.Field);
  public static void print_method_or_constructor(java.lang.reflect.Member);
}
```

The code for this example is quite straightforward. It uses the `Class.forName()` method to dynamically load the named class, and calls various methods of `Class` object to look up the superclass, interfaces, and members of the class. The example uses `Constructor`, `Field`, and `Method` objects to obtain information about each member of the class.

Example 12.1: Obtaining Class and Member Information with the Reflection API

```
import java.lang.reflect.*;
/** A program that displays a class synopsis for the named class. */
public class ShowClass {
  /** The main method.  Print info about the named class. */
  public static void main(String[] args) throws ClassNotFoundException {
    Class c = Class.forName(args[0]);
    print_class(c);
  }
  /** Display the modifiers, name, superclass, and interfaces of a class
   *  or interface. Then go and list all constructors, fields, and methods. */
  public static void print_class(Class c)
  {
    // Print modifiers, type (class or interface), name, and superclass.
    if (c.isInterface()) {
      // The modifiers will include the "interface" keyword here...
      System.out.print(Modifier.toString(c.getModifiers()) +  c.getName());
    }
    else
      System.out.print(Modifier.toString(c.getModifiers()) + " class " +
                       c.getName() +
                       " extends " + c.getSuperclass().getName());
    // Print interfaces or super-interfaces of the class or interface.
    Class[] interfaces = c.getInterfaces();
    if ((interfaces != null) && (interfaces.length > 0)) {
      if (c.isInterface()) System.out.println(" extends ");
      else System.out.print(" implements ");
      for(int i = 0; i < interfaces.length; i++) {
        if (i > 0) System.out.print(", ");
        System.out.print(interfaces[i].getName());
      }
    }
    System.out.println(" {");            // Begin class member listing.
    // Now look up and display the members of the class.
    System.out.println(" // Constructors");
    Constructor[] constructors = c.getDeclaredConstructors();
    for(int i = 0; i < constructors.length; i++)      // Display constructors.
      print_method_or_constructor(constructors[i]);
    System.out.println(" // Fields");
    Field[] fields = c.getDeclaredFields();           // Look up fields.
    for(int i = 0; i < fields.length; i++)            // Display them.
      print_field(fields[i]);
    System.out.println(" // Methods");
    Method[] methods = c.getDeclaredMethods();        // Look up methods.
    for(int i = 0; i < methods.length; i++)           // Display them.
      print_method_or_constructor(methods[i]);
    System.out.println("}");             // End class member listing.
  }
  /** Return the name of an interface or primitive type, handling arrays. */
  public static String typename(Class t) {
    String brackets = "";
    while(t.isArray()) {
      brackets += "[]";
      t = t.getComponentType();
    }
    return t.getName() + brackets;
  }
  /** Return a string version of modifiers, handling spaces nicely. */
  public static String modifiers(int m) {
    if (m == 0) return "";
    else return Modifier.toString(m) + " ";
  }
  /** Print the modifiers, type, and name of a field. */
  public static void print_field(Field f) {
    System.out.println("  " +
                       modifiers(f.getModifiers()) +
                       typename(f.getType()) + " " + f.getName() + ";");
  }
  /** Print the modifiers, return type, name, parameter types, and exception
   *  type of a method or constructor.  Note the use of the Member interface
   *  to allow this method to work with both Method and Constructor objects. */
  public static void print_method_or_constructor(Member member) {
    Class returntype=null, parameters[], exceptions[];
    if (member instanceof Method) {
      Method m = (Method) member;
      returntype = m.getReturnType();
      parameters = m.getParameterTypes();
      exceptions = m.getExceptionTypes();
    } else {
      Constructor c = (Constructor) member;
      parameters = c.getParameterTypes();
      exceptions = c.getExceptionTypes();
    }
    System.out.print("  " + modifiers(member.getModifiers()) +
                     ((returntype!=null)? typename(returntype)+" " : "") +
                     member.getName() + "(");
    for(int i = 0; i < parameters.length; i++) {
      if (i > 0) System.out.print(", ");
      System.out.print(typename(parameters[i]));
    }
    System.out.print(")");
    if (exceptions.length > 0) System.out.print(" throws ");
    for(int i = 0; i < exceptions.length; i++) {
      if (i > 0) System.out.print(", ");
      System.out.print(typename(exceptions[i]));
    }
    System.out.println(";");
  }
}
```

PREVIOUS | HOME | NEXT
Formatted Messages | BOOK INDEX | Invoking a Named Method

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

30.5 java.util.EmptyStackException (JDK 1.0)

Signals that a `Stack` object is empty.

```
public class EmptyStackException extends RuntimeException {
    // Public Constructor
            public EmptyStackException();
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->RuntimeException->EmptyStackException

30.7 java.util.EventListener (JDK 1.1)

`EventListener` is a base interface for the Java 1.1 AWT event model. This interface defines no method or constants. Its purpose is to serve as a tag to identify objects that serve as event listeners. The event listener interfaces in the `java.awt.event`, and `java.beans` packages extend this interface.

```
public interface EventListener {
}
```

Extended By:

ActionListener, AdjustmentListener, ComponentListener, ContainerListener, FocusListener, ItemListener, KeyListener, MouseListener, MouseMotionListener, PropertyChangeListener, TextListener, VetoableChangeListener, WindowListener

Passed To:

AWTEventMulticaster(), AWTEventMulticaster.addInternal(), AWTEventMulticaster.remove(), AWTEventMulticaster.removeInternal(), AWTEventMulticaster.save()

Returned By:

AWTEventMulticaster.addInternal(), AWTEventMulticaster.remove(), AWTEventMulticaster.removeInternal()

Type Of:

AWTEventMulticaster.a, AWTEventMulticaster.b

PREVIOUS
java.util.Enumeration (JDK 1.0)

HOME
BOOK INDEX

NEXT
java.util.EventObject (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

30.14 java.util.NoSuchElementException (JDK 1.0)

Signals that there are no elements in an object (such as a `Vector`) or that there are no more elements in an object (such as an `Enumeration`).

```
public class NoSuchElementException extends RuntimeException {
    // Public Constructors
            public NoSuchElementException();
            public NoSuchElementException(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->RuntimeException->NoSuchElementException

30.25 java.util.TooManyListenersException (JDK 1.1)

An exception of this type signals that an AWT component or Java bean may only have one `EventListener` object registered for some specific type of event. That is, it signals that a particular event is a "unicast" event rather than a "multicast" event.

This exception type serves a formal purpose in the AWT and JavaBeans event model. Its presence in the `throws` clause of an `EventListener` registration method (even if the method never actually throws the exception) signals that an event is a unicast event.

```
public class TooManyListenersException extends Exception {
    // Public Constructors
            public TooManyListenersException();
            public TooManyListenersException(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->TooManyListenersException

10. Java Beans

Contents:

The JavaBeans API provides a framework for defining reusable, embeddable, modular software components. The JavaBeans Specification includes the following definition of a bean: "a reusable software component that can be manipulated visually in a builder tool." As you can see, this is a rather loose definition; beans can take a variety of forms. At the simplest level, individual AWT components (in Java 1.1) are all beans, while at a much higher level, an embeddable spreadsheet application might also function as a bean. Most beans, however, will probably fall somewhere between these two extremes.

One of the goals of the JavaBeans model is interoperability with similar component frameworks. So, for example, a native Windows program can, with an appropriate bridge or wrapper object, use a Java bean as if it were a COM or ActiveX component. The details of this sort of interoperability are beyond the scope of this chapter, however.

Beans can be used at three levels, by three different categories of programmers:

- If you are developing GUI editors, application builders, or other "beanbox" tools, you need the JavaBeans API to manipulate beans within these tools. *beanbox* is the name of the sample bean manipulation program provided by Sun in its Beans Development Kit (BDK). The term is a useful one, and I'll use it to describe any kind of graphical design tool or application builder that manipulates beans.

- If you are writing actual beans, you need the JavaBeans API to write code that can be used in any conforming beanbox.
- If you are writing applications that use beans developed by other programmers, or using a beanbox tool to combine those beans into an application, you do not actually need to be familiar with the JavaBeans API. You only need to be familiar with the documentation for individual beans that you are using.

This chapter explains how to use the JavaBeans API at the second level, or in other words, it describes how to write beans. It covers the following topics:

- Basic bean concepts and terminology
- Requirements for the simplest beans
- Packaging beans in JAR files
- Providing additional information about beans with the `BeanInfo` class
- Defining property editors to allow custom editing of bean properties
- Defining bean customizers to allow customization of an entire bean
- The various naming conventions and requirements imposed by the JavaBeans model

10.1 Bean Basics

We begin our discussion of beans with some basic concepts and terminology. Any object that conforms to certain basic rules can be a bean; there is no `Bean` class that all beans are required to subclass. Many beans are AWT components, but it is also quite possible, and often useful, to write "invisible" beans that do not have an on-screen appearance. (Just because a bean does not have an on-screen appearance in a finished application does not mean that it won't be visually manipulated by a beanbox tool, however.)

A bean exports properties, events, and methods. A *property* is a piece of the bean's internal state that can be programmatically set and queried, usually through a standard pair of `get` and `set` accessor methods. A bean may generate *events* in the same way that an AWT component, such as a `Button`, generates `ActionEvent` events. The JavaBeans API uses the same event model as the Java 1.1 AWT does. See Chapter 7, *Events*, for a full discussion of this model. A bean defines an event if it provides methods for adding and removing event listener objects from a list of interested listeners for that event. Finally, the *methods* exported by a bean are simply any `public` methods defined by the bean, excluding those methods used to get and set property values and register and remove event listeners.

In addition to the regular sort of properties described above, the JavaBeans API also provides support for "indexed properties," "bound properties," and "constrained properties." An indexed property is any property that has an array value and for which the bean provides methods to get and set individual elements of the array, as well as methods to get and set the entire array. A bound property is one that sends out a notification event when its value changes, while a constrained property is one that sends out a notification event when its value changes and allows the change to be vetoed by listeners.

Because Java allows dynamic loading of classes, beanbox programs can load arbitrary beans. The beanbox tool determines the properties, events, and methods a bean supports by using an "introspection" mechanism that is based on the `java.lang.reflect` reflection mechanism for obtaining information about the members of a class. A bean can also provide an auxiliary `BeanInfo` class that provides additional information about the bean. The `BeanInfo` class provides this additional information in the form of a number of `FeatureDescriptor` objects, each one describing a single feature of the bean. `FeatureDescriptor` has a number of subclasses: `BeanDescriptor`, `PropertyDescriptor`, `IndexedPropertyDescriptor`, `EventSetDescriptor`, `MethodDescriptor`, and `ParameterDescriptor`.

One of the primary tasks of a beanbox application is to allow the user to customize a bean by setting property values. A beanbox defines "property editors" for commonly used property types, such as numbers, strings, fonts, and colors. If a bean has properties of more complicated types, however, it may need to define a `PropertyEditor` class that enables the beanbox to let the user set values for that property.

In addition, a complex bean may not be satisfied with the property-by-property customization mechanism provided by most beanboxes. Such a bean may want to define a `Customizer` class, which creates a graphical interface that allows the user to configure a bean in some useful way. A particularly complex bean may even define customizers that serve as "wizards" that guide the user step-by-step through the customization process.

PREVIOUS	HOME	NEXT
Advanced Serialization	BOOK INDEX	A Simple Bean

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

25.38 java.lang.NegativeArraySizeException (JDK 1.0)

Signals an attempt to allocate an array with fewer than zero elements.

```
public class NegativeArraySizeException extends RuntimeException {
    // Public Constructors
            public NegativeArraySizeException();
            public NegativeArraySizeException(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->RuntimeException->NegativeArraySizeException

Thrown By:

Array.newInstance()

25.2 java.lang.ArithmeticException (JDK 1.0)

A `RuntimeException` that signals an exceptional arithmetic condition, such as integer division by zero.

```
public class ArithmeticException extends RuntimeException {
    // Public Constructors
            public ArithmeticException();
            public ArithmeticException(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->RuntimeException->ArithmeticException

Thrown By:

BigDecimal.divide(), BigDecimal.setScale(), BigInteger.add(), BigInteger.clearBit(), BigInteger.divide(), BigInteger.divideAndRemainder(), BigInteger.flipBit(), BigInteger.modInverse(), BigInteger.pow(), BigInteger.remainder(), BigInteger.setBit(), BigInteger.testBit()

25.3 java.lang.ArrayIndexOutOfBoundsException (JDK 1.0)

Signals that an array index less than zero or greater than or equal to the array size has been used.

```
public class ArrayIndexOutOfBoundsException extends IndexOutOfBoundsException {
    // Public Constructors
            public ArrayIndexOutOfBoundsException();
            public ArrayIndexOutOfBoundsException(int index);
            public ArrayIndexOutOfBoundsException(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->RuntimeException-> IndexOutOfBoundsException->ArrayIndexOutOfBoundsException

Thrown By:

Array.get(), Array.getBoolean(), Array.getByte(), Array.getChar(), Array.getDouble(), Array.getFloat(), Array.getInt(), Array.getLong(), Array.getShort(), Array.set(), Array.setBoolean(), Array.setByte(), Array.setChar(), Array.setDouble(), Array.setFloat(), Array.setInt(), Array.setLong(), Array.setShort()

25.4 java.lang.ArrayStoreException (JDK 1.0)

Signals an attempt to store the wrong type of object into an array.

```
public class ArrayStoreException extends RuntimeException {
    // Public Constructors
            public ArrayStoreException();
            public ArrayStoreException(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->RuntimeException->ArrayStoreException

25.9 java.lang.ClassCastException (JDK 1.0)

Signals an invalid cast of an object to a type of which it is not an instance.

```
public class ClassCastException extends RuntimeException {
    // Public Constructors
            public ClassCastException();
            public ClassCastException(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->RuntimeException->ClassCastException

25.10 java.lang.ClassCircularityError (JDK 1.0)

Signals that a circular dependency has been detected while performing initialization for a class.

```
public class ClassCircularityError extends LinkageError {
    // Public Constructors
            public ClassCircularityError();
            public ClassCircularityError(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Error->LinkageError->ClassCircularityError

25.11 java.lang.ClassFormatError (JDK 1.0)

Signals an error in the binary format of a class file.

```
public class ClassFormatError extends LinkageError {
    // Public Constructors
            public ClassFormatError();
            public ClassFormatError(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Error->LinkageError->ClassFormatError

25.13 java.lang.ClassNotFoundException (JDK 1.0)

Signals that a class to be loaded could not be found.

```
public class ClassNotFoundException extends Exception {
    // Public Constructors
            public ClassNotFoundException();
            public ClassNotFoundException(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->ClassNotFoundException

Thrown By:

Beans.instantiate(), Class.forName(), ClassLoader.findSystemClass(), ClassLoader.loadClass(), Externalizable.readExternal(), ObjectInput.readObject(), ObjectInputStream.defaultReadObject(), ObjectInputStream.readObject(), ObjectInputStream.resolveClass()

25.14 java.lang.CloneNotSupportedException (JDK 1.0)

Signals that the `clone()` method has been called for an object of a class that does not implement the `Cloneable` interface.

```
public class CloneNotSupportedException extends Exception {
    // Public Constructors
            public CloneNotSupportedException();
            public CloneNotSupportedException(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->CloneNotSupportedException

Thrown By:

Object.clone()

25.15 java.lang.Cloneable (JDK 1.0)

This interface defines no methods or variables, but indicates that the class that implements it may be cloned (i.e., copied) by calling the `Object` method `clone()`. Calling `clone()` for an object that does not implement this interface (and does not override `clone()` with its own implementation) causes a `CloneNotSupportedException` to be thrown.

```
public interface Cloneable {
}
```

Extended By:

CharacterIterator

Implemented By:

BitSet, BreakIterator, Calendar, Collator, Date, DateFormat, DateFormatSymbols, DecimalFormatSymbols, Format, GridBagConstraints, Hashtable, ImageFilter, Insets, Locale, NumberFormat, TimeZone, Vector

25.18 java.lang.Error (JDK 1.0)

This class forms the root of the error hierarchy in Java. Subclasses of `Error`, unlike subclasses of `Exception`, should generally not be caught, and generally cause termination of the program. Subclasses of `Error` need not be declared in the `throws` clause of a method definition.

`getMessage()` returns a message associated with the error. See `Throwable` for other methods.

```
public class Error extends Throwable {
    // Public Constructors
            public Error();
            public Error(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Error

Extended By:

AWTError, LinkageError, ThreadDeath, VirtualMachineError

25.19 java.lang.Exception (JDK 1.0)

This class forms the root of the exception hierarchy in Java. An `Exception` signals an abnormal condition that must be specially handled to prevent program termination. Exceptions may be caught and handled. Exceptions that are not subclasses of `RuntimeException` must be declared in the `throws` clause of any method that can throw them.

`getMessage()` returns a message associated with the exception. See `Throwable` for other methods.

```
public class Exception extends Throwable {
    // Public Constructors
            public Exception();
            public Exception(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception

Extended By:

AWTException, ClassNotFoundException, CloneNotSupportedException, DataFormatException, IllegalAccessException, InstantiationException, InterruptedException, IntrospectionException, InvocationTargetException, IOException, NoSuchFieldException, NoSuchMethodException, ParseException, PropertyVetoException, RuntimeException, TooManyListenersException, UnsupportedFlavorException

Passed To:

WriteAbortedException()

Type Of:

WriteAbortedException.detail

25.22 java.lang.IllegalAccessError (JDK 1.0)

Signals an attempted use of a class, method, or variable that is not accessible.

```
public class IllegalAccessError extends IncompatibleClassChangeError {
    // Public Constructors
            public IllegalAccessError();
            public IllegalAccessError(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Error->LinkageError->IncompatibleClassChangeError->IllegalAccessError

25.23 java.lang.IllegalAccessException (JDK 1.0)

Signals that a class or initializer is not accessible. Thrown by `Class.newInstance()`.

```
public class IllegalAccessException extends Exception {
    // Public Constructors
            public IllegalAccessException();
            public IllegalAccessException(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->IllegalAccessException

Thrown By:

Class.newInstance(), Constructor.newInstance(), Field.get(), Field.getBoolean(), Field.getByte(), Field.getChar(), Field.getDouble(), Field.getFloat(), Field.getInt(), Field.getLong(), Field.getShort(), Field.set(), Field.setBoolean(), Field.setByte(), Field.setChar(), Field.setDouble(), Field.setFloat(), Field.setInt(), Field.setLong(), Field.setShort(), Method.invoke()

25.24 java.lang.IllegalArgumentException (JDK 1.0)

Signals an illegal argument to a method. See subclasses `IllegalThreadStateException` and `NumberFormatException`.

```
public class IllegalArgumentException extends RuntimeException {
    // Public Constructors
            public IllegalArgumentException();
            public IllegalArgumentException(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->RuntimeException->IllegalArgumentException

Extended By:

IllegalThreadStateException, NumberFormatException

Thrown By:

Array.get(), Array.getBoolean(), Array.getByte(), Array.getChar(), Array.getDouble(), Array.getFloat(), Array.getInt(), Array.getLength(), Array.getLong(), Array.getShort(), Array.newInstance(), Array.set(), Array.setBoolean(), Array.setByte(), Array.setChar(), Array.setDouble(), Array.setFloat(), Array.setInt(), Array.setLong(), Array.setShort(), BigDecimal.divide(), BigDecimal.setScale(), BigInteger(), Constructor.newInstance(), Field.get(), Field.getBoolean(), Field.getByte(), Field.getChar(), Field.getDouble(), Field.getFloat(), Field.getInt(), Field.getLong(), Field.getShort(), Field.set(), Field.setBoolean(), Field.setByte(), Field.setChar(), Field.setDouble(), Field.setFloat(), Field.setInt(), Field.setLong(), Field.setShort(), Method.invoke(), PropertyEditor.setAsText(), PropertyEditorSupport.setAsText()

PREVIOUS | HOME | NEXT

java.lang.IllegalAccessException (JDK 1.0) | BOOK INDEX | java.lang.IllegalMonitorStateException (JDK 1.0)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

25.25 java.lang.IllegalMonitorStateException (JDK 1.0)

Signals an illegal monitor state. It is thrown by the `Object` `notify()` and `wait()` methods used for thread synchronization.

```
public class IllegalMonitorStateException extends RuntimeException {
    // Public Constructors
            public IllegalMonitorStateException();
            public IllegalMonitorStateException(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->RuntimeException->IllegalMonitorStateException

25.26 java.lang.IllegalStateException (JDK 1.1)

An exception of this type signals that a method has been invoked on an object that is not in an appropriate state to be able to perform the requested operation.

```
public class IllegalStateException extends RuntimeException {
    // Public Constructors
            public IllegalStateException();
            public IllegalStateException(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->RuntimeException->IllegalStateException

Extended By:

IllegalComponentStateException

25.27 java.lang.IllegalThreadStateException (JDK 1.0)

Signals that a thread is not in the appropriate state for an attempted operation to succeed.

```
public class IllegalThreadStateException extends IllegalArgumentException {
    // Public Constructors
            public IllegalThreadStateException();
            public IllegalThreadStateException(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->RuntimeException-> IllegalArgumentException->IllegalThreadStateException

25.28 java.lang.IncompatibleClassChangeError (JDK 1.0)

This is the superclass of a group of related error types. It signals some kind of illegal use of a legal class.

```
public class IncompatibleClassChangeError extends LinkageError {
    // Public Constructors
            public IncompatibleClassChangeError();
            public IncompatibleClassChangeError(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Error->LinkageError->IncompatibleClassChangeError

Extended By:

AbstractMethodError, IllegalAccessError, InstantiationError, NoSuchFieldError, NoSuchMethodError

25.29 java.lang.IndexOutOfBoundsException (JDK 1.0)

Signals that an index is out of bounds. See the subclasses `ArrayIndexOutOfBoundsException` and `StringIndexOutOfBoundsException`.

```
public class IndexOutOfBoundsException extends RuntimeException {
    // Public Constructors
            public IndexOutOfBoundsException();
            public IndexOutOfBoundsException(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->RuntimeException->IndexOutOfBoundsException

Extended By:

ArrayIndexOutOfBoundsException, StringIndexOutOfBoundsException

25.30 java.lang.InstantiationError (JDK 1.0)

Signals an attempt to instantiate an interface or abstract class.

```
public class InstantiationError extends IncompatibleClassChangeError {
    // Public Constructors
            public InstantiationError();
            public InstantiationError(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Error->LinkageError->IncompatibleClassChangeError->InstantiationError

25.31 java.lang.InstantiationException (JDK 1.0)

Signals an attempt to instantiate an interface or an abstract class.

```
public class InstantiationException extends Exception {
    // Public Constructors
            public InstantiationException();
            public InstantiationException(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->InstantiationException

Thrown By:

Class.newInstance(), Constructor.newInstance()

25.33 java.lang.InternalError (JDK 1.0)

Signals an internal error in the Java interpreter.

```
public class InternalError extends VirtualMachineError {
    // Public Constructors
            public InternalError();
            public InternalError(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Error->VirtualMachineError->InternalError

25.34 java.lang.InterruptedException (JDK 1.0)

Signals that the thread has been interrupted.

```
public class InterruptedException extends Exception {
    // Public Constructors
            public InterruptedException();
            public InterruptedException(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->InterruptedException

Thrown By:

EventQueue.getNextEvent(), MediaTracker.waitForAll(), MediaTracker.waitForID(), Object.wait(), PixelGrabber.grabPixels(), Process.waitFor(), Thread.join(), Thread.sleep()

25.35 java.lang.LinkageError (JDK 1.0)

The superclass of a group of errors that signal problems linking a class or resolving dependencies between classes.

```
public class LinkageError extends Error {
    // Public Constructors
            public LinkageError();
            public LinkageError(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Error->LinkageError

Extended By:

ClassCircularityError, ClassFormatError, ExceptionInInitializerError, IncompatibleClassChangeError, NoClassDefFoundError, UnsatisfiedLinkError, VerifyError

25.39 java.lang.NoClassDefFoundError (JDK 1.0)

Signals that the definition of a specified class could not be found.

```
public class NoClassDefFoundError extends LinkageError {
    // Public Constructors
            public NoClassDefFoundError();
            public NoClassDefFoundError(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Error->LinkageError->NoClassDefFoundError

25.40 java.lang.NoSuchFieldError (JDK 1.0)

Signals that a specified field could not be found.

```
public class NoSuchFieldError extends IncompatibleClassChangeError {
    // Public Constructors
            public NoSuchFieldError();
            public NoSuchFieldError(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Error->LinkageError->IncompatibleClassChangeError->NoSuchFieldError

25.41 java.lang.NoSuchFieldException (JDK 1.1)

This exception signals that the specified field does not exist in the specified class.

```
public class NoSuchFieldException extends Exception {
    // Public Constructors
            public NoSuchFieldException();
            public NoSuchFieldException(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->NoSuchFieldException

Thrown By:

Class.getDeclaredField(), Class.getField()

25.42 java.lang.NoSuchMethodError (JDK 1.0)

Signals that a specified method could not be found.

```
public class NoSuchMethodError extends IncompatibleClassChangeError {
    // Public Constructors
            public NoSuchMethodError();
            public NoSuchMethodError(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Error->LinkageError->IncompatibleClassChangeError->NoSuchMethodError

25.43 java.lang.NoSuchMethodException (JDK 1.0)

This exception signals that the specified method does not exist in the specified class.

```
public class NoSuchMethodException extends Exception {
    // Public Constructors
            public NoSuchMethodException();
            public NoSuchMethodException(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->NoSuchMethodException

Thrown By:

Class.getConstructor(), Class.getDeclaredConstructor(), Class.getDeclaredMethod(), Class.getMethod()

25.44 java.lang.NullPointerException (JDK 1.0)

Signals an attempt to access a field or invoke a method of a `null` object.

```
public class NullPointerException extends RuntimeException {
    // Public Constructors
            public NullPointerException();
            public NullPointerException(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->RuntimeException->NullPointerException

25.46 java.lang.NumberFormatException (JDK 1.0)

Signals an illegal number format.

```
public class NumberFormatException extends IllegalArgumentException {
    // Public Constructors
            public NumberFormatException();
            public NumberFormatException(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->RuntimeException-> IllegalArgumentException->NumberFormatException

Thrown By:

BigDecimal(), BigDecimal.valueOf(), BigInteger(), Byte(), Byte.decode(), Byte.parseByte(), Byte.valueOf(), Color.decode(), Double(), Double.valueOf(), Float(), Float.valueOf(), Integer(), Integer.decode(), Integer.parseInt(), Integer.valueOf(), Long(), Long.parseLong(), Long.valueOf(), Short(), Short.decode(), Short.parseShort(), Short.valueOf()

25.48 java.lang.OutOfMemoryError (JDK 1.0)

Signals that the interpreter has run out of memory (and that garbage collection is unable to free any memory).

```
public class OutOfMemoryError extends VirtualMachineError {
    // Public Constructors
            public OutOfMemoryError();
            public OutOfMemoryError(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Error->VirtualMachineError->OutOfMemoryError

25.52 java.lang.RuntimeException (JDK 1.0)

This exception type is not used directly, but serves as a superclass of a group of run-time exceptions that need not be declared in the `throws` clause of a method definition. These exceptions need not be declared because they are run-time conditions that can generally occur in any Java method. Thus, declaring them would be unduly burdensome, and Java does not require it.

```
public class RuntimeException extends Exception {
    // Public Constructors
            public RuntimeException();
            public RuntimeException(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->RuntimeException

Extended By:

ArithmeticException, ArrayStoreException, ClassCastException, EmptyStackException, IllegalArgumentException, IllegalMonitorStateException, IllegalStateException, IndexOutOfBoundsException, MissingResourceException, NegativeArraySizeException, NoSuchElementException, NullPointerException, SecurityException

25.53 java.lang.SecurityException (JDK 1.0)

Signals that an operation is not permitted for security reasons.

```
public class SecurityException extends RuntimeException {
    // Public Constructors
            public SecurityException();
            public SecurityException(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->RuntimeException->SecurityException

Thrown By:

Beans.setDesignTime(), Beans.setGuiAvailable(), Class.getConstructor(), Class.getConstructors(), Class.getDeclaredClasses(), Class.getDeclaredConstructor(), Class.getDeclaredConstructors(), Class.getDeclaredField(), Class.getDeclaredFields(), Class.getDeclaredMethod(), Class.getDeclaredMethods(), Class.getField(), Class.getFields(), Class.getMethod(), Class.getMethods(), ObjectInputStream.enableResolveObject(), ObjectOutputStream.enableReplaceObject()

25.56 java.lang.StackOverflowError (JDK 1.0)

Signals that a stack overflow has occurred within the Java interpreter.

```
public class StackOverflowError extends VirtualMachineError {
    // Public Constructors
            public StackOverflowError();
            public StackOverflowError(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Error->VirtualMachineError->StackOverflowError

25.59 java.lang.StringIndexOutOfBoundsException (JDK 1.0)

Signals that the index used to access a character of a `String` or `StringBuffer` is less than zero or is too large.

```
public class StringIndexOutOfBoundsException extends IndexOutOfBoundsException {
    // Public Constructors
            public StringIndexOutOfBoundsException();
            public StringIndexOutOfBoundsException(String s);
            public StringIndexOutOfBoundsException(int index);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->RuntimeException-> IndexOutOfBoundsException->StringIndexOutOfBoundsException

25.62 java.lang.ThreadDeath (JDK 1.0)

Signals that a thread should terminate. This error is thrown in a thread when the `Thread.stop()` method is called for that thread. This is an unusual `Error` type that simply causes a thread to be terminated, but does not print an error message or cause the interpreter to exit. You may catch `ThreadDeath` errors to do any necessary cleanup for a thread, but if you do, you must re-throw the error, so that the thread actually terminates.

```
public class ThreadDeath extends Error {
    // Default Constructor: public ThreadDeath()
}
```

Hierarchy:

Object->Throwable(Serializable)->Error->ThreadDeath

25.65 java.lang.UnknownError (JDK 1.0)

Signals that an unknown error has occurred at the level of the Java Virtual Machine.

```
public class UnknownError extends VirtualMachineError {
    // Public Constructors
            public UnknownError();
            public UnknownError(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Error->VirtualMachineError->UnknownError

25.66 java.lang.UnsatisfiedLinkError (JDK 1.0)

Signals that Java cannot satisfy all of the links in a class that it has loaded.

```
public class UnsatisfiedLinkError extends LinkageError {
    // Public Constructors
            public UnsatisfiedLinkError();
            public UnsatisfiedLinkError(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Error->LinkageError->UnsatisfiedLinkError

25.67 java.lang.VerifyError (JDK 1.0)

Signals that a class has not passed the byte-code verification procedures.

```
public class VerifyError extends LinkageError {
    // Public Constructors
            public VerifyError();
            public VerifyError(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Error->LinkageError->VerifyError

25.68 java.lang.VirtualMachineError (JDK 1.0)

An abstract error type that serves as superclass for a group of errors related to the Java Virtual Machine. See `InternalError`, `UnknownError`, `OutOfMemoryError`, and `StackOverflowError`.

```
public abstract class VirtualMachineError extends Error {
    // Public Constructors
            public VirtualMachineError();
            public VirtualMachineError(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Error->VirtualMachineError

Extended By:

InternalError, OutOfMemoryError, StackOverflowError, UnknownError

24.22 java.io.FileWriter (JDK 1.1)

`FileWriter` is a convenience subclass of `OutputStreamWriter` that is useful when you want to write text (as opposed to binary data) to a file. You create a `FileWriter` by specifying the file to be written to, and optionally specifying whether the data should be appended to the end of an existing file instead of overwriting that file.

The `FileWriter` class creates an internal `FileOutputStream` to write bytes to the specified file, and uses the functionality of its superclass, `OutputStreamWriter`, to convert the Unicode characters written to the stream characters into bytes using the default encoding of the default locale. (If you want to use an encoding other than the default, you cannot use `FileWriter`; in that case you must create your own `OutputStreamWriter` and `FileOutputStream`.)

Because `FileWriter` is a trivial subclass of `OutputStreamWriter`, it does not define any methods of its own, but simply inherits them from its superclass.

```
public class FileWriter extends OutputStreamWriter {
    // Public Constructors
            public FileWriter(String fileName) throws IOException;
            public FileWriter(String fileName, boolean append) throws IOException;
            public FileWriter(File file) throws IOException;
            public FileWriter(FileDescriptor fd);
}
```

Hierarchy:

Object->Writer->OutputStreamWriter->FileWriter

28.14 java.net.ProtocolException (JDK 1.0)

Signals a protocol error in the `Socket` class.

```
public class ProtocolException extends IOException {
    // Public Constructors
            public ProtocolException(String host);
            public ProtocolException();
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->IOException->ProtocolException

Thrown By:

HttpURLConnection.setRequestMethod()

28.17 java.net.SocketException (JDK 1.0)

Signals an exceptional condition while using a socket.

```
public class SocketException extends IOException {
    // Public Constructors
            public SocketException(String msg);
            public SocketException();
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->IOException->SocketException

Extended By:

BindException, ConnectException, NoRouteToHostException

Thrown By:

DatagramSocket(), DatagramSocket.getSoTimeout(), DatagramSocket.setSoTimeout(), DatagramSocketImpl.bind(), DatagramSocketImpl.create(), DatagramSocketImpl.getOption(), DatagramSocketImpl.setOption(), MulticastSocket.getInterface(), MulticastSocket.setInterface(), ServerSocket.setSoTimeout(), Socket(), Socket.getSoLinger(), Socket.getSoTimeout(), Socket.getTcpNoDelay(), Socket.setSoLinger(), Socket.setSoTimeout(), Socket.setTcpNoDelay(), SocketImpl.getOption(), SocketImpl.setOption()

19.6 java.awt.datatransfer.UnsupportedFlavorException (JDK 1.1)

An exception of this type signals that a `Transferable` object could not provide data in the requested format.

```
public class UnsupportedFlavorException extends Exception {
    // Public Constructor
            public UnsupportedFlavorException(DataFlavor flavor);
}
```

Hierarchy:

`Object->Throwable(Serializable)->Exception->UnsupportedFlavorException`

Thrown By:

`StringSelection.getTransferData(), Transferable.getTransferData()`

22.13 java.awt.peer.LightweightPeer (JDK 1.1)

```
public interface LightweightPeer extends ComponentPeer {
}
```

Returned By:

Toolkit.createComponent()

22.19 java.awt.peer.PanelPeer (JDK 1.0)

```
public interface PanelPeer extends ContainerPeer {
}
```

Hierarchy:

(PanelPeer(ContainerPeer(ComponentPeer)))

Returned By:

Toolkit.createPanel()

31.6 java.util.zip.DataFormatException (JDK 1.1)

Signals that invalid or corrupt data has been encountered while uncompressing data.

```
public class DataFormatException extends Exception {
    // Public Constructors
            public DataFormatException();
            public DataFormatException(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->DataFormatException

Thrown By:

Inflater.inflate()

31.14 java.util.zip.ZipException (JDK 1.1)

Signals that an error has occurred in reading or writing a ZIP file.

```
public class ZipException extends IOException {
    // Public Constructors
            public ZipException();
            public ZipException(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->IOException->ZipException

Thrown By:

ZipFile()

24.9 java.io.CharConversionException (JDK 1.1)

This exception signals an error when converting bytes to characters or vice versa.

```
public class CharConversionException extends IOException {
    // Public Constructors
            public CharConversionException();
            public CharConversionException(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->IOException->CharConversionException

24.14 java.io.EOFException (JDK 1.0)

An `IOException` that signals the end-of-file.

```
public class EOFException extends IOException {
    // Public Constructors
            public EOFException();
            public EOFException(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->IOException->EOFException

24.19 java.io.FileNotFoundException (JDK 1.0)

An `IOException` that signals that a specified file cannot be found.

```
public class FileNotFoundException extends IOException {
    // Public Constructors
            public FileNotFoundException();
            public FileNotFoundException(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->IOException->FileNotFoundException

Thrown By:

FileInputStream(), FileReader()

24.21 java.io.FileReader (JDK 1.1)

`FileReader` is a convenience subclass of `InputStreamReader` that is useful when you want to read text (as opposed to binary data) from a file. You create a `FileReader` by specifying the file to be read, in any of three possible forms. The `FileReader` constructor internally creates a `FileInputStream` to read bytes from the specified file, and uses the functionality of its superclass, `InputStreamReader`, to convert those bytes from characters in the local encoding to the Unicode characters used by Java.

Because `FileReader` is a trivial subclass of `InputStreamReader`, it does not define any `read()` methods or other methods of its own. Instead, it inherits all its methods from its superclass.

If you want to read Unicode characters from a file that uses some encoding other than the default encoding for the locale, you must explicitly create your own `InputStreamReader` to perform the byte-to-character conversion.

```
public class FileReader extends InputStreamReader {
    // Public Constructors
            public FileReader(String fileName) throws FileNotFoundException;
            public FileReader(File file) throws FileNotFoundException;
            public FileReader(FileDescriptor fd);
}
```

Hierarchy:

Object->Reader->InputStreamReader->FileReader

24.32 java.io.InvalidObjectException (JDK 1.1)

This exception should be thrown by the `validateObject()` method of an object that implements the `ObjectInputValidation` interface when a deserialized object fails an input validation test for any reason.

```
public class InvalidObjectException extends ObjectStreamException {
    // Public Constructor
            public InvalidObjectException(String reason);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->IOException->ObjectStreamException->InvalidObjectException

Thrown By:

ObjectInputStream.registerValidation(), ObjectInputValidation.validateObject()

24.33 java.io.IOException (JDK 1.0)

Signals that an exceptional condition has occurred during input or output. This class has several more specific subclasses. See `EOFException`, `FileNotFoundException`, `InterruptedIOException`, and `UTFDataFormatException`.

```
public class IOException extends Exception {
    // Public Constructors
            public IOException();
            public IOException(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->IOException

Extended By:

CharConversionException, EOFException, FileNotFoundException, InterruptedIOException, MalformedURLException, ObjectStreamException, ProtocolException, SocketException, SyncFailedException, UnknownHostException, UnknownServiceException, UnsupportedEncodingException, UTFDataFormatException, ZipException

Thrown By:

`Many methods`

java.io.InvalidObjectException (JDK 1.1) | BOOK INDEX | java.io.LineNumberInputStream (JDK 1.0; Deprecated.)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

24.36 java.io.NotActiveException (JDK 1.1)

This exception is thrown in several circumstances. It indicates that the invoked method was not invoked at the right time or in the correct context. Typically it means that an `ObjectOutputStream` or `ObjectInputStream` is not currently active, and therefore the requested operation could not be performed.

```
public class NotActiveException extends ObjectStreamException {
    // Public Constructors
            public NotActiveException(String reason);
            public NotActiveException();
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->IOException->ObjectStreamException->NotActiveException

Thrown By:

ObjectInputStream.defaultReadObject(), ObjectInputStream.registerValidation()

24.37 java.io.NotSerializableException (JDK 1.1)

This exception signals that an object could not be serialized. It is thrown when serialization is attempted on an instance of a class that does not implement the `Serializable` interface. Note that it is also thrown when an attempt is made to serialize a `Serializable` object that refers to (or "contains") an object that is not `Serializable`.

A subclass of a class that is `Serializable` can prevent itself from being serialized by throwing this exception from its `writeObject()` and/or `readObject()` methods.

```
public class NotSerializableException extends ObjectStreamException {
    // Public Constructors
            public NotSerializableException(String classname);
            public NotSerializableException();
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->IOException->ObjectStreamException->NotSerializableException

24.44 java.io.ObjectStreamException (JDK 1.1)

This class is the superclass of a number of more specific exception types that may be raised in the process of serializing and deserializing objects with the `ObjectOutputStream` and `ObjectInputStream` classes.

```
public abstract class ObjectStreamException extends IOException {
    // Protected Constructors
            protected ObjectStreamException(String classname);
            protected ObjectStreamException();
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->IOException->ObjectStreamException

Extended By:

InvalidClassException, InvalidObjectException, NotActiveException, NotSerializableException, OptionalDataException, StreamCorruptedException, WriteAbortedException

24.59 java.io.Serializable (JDK 1.1)

The `Serializable` interface defines no methods or constants. A class should implement this interface simply to indicate that it allows itself to be serialized and deserialized with `ObjectOutputStream.writeObject()` and `ObjectInputStream.readObject()`.

Objects that need special handling during serialization or deserialization may implement one or both of the following methods. Note, however, that these methods are not part of the `Serializable` interface:

```
private void writeObject(java.io.ObjectOutputStream out)
                  throws IOException
private void readObject(java.io.ObjectInputStream in)
                    throws IOException, ClassNotFoundException;
```

Typically the `writeObject()` method performs any necessary cleanup or preparation for serialization, invokes the `defaultWriteObject()` method of the `ObjectOutputStream` to serialize the non-`transient` fields of the class, and then optionally writes any additional data that are required. Similarly, the `readObject()` method typically invokes the `defaultReadObject()` method of the `ObjectInputStream`, reads any additional data written by the corresponding `writeObject()` method, and finally performs any extra initialization required by the object. The `readObject()` method may also register an `ObjectInputValidation` object to validate the object once it is completely deserialized.

```
public interface Serializable {
}
```

Extended By:

Externalizable

Implemented By:

BitSet, Boolean, BorderLayout, BreakIterator, Calendar, CardLayout, Character, CheckboxGroup, Class, Collator, Color, Component, Cursor, Date, DateFormatSymbols, DecimalFormatSymbols, Dimension, Event, EventObject, File, FlowLayout, Font, FontMetrics, Format, GridBagConstraints, GridBagLayout, GridLayout, Hashtable, InetAddress, Insets, Locale, MediaTracker, MenuComponent, MenuShortcut, Number, ObjectStreamClass, Point, Polygon, PropertyChangeSupport, Random, Rectangle, String, StringBuffer, SystemColor, Throwable, TimeZone, URL, Vector, VetoableChangeSupport

PREVIOUS | HOME | NEXT
java.io.SequenceInputStream (JDK 1.0) | BOOK INDEX | java.io.StreamCorruptedException (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

24.60 java.io.StreamCorruptedException (JDK 1.1)

This exception signals that the data stream being read by an `ObjectInputStream` has been corrupted and does not contain valid serialized object data.

```
public class StreamCorruptedException extends ObjectStreamException {
    // Public Constructors
            public StreamCorruptedException(String reason);
            public StreamCorruptedException();
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->IOException->ObjectStreamException->StreamCorruptedException

Thrown By:

ObjectInputStream(), ObjectInputStream.readStreamHeader()

24.65 java.io.SyncFailedException (JDK 1.1)

Signals that a call to `FileDescriptor.sync()` did not complete successfully.

```
public class SyncFailedException extends IOException {
    // Public Constructor
            public SyncFailedException(String desc);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->IOException->SyncFailedException

Thrown By:

FileDescriptor.sync()

24.66 java.io.UnsupportedEncodingException (JDK 1.1)

This exception signals that a requested character encoding is not supported by the current Java Virtual Machine.

```
public class UnsupportedEncodingException extends IOException {
    // Public Constructors
            public UnsupportedEncodingException();
            public UnsupportedEncodingException(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->IOException->UnsupportedEncodingException

Thrown By:

ByteArrayOutputStream.toString(), InputStreamReader(), OutputStreamWriter(), String(), String.getBytes()

24.67 java.io.UTFDataFormatException (JDK 1.0)

An IOException that signals that a malformed UTF-8 string has been encountered by a class that implements the DataInput interface. UTF-8 is an ASCII-compatible "transformation format" for Unicode characters that is often used to store and transmit Unicode text.

```
public class UTFDataFormatException extends IOException {
    // Public Constructors
            public UTFDataFormatException();
            public UTFDataFormatException(String s);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->IOException->UTFDataFormatException

11. Internationalization

Contents:

Internationalization is the process of making a program flexible enough to run correctly in any locale, as discussed in Chapter 4, *What's New in Java 1.1*. The required corollary to internationalization is localization--the process of arranging for a program to run in a specific locale.

There are several distinct steps to the task of internationalization. Java 1.1 addresses these steps with several different mechanisms:

- A program must be able to read, write, and manipulate localized text. Java uses the Unicode character encoding, which by itself is a huge step towards internationalization. In addition, in Java 1.1 the `InputStreamReader` and `OutputStreamWriter` classes convert text from a locale-specific encoding to Unicode and from Unicode to a locale-specific encoding, respectively.

- A program must conform to local customs when displaying dates and times, formatting numbers, and sorting strings. Java addresses these issues with the classes in the new `java.text` package.

- A program must display all user-visible text in the local language. Translating the messages a program displays is always one of the main tasks in localizing a program. A more important task is writing the program so that all user-visible text is fetched at runtime, rather than hard-coded directly into the program. Java 1.1 facilitates this process with the `ResourceBundle` class and its subclasses in the `java.util` package.

This chapter discusses all three of these aspects of internationalization.

11.1 A Word About Locales

A locale represents a geographic, political, or cultural region. In Java 1.1, locales are represented by the `java.util.Locale` class. A locale is frequently defined by a language, which is represented by its standard lowercase two-letter code, such as en (English) or fr (French). Sometimes, however, language alone is not sufficient to uniquely specify a locale, and a country is added to the specification. A country is represented by an uppercase two-letter code. For example, the United States English locale (en_US) is distinct from the British English locale (en_GB), and the French spoken in Canada (fr_CA) is different from the French spoken in France (fr_FR). Occasionally, the scope of a locale is further narrowed with the addition of a system-dependent "variant" string.

The `Locale` class maintains a static default locale, which can be set and queried with `Locale.setDefault()` and `Locale.getDefault()`. Locale-sensitive methods in Java 1.1 typically come in two forms. One uses the default locale and the other uses a `Locale` object that is explicitly specified as an argument. A program can create and use any number of non-default `Locale` objects, although it is more common simply to rely on the default locale, which is inherited from the underlying default locale on the native platform. Locale-sensitive classes in Java often provide a method to query the list of locales that they support.

Finally, note that AWT components in Java 1.1 have a locale property, so it is possible for different components to use different locales. (Most components, however, are not locale-sensitive; they behave the same in any locale.)

23.8 java.beans.IntrospectionException (JDK 1.1)

Signals that introspection on a Java bean could not be completed. Typically this indicates a bug in the way the bean or its associated `BeanInfo` class is defined.

```
public class IntrospectionException extends Exception {
    // Public Constructor
            public IntrospectionException(String mess);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->IntrospectionException

Thrown By:

EventSetDescriptor(), IndexedPropertyDescriptor(), Introspector.getBeanInfo(), PropertyDescriptor()

23.11 java.beans.ParameterDescriptor (JDK 1.1)

A `ParameterDescriptor` object is a type of `FeatureDescriptor` that describes an argument or parameter to a method of a Java bean. The `BeanInfo` class for a Java bean optionally creates `ParameterDescriptor` objects that describe the parameters of the methods that the bean exports. While the `BeanInfo` class creates and initializes `ParameterDescriptor` objects, it is typically only application builders and similar tools that use these objects to obtain information about method parameters supported by the bean.

The `ParameterDescriptor` class is a trivial subclass of `FeatureDescriptor`, and does not provide any new methods. Thus, you should use the methods of `FeatureDescriptor` to provide information about method parameters.

```
public class ParameterDescriptor extends FeatureDescriptor {
    // Default Constructor: public ParameterDescriptor()
}
```

Hierarchy:

Object->FeatureDescriptor->ParameterDescriptor

Passed To:

MethodDescriptor()

Returned By:

MethodDescriptor.getParameterDescriptors()

PREVIOUS java.beans.MethodDescriptor (JDK 1.1)

HOME BOOK INDEX

NEXT java.beans.PropertyChangeEvent (JDK 1.1)

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

28.2 java.net.ConnectException (JDK 1.1)

This exception signals that a socket could not be connected to a remote address and port. This means that the remote host could be reached, but is not responding, perhaps because there is no process on that host that is listening on the specified port.

```
public class ConnectException extends SocketException {
    // Public Constructors
            public ConnectException(String msg);
            public ConnectException();
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->IOException->SocketException->ConnectException

28.11 java.net.MalformedURLException (JDK 1.0)

Signals that an unparseable URL specification has been passed to a method.

```
public class MalformedURLException extends IOException {
    // Public Constructors
            public MalformedURLException();
            public MalformedURLException(String msg);
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->IOException->MalformedURLException

Thrown By:

URL(default.htm)

28.13 java.net.NoRouteToHostException (JDK 1.1)

This exception signals that a socket could not be connected to a remote host, because the host could not be contacted. Typically this means that some link in the network between the local machine and the remote host is down, or that the host is behind a firewall.

```
public class NoRouteToHostException extends SocketException {
    // Public Constructors
            public NoRouteToHostException(String msg);
            public NoRouteToHostException();
}
```

Hierarchy:

Object->Throwable(Serializable)->Exception->IOException->SocketException->NoRouteToHostException

22.2 java.awt.peer.CanvasPeer (JDK 1.0)

```
public interface CanvasPeer extends ComponentPeer {
}
```

Returned By:

Toolkit.createCanvas()

22.10 java.awt.peer.FontPeer (JDK 1.1)

```
public interface FontPeer {
}
```

Returned By:

Font.getPeer(), Toolkit.getFontPeer()

10.9 Naming Patterns and Conventions

As we've seen, beanbox programs may rely on introspection of a bean to determine the list of properties, events, and methods it supports. In order for this to work, bean developers must follow a set of standard naming conventions, sometimes referred to as JavaBeans "design patterns." These patterns specify, for example, that the getter and setter accessor methods for a property should begin with `get` and `set`. Not all of the patterns are absolute requirements. If a bean has accessor methods with different names, it is possible to use a `PropertyDescriptor` object, specified in a `BeanInfo` class, to specify the accessor methods for the property. Note, however, that although an accessor method name is not required to follow the pattern, the method is required to have the exact type signature specified by the pattern.

This section lists the design patterns for bean properties, events, and methods. It also lists other conventions and requirements that you should keep in mind when developing beans.

Java Bean Patterns

Beans

class name:

Any

superclass:

Any

constructor:

Must have a no-argument constructor, or a serialized template file

packaging:

JAR file manifest entry specifies `Java-Bean: True`

Properties (property *p* of type *T*)

getter:

```
public T getP()
```

setter:

```
public void setP(T value)
```

Boolean Properties (property *p* of type boolean)

getter:

```
public boolean getP()
```

(boolean value)

or `public boolean is P()` setter:

```
public void setP
```

Indexed Properties (property *p* of type *T*[])

array getter:

```
public T[] getP()
```

array setter:

```
public void setP(T[] value)
```

element getter:

```
public T getP(int index)
```

element setter:

```
public void setP(int index, T value)
```

Bound Properties (property *p* of type *T*)

getter:

Same as regular property

setter:

Same as regular property

listeners:

One event listener list for all bound properties of a bean

listener registration:

```
public void addPropertyChangeListener (PropertyChangeListener l)
```

listener removal:

```
public void removePropertyChangeListener (PropertyChangeListener l)
```

Constrained Properties (property *p* of type *T*)

getter:

Same as regular property

setter:

```
public void setP(T value)
                                      throws PropertyVetoException
```

listeners:

One event listener list for all constrained properties of a bean

listener registration:

```
public void addVetoableChangeListener (VetoableChangeListener l)
```

listener removal:

```
public void removeVetoableChangeListener
            (VetoableChangeListener l)
```

Events (event named *E*)

event class name:

```
EEvent
```

listener name:

```
EListener
```

listener methods:

```
public void methodname(EEvent e)
```

listener registration:

```
public void addEListener(EListener l)
```

listener removal:

```
public void removeEListener(EListener l)
```

Unicast Events (event named *E* only one listener allowed)

listener registration:

```
public void addEListener(EListener l)
       throws TooManyListenersException
```

listener removal:

```
public void removeEListener(EListener l)
```

Methods

method name:

Any

method args:

Any; some tools only recognize no-argument methods

BeanInfo Classes (for bean *B*)

class name:

`*B*BeanInfo`

Property Editors (for properties of type *T*)

class name:

`*T*Editor`

constructor:

Must have a no-argument constructor

Property Editors (for individual properties)

class name:

Any; register via *PropertyDescriptor*

constructor:

Must have a no-argument constructor

Customizers (for bean *B*)

class name:

Any; register via *BeanDescriptor* (*BCustomizer* by convention)

superclass:

Must be a component; typically extends *Panel* *constructor:*

Must have a no-argument constructor

Documentation File (for bean *B*)

default docs:

`B.html`

localized docs:

`locale/B.html`

11.2 Unicode

Java uses the Unicode character encoding. Java 1.0 used Unicode version 1.1, while Java 1.1 has adopted the newer Unicode 2.0 standard. Unicode is a 16-bit character encoding established by the Unicode Consortium, which describes the standard as follows (see http://unicode.org):

> The Unicode Worldwide Character Standard is a character coding system designed to support the interchange, processing, and display of the written texts of the diverse languages of the modern world. In addition, it supports classical and historical texts of many written languages.
>
> In its current version (2.0), the Unicode standard contains 38,885 distinct coded characters derived from 25 supported scripts. These characters cover the principal written languages of the Americas, Europe, the Middle East, Africa, India, Asia, and Pacifica.

In the canonical form of the Unicode encoding, which is what Java `char` and `String` types use, every character occupies two bytes. The Unicode characters \u0020 to \u007E are equivalent to the ASCII and ISO8859-1 (Latin-1) characters 0x20 through 0x7E. The Unicode characters \u00A0 to \u00FF are identical to the ISO8859-1 characters 0xA0 to 0xFF. Thus there is a trivial mapping between Latin-1 and Unicode characters. A number of other portions of the Unicode encoding are based on pre-existing standards, such as ISO8859-5 (Cyrillic) and ISO8859-8 (Hebrew), though the mappings between these standards and Unicode may not be as trivial as the Latin-1 mapping.

Note that Unicode support is quite limited on many platforms. One of the difficulties with the use of Unicode is the poor availability of fonts to display all of the Unicode characters. Figure 11.1 shows the characters that are available on a typical configuration of the U.S. English Windows 95 platform. Note the special box glyph used to indicate undefined characters.

Figure 11.1: Some Unicode characters and their encodings

[Graphic: Figure 11-1]

Unicode is similar to, but not the same as, ISO 10646, the UCS (Universal Character Set) encoding. UCS is a 2- or 4-byte encoding originally intended to contain all national standard character encodings. For example, it was to include the separate Chinese, Japanese, Korean, and Vietnamese encodings for Han ideographic characters. Unicode, in contrast, "unifies" these disparate encodings into a single set of Han characters that work for all four countries. Unicode has been so successful, however, that ISO 10646 has adopted it in place of non-unified encodings. Thus, ISO 10646 is effectively Unicode, with the option of two extra bytes for expansion purposes.

Unicode is a trademark of the Unicode Consortium. Version 2.0 of the standard is defined by the book *The Unicode Standard, Version 2.0* (published by Addison-Wesley, ISBN 0-201-48345-9). Further information about the Unicode standard and the Unicode Consortium can be obtained at http://unicode.org/.

Table 11.1 provides an overview of the Unicode 2.0 encoding.

Table 11.1: Outline of the Unicode 2.0 Encoding

Start	End	Description
0000	1FFF	**Alphabets**
0000	007F	Basic Latin
0080	00FF	Latin-1 Supplement
0100	017F	Latin Extended-A
0180	024F	Latin Extended-B
0250	02AF	IPA Extensions
02B0	02FF	Spacing Modifier Letters
0300	036F	Combining Diacritical Marks
0370	03FF	Greek
0400	04FF	Cyrillic
0530	058F	Armenian
0590	05FF	Hebrew
0600	06FF	Arabic
0900	097F	Devanagari
0980	09FF	Bengali
0A00	0A7F	Gurmukhi
0A80	0AFF	Gujarati
0B00	0B7F	Oriya
0B80	0BFF	Tamil
0C00	0C7F	Telugu
0C80	0CFF	Kannada
0D00	0D7F	Malayalam
0E00	0E7F	Thai
0E80	0EFF	Lao
0F00	0FBF	Tibetan
10A0	10FF	Georgian
1100	11FF	Hangul Jamo

1E00	1EFF	Latin Extended Additional
1F00	1FFF	Greek Extended
2000	2FFF	**Symbols and Punctuation**
2000	206F	General Punctuation
2070	209F	Superscripts and Subscripts
20A0	20CF	Currency Symbols
20D0	20FF	Combining Marks for Symbols
2100	214F	Letterlike Symbols
2150	218F	Number Forms
2190	21FF	Arrows
2200	22FF	Mathematical Operators
2300	23FF	Miscellaneous Technical
2400	243F	Control Pictures
2440	245F	Optical Character Recognition
2460	24FF	Enclosed Alphanumerics
2500	257F	Box Drawing
2580	259F	Block Elements
25A0	25FF	Geometric Shapes
2600	26FF	Miscellaneous Symbols
2700	27BF	Dingbats
3000	33FF	**CJK Auxiliary**
3000	303F	CJK Symbols and Punctuation
3040	309F	Hiragana
30A0	30FF	Katakana
3100	312F	Bopomofo
3130	318F	Hangul Compatibility Jamo
3190	319F	Kanbun
3200	32FF	Enclosed CJK Letters and Months
3300	33FF	CJK Compatibility
4E00	9FFF	**CJK Unified Ideographs** Han characters used in China, Japan, Korea, Taiwan, and Vietnam
AC00	D7A3	**Hangul Syllables**

D800	DFFF	**Surrogates**
D800	DB7F	High Surrogates
DB80	DBFF	High Private Use Surrogates
DC00	DFFF	Low Surrogates
E000	F8FF	**Private Use**
F900	FFFF	**Miscellaneous**
F900	FAFF	CJK Compatibility Ideographs
FB00	FB4F	Alphabetic Presentation Forms
FB50	FDFF	Arabic Presentation Forms-A
FE20	FE2F	Combining Half Marks
FE30	FE4F	CJK Compatibility Forms
FE50	FE6F	Small Form Variants
FE70	FEFE	Arabic Presentation Forms-B
FEFF	FEFF	Specials
FF00	FFEF	Halfwidth and Fullwidth Forms
FFF0	FFFF	Specials

Unicode and Local Encodings

While Java programs use Unicode text internally, Unicode is not the customary character encoding for most countries or locales. Thus, an important requirement for Java programs is to be able to convert text from the local encoding to Unicode as it is read (from a file or network, for example) and to be able to convert text from Unicode to the local encoding as it is written. In Java 1.0, this requirement is not well supported. In Java 1.1, however, the conversion can be done with the `java.io.InputStreamReader` and `java.io.OutputStreamWriter` classes, respectively. These classes load an appropriate `ByteToCharConverter` or `CharToByteConverter` class to perform the conversion. Note that these converter classes are part of the `sun.io` package and are not for public use (although an explicit conversion interface may be defined in a later release of Java).

The UTF-8 Encoding

The canonical two-bytes per character encoding is useful for the manipulation of character data and is the internal representation used throughout Java. However, because a large amount of text used by Java programs is 8-bit text, and because there are so many existing computer systems that support only 8-bit characters, the 16-bit canonical form is usually not the most efficient way to store Unicode text nor the most portable way to transmit it.

Because of this, other encodings called "transformation formats" have been developed. Java provides simple support for the UTF-8 encoding with the `DataInputStream.readUTF()` and `DataOutputStream.writeUTF()` methods. UTF-8 is a variable-width or "multi-byte" encoding format; this means that different characters require different numbers of bytes. In UTF-8, the standard ASCII characters occupy only one byte, and remain untouched by the encoding (i.e., a string of ASCII characters is a legal UTF-8 string). As a tradeoff, however, other Unicode characters occupy two or three bytes.

In UTF-8, Unicode characters between `\u0000` and `\u007F` occupy a single byte, which has a value of between `0x00` and `0x7F`, and which always has its high-order bit set to `0`. Characters between `\u0080` and `\u07FF` occupy two bytes, and characters between `\u0800` and `\uFFFF` occupy three bytes. The first byte of a two-byte character always has high-order bits `110`, and the first byte of a three-byte character always has high-order bits `1110`. Since single-byte characters always have 0 as their high-order bit, the one-, two-, and three-byte characters can easily be distinguished from each other.

The second and third bytes of two- and three-byte characters always have high-order bits `10`, which distinguishes them from one-byte characters, and also distinguishes them from the first byte of a two- or three-byte sequence. This is important because it allows a program to locate the start of a character in a multi-byte sequence.

The remaining bits in each character (i.e., the bits that are not part of one of the required high-order bit sequences) are used to encode the actual Unicode character data. In the single-byte form, there are seven bits available, suitable for encoding characters up to `\u007F`. In the two-byte form, there are 11 data bits available, which is enough to encode values to `\u07FF`, and in the three-byte form there are 16 available data bits, which is enough to encode all 16-bit Unicode characters. Table 11.2 summarizes the UTF-8 encoding.

Table 11.2: The UTF-8 Encoding

Start Character	**End Character**	**Required Data Bits**	**Binary Byte Sequence (`x` = data bits)**
\u0000	\u007F	7	0xxxxxxx
\u0080	\u07FF	11	110xxxxx 10xxxxxx
\u0800	\uFFFF	16	1110xxxx 10xxxxxx 10xxxxxx

The UTF-8 has the following desirable features:

- All ASCII characters are one-byte UTF-8 characters. A legal ASCII string is a legal UTF-8 string.
- Any non-ASCII character (i.e., any character with the high-order bit set) is part of a multi-byte

character.

- The first byte of any UTF-8 character indicates the number of additional bytes in the character.
- The first byte of a multi-byte character is easily distinguished from the subsequent bytes. Thus, it is easy to locate the start of a character from an arbitrary position in a data stream.
- It is easy to convert between UTF-8 and Unicode.
- The UTF-8 encoding is relatively compact. For text with a large percentage of ASCII characters, it is more compact than Unicode. In the worst case, a UTF-8 string is only 50% larger than the corresponding Unicode string.

Java actually uses a slightly modified form of UTF-8. The Unicode character `\u0000` is encoded using a two-byte sequence, so that an encoded Unicode string never contains null characters.

11.3 Character Encodings

Text representation has traditionally been one of the most difficult problems of internationalization. Java 1.1, however, solves this problem quite elegantly and hides the difficult issues. Java uses Unicode internally, so it can represent essentially any character in any commonly used written language. As noted above, the remaining task is to convert Unicode to and from locale-specific encodings. Java 1.1 includes quite a few internal "byte-to-char" converters and "char-to-byte" converters that handle converting locale-specific character encodings to Unicode and vice versa. While the converters themselves are not public, they are accessed through the `InputStreamReader` and `OutputStreamWriter` classes, which are two of the new character streams included in the `java.io` package.

Any program can automatically handle locale-specific encodings simply by using these new character stream classes to do their textual input and output. (And in addition to automatic encoding conversion, the program also benefits from the greatly improved efficiency of these new classes over the byte streams of Java 1.0.)

Example 11.1 shows a simple program that works with character encodings. It converts a file from one specified encoding to another by converting from the first encoding to Unicode and then from Unicode to the second encoding. Note that most of the program is taken up with the mechanics of parsing argument lists, handling exceptions, and so on. Only a few lines are required to create the `InputStreamReader` and `OutputStreamWriter` classes that perform the two halves of the conversion. Also note that exceptions are handled by calling `LocalizedError.display()`. This method is not part of the Java 1.1 API; it is a custom method shown in Example 11.6 at the end of this chapter.

Example 11.1: Working with Character Encodings

```
import java.io.*;
/** A program to convert from one character encoding to another. */
public class ConvertEncoding {
  public static void main(String[] args) {
    String from = null, to = null;
    String infile = null, outfile = null;
    for(int i = 0; i < args.length; i++) {  // Parse command-line arguments.
      if (i == args.length-1) usage();      // All legal args require another.
      if (args[i].equals("-from")) from = args[++i];
      else if (args[i].equals("-to")) to = args[++i];
      else if (args[i].equals("-in")) infile = args[++i];
      else if (args[i].equals("-out")) outfile = args[++i];
      else usage();
    }
    try { convert(infile, outfile, from, to); }  // Attempt conversion.
    catch (Exception e) {                        // Handle possible exceptions.
      LocalizedError.display(e);  // Defined at the end of this chapter.
      System.exit(1);
    }
  }
  public static void usage() {
    System.err.println("Usage: java ConvertEncoding <options>\n" +
                       "Options:\n\t-from <encoding>\n\t-to <encoding>\n\t" +
                       "-in <file>\n\t-out <file>");
    System.exit(1);
  }
  public static void convert(String infile, String outfile,
                             String from, String to)
       throws IOException, UnsupportedEncodingException
  {
    // Set up byte streams.
    InputStream in;
    if (infile != null) in = new FileInputStream(infile);
    else in = System.in;
    OutputStream out;
    if (outfile != null) out = new FileOutputStream(outfile);
    else out = System.out;
    // Use default encoding if no encoding is specified.
    if (from == null) from = System.getProperty("file.encoding");
    if (to == null) to = System.getProperty("file.encoding");
    // Set up character streams.
    Reader r = new BufferedReader(new InputStreamReader(in, from));
    Writer w = new BufferedWriter(new OutputStreamWriter(out, to));
    // Copy characters from input to output.  The InputStreamReader converts
    // from the input encoding to Unicode, and the OutputStreamWriter converts
    // from Unicode to the output encoding.  Characters that cannot be
    // represented in the output encoding are output as '?'
    char[] buffer = new char[4096];
    int len;
    while((len = r.read(buffer)) != -1)  // Read a block of input.
      w.write(buffer, 0, len);           // And write it out.
    r.close();                           // Close the input.
    w.flush();                           // Flush and close output.
    w.close();
  }
}
```

11.4 Handling Local Customs

The second problem of internationalization is the task of following local customs and conventions in areas like date and time formatting. The `java.text` package defines classes to help with this duty.

The `NumberFormat` class is used to format numbers, monetary amounts, and percentages in a locale-dependent way for display to the user. This is necessary because different locales have different conventions for number formatting. For example, in France, a comma is used as a decimal separator instead of a period, as used in many English speaking countries. A `NumberFormat` object can use the default locale or any locale you specify.

The `DateFormat` class is used to format dates and times in a locale-dependent way for display to the user. Different countries have different conventions. Should the month or day be displayed first? Should periods or colons be used to separate fields of the time? What are the names of the months in the language of the locale? A `DateFormat` object can simply use the default locale or it can use any locale you specify. The `DateFormat` class is used in conjunction with the `TimeZone` and `Calendar` classes of `java.util`. The `TimeZone` object tells the `DateFormat` what time zone the date should be interpreted in, while the `Calendar` object specifies how the date itself should be broken down into days, weeks, months, and years. Almost all locales use the standard `GregorianCalendar`.

The `Collator` class is used to compare strings in a locale-dependent way. This is necessary because different languages "alphabetize" strings in different ways (and some languages don't even use alphabets). In traditional Spanish, for example, the letters "ch" are treated as a single character that comes between "c" and "d" for the purposes of sorting. When you need to sort strings or search for a string within Unicode text, you should use a `Collator` object, either one created to work with the default locale or one created for a specified locale.

The `BreakIterator` class allows you to locate character, word, line, and sentence boundaries in a locale-dependent way. This is useful when you need to recognize such boundaries in Unicode text, as when you are implementing a word-wrapping algorithm.

Example 11.3 shows a class that uses the `NumberFormat` and `DateFormat` classes to display a hypothetical stock portfolio to the user following local conventions. The program uses various `NumberFormat` and `DateFormat` objects to format (using the `format()` method) different types of numbers and dates. These `Format` objects all operate using the default locale, but could have been created with an explicitly specified locale. Example 11.2 shows the output of this program in American, Canadian, and French locales. Note the different treatment of dates, numbers, and monetary quantities in these three locales.

Example 11.2: Dates and Numbers Formatted for Three Locales

```
# American English locale (en_US)
Portfolio value at April 08, 1997 6:57:40 PM PDT:
Symbol  Shares  Bought On       At      Quote   Change
XXX     400     6/15/96         $11.90  $13.00  9%
YYY     1,100   9/14/96         $71.09  $27.25  -61%
ZZZ     6,000   6/27/91         $23.37  $89.12  281%
# Canadian English locale (en_CA)
Portfolio value at April 8, 1997 9:57:40 CDT PM:
Symbol  Shares  Bought On       At      Quote   Change
XXX     400     15/06/96        $11.90  $13.00  9%
YYY     1,100   14/09/96        $71.09  $27.25  -61%
ZZZ     6,000   27/06/91        $23.37  $89.12  281%
# French locale (fr_FR)
Portfolio value at 9 avril 1997 03:57:40 GMT+02:00:
Symbol  Shares  Bought On       At      Quote   Change
XXX     400     15/06/96        11,90 F 13,00 F 9%
YYY     1 100   14/09/96        71,09 F 27,25 F -61%
ZZZ     6 000   27/06/91        23,37 F 89,12 F 281%
```

Example 11.3: Formatting Dates and Numbers in an Internationalized Program

```
import java.text.*;
import java.util.Date;
/**
 * A partial implementation of a hypothetical stock portfolio class.
 * We use it only to demonstrate number and date internationalization.
 */
public class Portfolio {
  EquityPosition[] positions;
  Date lastQuoteTime = new Date();
  public void print() {
    // Obtain NumberFormat and DateFormat objects to format our data.
    NumberFormat number = NumberFormat.getInstance();
    NumberFormat price = NumberFormat.getCurrencyInstance();
    NumberFormat percent = NumberFormat.getPercentInstance();
    DateFormat shortdate = DateFormat.getDateInstance(DateFormat.SHORT);
    DateFormat fulldate = DateFormat.getDateTimeInstance(DateFormat.LONG,
                                                         DateFormat.LONG);
    // Print some introductory data.
    System.out.println("Portfolio value at " +
                       fulldate.format(lastQuoteTime) + ":");
    System.out.println("Symbol\tShares\tBought On\tAt\t" +
                       "Quote\tChange");
    // Then display the table using the format() methods of the Format objects.
    for(int i = 0; i < positions.length; i++) {
      System.out.print(positions[i].name + "\t");
      System.out.print(number.format(positions[i].shares) + "\t");
      System.out.print(shortdate.format(positions[i].purchased) + "\t");
      System.out.print(price.format(positions[i].bought) + "\t");
      System.out.print(price.format(positions[i].current) + "\t");
      double change =
        (positions[i].current - positions[i].bought)/positions[i].bought;
      System.out.println(percent.format(change));
    }
  }
  static class EquityPosition {
    String name;             // Name of the stock.
    int shares;              // Number of shares held.
    Date purchased;          // When purchased.
    double bought, current;  // Purchase price and current price (per share).
    EquityPosition(String n, int s, Date when, double then, double now) {
      name = n; shares = s; purchased = when; bought = then; current = now;
    }
  }
}
```

PREVIOUS | HOME | NEXT
Character Encodings | BOOK INDEX | Localizing User-Visible Messages

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

10.2 A Simple Bean

As noted above, the AWT components in Java 1.1 can all function as beans. When you write a custom GUI component, it is not difficult to make it function as a bean as well. Example 10.1 shows the definition of a custom component, `MultiLineLabel`, that is also a bean. This component is like the standard `Label` component, but it can display labels that contain more than one line of text.

Much of the code in this example is taken up with the mechanics of breaking the label into separate lines and measuring the size of each of those lines. From the standpoint of custom AWT components, however, the most important code is in several required methods that make any component work correctly. First, there is the `paint()` method, of course. All components (including applets) use this method to display themselves on the screen. Second, the `getPreferredSize()` and `getMinimumSize()` methods return the preferred and minimum sizes for the component. These methods must be implemented so that layout managers can correctly lay the component out. (Note, though, that for compatibility with Java 1.0, this example defines the deprecated `preferredSize()` and `minimumSize()` methods instead.)

`MultiLineLabel` extends `Canvas`, which is a blank component intended primarily for subclassing. When a component is a subclass of `Canvas`, it is typically given its own native window in the underlying window system. In Java 1.1, however, it is possible to define "lightweight" custom components by extending `Component` instead of `Canvas`. A lightweight component does not have its own native window in the underlying window system.

What makes this component a bean is that all of its properties have `get` and `set` accessor methods. Because `MultiLineLabel` does not respond to user input in any way, it does not define any events, so there are no event listener registration methods required. Although it is not a formal requirement for a bean, most beanboxes attempt to instantiate a bean by invoking its no-argument constructor. Thus, a bean should define a constructor that expects no arguments.

Example 10.1: A Custom AWT Component and JavaBean

```
package oreilly.beans.yesno;
import java.awt.*;
import java.util.*;
/**
 * A custom component that displays multiple lines of text with specified
 * margins and alignment.  In Java 1.1, we could extend Component instead
 * of Canvas, making this a more efficient "Lightweight component."
 */
public class MultiLineLabel extends Canvas {
  // User-specified attributes
  protected String label;             // The label, not broken into lines
  protected int margin_width;         // Left and right margins
  protected int margin_height;        // Top and bottom margins
  protected int alignment;            // The alignment of the text
  public static final int LEFT = 0, CENTER = 1, RIGHT = 2; // alignment values
  // Computed state values
  protected int num_lines;            // The number of lines
  protected String[] lines;           // The label, broken into lines
  protected int[] line_widths;        // How wide each line is
  protected int max_width;            // The width of the widest line
  protected int line_height;          // Total height of the font
  protected int line_ascent;          // Font height above baseline
  protected boolean measured = false; // Have the lines been measured?
  // Here are five versions of the constructor.
  public MultiLineLabel(String label, int margin_width,
                        int margin_height, int alignment) {
    this.label = label;                 // Remember all the properties.
    this.margin_width = margin_width;
    this.margin_height = margin_height;
    this.alignment = alignment;
    newLabel();                         // Break the label up into lines.
  }
  public MultiLineLabel(String label, int margin_width, int margin_height) {
    this(label, margin_width, margin_height, LEFT);
  }
  public MultiLineLabel(String label, int alignment) {
    this(label, 10, 10, alignment);
  }
  public MultiLineLabel(String label) { this(label, 10, 10, LEFT); }
  public MultiLineLabel() { this(""); }
  // Methods to set and query the various attributes of the component.
  // Note that some query methods are inherited from the superclass.
  public void setLabel(String label) {
    this.label = label;
    newLabel();               // Break the label into lines.
    measured = false;         // Note that we need to measure lines.
    repaint();                // Request a redraw.
  }
  public void setFont(Font f) {
    super.setFont(f);         // Tell our superclass about the new font.
    measured = false;         // Note that we need to remeasure lines.
    repaint();                // Request a redraw.
  }
  public void setForeground(Color c) {
    super.setForeground(c);   // Tell our superclass about the new color.
    repaint();                // Request a redraw (size is unchanged).
  }
  public void setAlignment(int a) { alignment = a; repaint(); }
  public void setMarginWidth(int mw) { margin_width = mw; repaint(); }
  public void setMarginHeight(int mh) { margin_height = mh; repaint(); }
  public String getLabel() { return label; }
  public int getAlignment() { return alignment; }
  public int getMarginWidth() { return margin_width; }
  public int getMarginHeight() { return margin_height; }
  /**
   * This method is called by a layout manager when it wants to
   * know how big we'd like to be.  In Java 1.1, getPreferredSize() is
   * the preferred version of this method.  We use this deprecated version
   * so that this component can interoperate with 1.0 components.
   */
  public Dimension preferredSize() {
    if (!measured) measure();
    return new Dimension(max_width + 2*margin_width,
                         num_lines * line_height + 2*margin_height);
  }
  /**
   * This method is called when the layout manager wants to know
   * the bare minimum amount of space we need to get by.
   * For Java 1.1, we'd use getMinimumSize().
   */
  public Dimension minimumSize() { return preferredSize(); }
  /**
   * This method draws the label (same method that applets use).
   * Note that it handles the margins and the alignment, but that
   * it doesn't have to worry about the color or font--the superclass
   * takes care of setting those in the Graphics object we're passed.
   */
  public void paint(Graphics g) {
    int x, y;
    Dimension size = this.size();  // Use getSize() in Java 1.1.
    if (!measured) measure();
    y = line_ascent + (size.height - num_lines * line_height)/2;
    for(int i = 0; i < num_lines; i++, y += line_height) {
      switch(alignment) {
      default:
      case LEFT:    x = margin_width; break;
      case CENTER:  x = (size.width - line_widths[i])/2; break;
      case RIGHT:   x = size.width - margin_width - line_widths[i]; break;
      }
      g.drawString(lines[i], x, y);
    }
  }
  /** This internal method breaks a specified label up into an array of lines.
```

```
   *  It uses the StringTokenizer utility class. */
  protected synchronized void newLabel() {
    StringTokenizer t = new StringTokenizer(label, "\n");
    num_lines = t.countTokens();
    lines = new String[num_lines];
    line_widths = new int[num_lines];
    for(int i = 0; i < num_lines; i++) lines[i] = t.nextToken();
  }
  /** This internal method figures out how big the font is, and how wide each
   *  line of the label is, and how wide the widest line is. */
  protected synchronized void measure() {
    FontMetrics fm = this.getToolkit().getFontMetrics(this.getFont());
    line_height = fm.getHeight();
    line_ascent = fm.getAscent();
    max_width = 0;
    for(int i = 0; i < num_lines; i++) {
      line_widths[i] = fm.stringWidth(lines[i]);
      if (line_widths[i] > max_width) max_width = line_widths[i];
    }
    measured = true;
  }
}
```

Packaging a Bean

To prepare a bean for use in a beanbox, you must package it up in a JAR file, along with any other classes or resource files it requires. (JAR files are "Java archives"; they were introduced in Chapter 6, *Applets*.) Because a single bean can have many auxiliary files, and because a JAR file can contain multiple beans, the manifest of the JAR file must define which of the JAR file entries are beans. You create a JAR file with `c` option to the *jar* command. When you use the `m` option in conjunction with `c`, it tells *jar* to read a partial manifest file that you specify. *jar* uses the information in your partially-specified manifest file when creating the complete manifest for the JAR file. To identify a class file as a bean, you simply add the following line to the file's manifest entry:

```
Java-Bean: True
```

So, to package up the `MultiLineLabel` class in a JAR file, you must first create a manifest "stub" file. Create a file, perhaps named *manifest.stub*, with the following contents:

```
Name: oreilly/beans/MultiLineLabel.class
Java-Bean: True
```

Note that the forward slashes in the manifest file should not be changed to backward slashes on Windows systems. The format of the JAR manifest file requires forward slashes to separate directories regardless of the platform. Having created this partial manifest file, we can now go ahead and create the JAR file:

```
% jar cfm MultiLineLabel.jar manifest.stub oreilly/beans/MultiLineLabel.class
```

(On a Windows system, you do need to replace the forward slashes with backslashes in this command line.) If this bean required any auxiliary files, we would specify them on the end of the *jar* command line, along with the class file for the bean.

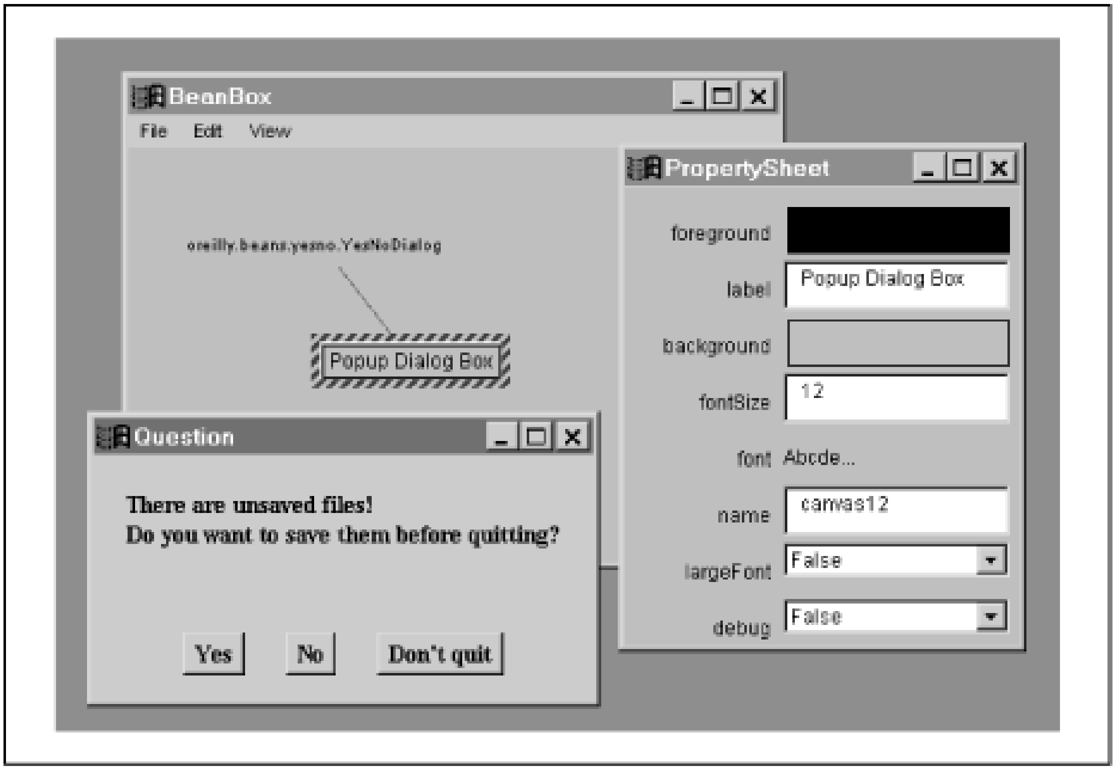
Installing a Bean

The procedure for installing a bean depends entirely upon the beanbox tool that you are using. For the *beanbox* tool shipped with the BDK, all you need to do is to copy the bean's JAR file into the *jars/* directory within the BDK directory. Once you have done this, the bean will appear on the palette of beans every time you start up the application. Alternatively, you can also load a bean's JAR file at runtime by selecting the **Load JAR** option from the **File** menu of *beanbox*.

10.3 A More Complex Bean

Example 10.2 shows another bean, `YesNoDialog`. This bean creates a dialog box that displays a simple message to the user and asks the user to respond by clicking the **Yes**, **No**, or **Cancel** button. Figure 10.1 shows the bean being manipulated in Sun's *beanbox* tool and also shows a dialog displayed by the bean.

Figure 10.1: The YesNoDialog bean in the *beanbox*



foreground
label Popup Dialog Box
background
fontSize 12
font Abcde...
name canvas12
largeFont False
debug False

There are unsaved files!
Do you want to save them before quitting?

The `YesNoDialog` bean uses a custom `AnswerEvent` type to notify `AnswerListener` objects when the user has dismissed the dialog by clicking on one of its three buttons. This new event class and listener interface are defined in the next section.

Note that `YesNoDialog` is a subclass of `Object`, not `Dialog`. This is a result of the requirement for a bean to have a no-argument constructor. Because all dialog boxes must be associated with a `Frame`, `Dialog` does not have a no-argument constructor, and this means that subclasses of `Dialog` cannot have meaningful no-argument constructors, either. As a result, `YesNoDialog` defers creation of its window and associated GUI components until it is actually popped up with the `display()` method. Another *beanbox* shortcoming is that it only recognizes methods with no arguments. [1] For this reason, the `display()` method has no arguments, and so no `Frame` can be specified through that method either. Since a parent `Frame` cannot be specified, `YesNoDialog` cannot create a `Dialog` object, and instead simulates a dialog box with a `Frame` window. An alternative would have been to define a bean property through which the required `Frame` could be specified.

> [1] The *beanbox* tool shipped with the February 1997 version of the BDK has a number of shortcomings. In part, this is due to the fact that the BDK is a new technology and not as stable or robust as the JDK. It is also because *beanbox* is intended as a test environment, not an actual programmer's tool.

Since `YesNoDialog` is not a subclass of `Component`, it has to define its own properties and accessor methods for fonts and colors; normally these would simply be inherited from `Component`. Since this bean is not a `Component` subclass, it is an "invisible" bean that does not have a graphical representation of its own. (It pops up a window when the `display()` method is called, but that is not the same as having a graphical representation that appears within another window.) Different tools treat invisible beans differently. *beanbox* simply displays the classname of invisible beans.

Notice that `YesNoDialog` does not use any classes from the `java.beans` package. One of the surprising things about beans is that they typically do not have to use any classes from this package. As we'll see in later sections, it is the auxiliary classes that are shipped with a bean that make heavy use of that package.

Example 10.2: The YesNoDialog Bean

```
package oreilly.beans.yesno;      // Put this bean in its own private package.
import java.awt.*;
import java.awt.event.*;
import java.util.*;
public class YesNoDialog {
  // Properties of the bean.
  protected String message, title;
  protected String yesLabel, noLabel, cancelLabel;
  protected int alignment;
  protected Font font = new Font("Serif", Font.PLAIN, 12);
  protected Color background = SystemColor.control;
  protected Color foreground = SystemColor.controlText;
  // Constants for the alignment property.
  public static final int LEFT = MultiLineLabel.LEFT;
  public static final int RIGHT = MultiLineLabel.RIGHT;
  public static final int CENTER = MultiLineLabel.CENTER;
  // Methods to query all of the bean properties.
  public String getMessage() { return message; }
  public String getTitle() { return title; }
  public String getYesLabel() { return yesLabel; }
  public String getNoLabel() { return noLabel; }
  public String getCancelLabel() { return cancelLabel; }
  public int getAlignment() { return alignment; }
  public Font getFont() { return font; }
  public Color getBackground() { return background; }
  public Color getForeground() { return foreground; }
  // Methods to set all of the bean properties.
  public void setMessage(String m) { message = m; }
  public void setTitle(String t) { title=t; }
  public void setYesLabel(String l) { yesLabel = l; }
  public void setNoLabel(String l) { noLabel = l; }
  public void setCancelLabel(String l) { cancelLabel = l; }
  public void setAlignment(int a) { alignment = a; }
  public void setFont(Font f) { font = f; }
  public void setBackground(Color bg) { background = bg; }
  public void setForeground(Color fg) { foreground = fg; }
  /** This field holds a list of registered ActionListeners.
   *  Vector is internally synchronized to prevent race conditions. */
  protected Vector listeners = new Vector();
  /** Register an action listener to be notified when a button is pressed. */
  public void addAnswerListener(AnswerListener l) {
    listeners.addElement(l);
  }
  /** Remove an Answer listener from our list of interested listeners. */
  public void removeAnswerListener(AnswerListener l) {
    listeners.removeElement(l);
  }
  /** Send an event to all registered listeners. */
  public void fireEvent(AnswerEvent e) {
    // Make a copy of the list and fire the events using that copy.
    // This means that listeners can be added or removed from the original
    // list in response to this event.  We ought to be able to just use an
    // enumeration for the vector, but that doesn't copy the list internally.
    Vector list = (Vector) listeners.clone();
    for(int i = 0; i < list.size(); i++) {
      AnswerListener listener = (AnswerListener)list.elementAt(i);
      switch(e.getID()) {
      case AnswerEvent.YES: listener.yes(e); break;
      case AnswerEvent.NO:  listener.no(e); break;
      case AnswerEvent.CANCEL: listener.cancel(e); break;
      }
    }
  }
  /** The no-argument bean constructor, with default property values */
  public YesNoDialog() {
    this("Question", "Your\nMessage\nHere", "Yes", "No", "Cancel", LEFT);
  }
  /** A constructor for programmers using this class "by hand". */
  public YesNoDialog(String title, String message,
                     String yesLabel, String noLabel, String cancelLabel,
                     int alignment) {
    this.title = title;
    this.message = message;
    this.yesLabel = yesLabel;
    this.noLabel = noLabel;
    this.cancelLabel = cancelLabel;
    this.alignment = alignment;
  }
  /** This method makes the bean display the dialog box. */
  public void display() {
    // Create a frame with the specified title.  It would be nice to
    // use a Dialog, but that needs to be passed a Frame argument, and
    // the BDK beanbox tool only seems to work with no-argument methods.
    final Frame frame = new Frame(title);
    // Specify a LayoutManager for it.
    frame.setLayout(new BorderLayout(15, 15));
    // Specify font and colors, if any are specified.
    if (font != null) frame.setFont(font);
    if (background != null) frame.setBackground(background);
    if (foreground != null) frame.setForeground(foreground);
    // Put the message label in the middle of the window.
    frame.add("Center", new MultiLineLabel(message, 20, 20, alignment));
    // Create an action listener for use by the buttons of the dialog.
    // When a button is pressed, this listener first closes the dialog box.
    // Then, it creates an AnswerEvent object that corresponds to the
    // button that was pressed, and sends that new event to all registered
    // listeners, using the fireEvent() method defined above.
    ActionListener listener = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        frame.dispose();     // Pop down window.
        if (listeners != null) {      // Notify any registered listeners.
          String cmd = e.getActionCommand();
          int type;
          if (cmd.equals("yes")) type = AnswerEvent.YES;
          else if (cmd.equals("no")) type = AnswerEvent.NO;
          else type = AnswerEvent.CANCEL;
          fireEvent(new AnswerEvent(YesNoDialog.this, type));
        }
      }
    };
    // Create a panel for the dialog buttons and put it at the bottom
    // of the dialog.  Specify a FlowLayout layout manager for it.
    Panel buttonbox = new Panel();
    buttonbox.setLayout(new FlowLayout(FlowLayout.CENTER, 25, 15));
    frame.add("South", buttonbox);
    // Create each specified button, specifying the action listener
    // and action command for each, and adding them to the buttonbox.
    if ((yesLabel != null) && (yesLabel.length() > 0)) {
      Button yes = new Button(yesLabel);        // Create button.
      yes.setActionCommand("yes");              // Set action command.
      yes.addActionListener(listener);          // Set listener.
      buttonbox.add(yes);                       // Add button to the panel.
    }
```

```
    if ((noLabel != null) && (noLabel.length() > 0)) {
      Button no = new Button(noLabel);
      no.setActionCommand("no");
      no.addActionListener(listener);
      buttonbox.add(no);
    }
    if ((cancelLabel != null) && (cancelLabel.length() > 0)) {
      Button cancel = new Button(cancelLabel);
      cancel.setActionCommand("cancel");
      cancel.addActionListener(listener);
      buttonbox.add(cancel);
    }
    // Finally, set the dialog to its preferred size and display it.
    frame.pack();
    frame.show();
  }
  /**
   * A main method that demonstrates how to use this class, and allows testing.
   */
  public static void main(String[] args) {
    // Create an instance of InfoDialog, with title and message specified:
    YesNoDialog d =
      new YesNoDialog("YesNoDialog Test",
                      "There are unsaved files.\n" +
                      "Do you want to save them before quitting?",
                      "Yes, save and quit",
                      "No, quit without saving",
                      "Cancel; don't quit",
                      YesNoDialog.CENTER);
    // Register an action listener for the dialog.  This one just prints
    // the results out to the console.
    d.addAnswerListener(new AnswerListener() {
      public void yes(AnswerEvent e) { System.out.println("Yes"); }
      public void no(AnswerEvent e) { System.out.println("No"); }
      public void cancel(AnswerEvent e) { System.out.println("Cancel"); }
    });
    // Now pop the dialog up.  It will pop itself down automatically.
    d.display();
  }
}
```

10.4 Custom Events

Beans can use the standard AWT event types defined in the `java.awt.event` package, but they do not have to. Our `YesNoDialog` class defines its own event type, `AnswerEvent`. Defining a new event class is really quite simple; `AnswerEvent` is shown in Example 10.3.

Example 10.3: The AnswerEvent Class

```
package oreilly.beans.yesno;
public class AnswerEvent extends java.util.EventObject {
  protected int id;
  public static final int YES = 0, NO = 1, CANCEL = 2;
  public AnswerEvent(Object source, int id) {
    super(source);
    this.id = id;
  }
  public int getID() { return id; }
}
```

Along with the `AnswerEvent` class, `YesNoDialog` also defines a new type of event listener interface, `ActionListener`, that defines the methods that must be implemented by any object that wants to receive notification from a `YesNoDialog`. The definition of `AnswerListener` is shown in Example 10.4.

Example 10.4: The AnswerListener Interface

```
package oreilly.beans.yesno;
public interface AnswerListener extends java.util.EventListener {
  public void yes(AnswerEvent e);
  public void no(AnswerEvent e);
  public void cancel(AnswerEvent e);
}
```

PREVIOUS
A More Complex Bean

HOME
BOOK INDEX

NEXT
Specifying Bean Information

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

10.5 Specifying Bean Information

The `YesNoDialog` class itself and the `AnswerEvent` and `AnswerListener` classes it relies on are all a required part of our bean. When an application that uses the bean is shipped, it has to include all three of these class files. There are other kinds of classes, however, that are often bundled with a bean that are not intended for use by the application developer. These classes are used by the beanbox tool that manipulates the bean. The bean class itself does not refer to any of these auxiliary beanbox classes so that it is not dependent on them and they do not have to be shipped with the bean in finished products.

The first of these optional, auxiliary classes is a `BeanInfo` class. As explained earlier, a beanbox discovers the properties, events, and methods exported by a bean through "introspection" based on the Java Reflection API. A bean developer who wants to provide additional information about a bean, or who wants to refine the (somewhat rough) information available through introspection, should define a class that implements the `BeanInfo` interface to provide that information. A `BeanInfo` class typically subclasses `SimpleBeanInfo`, which provides a no-op implementation of the `BeanInfo` interface. When you only want to override one or two methods, it is easier to subclass `SimpleBeanInfo` than to implement `BeanInfo` directly. Beanbox tools rely on a naming convention in order to find the `BeanInfo` class for a given bean: a `BeanInfo` class should have the same name as the bean, with the string "BeanInfo" appended. Example 10.5 shows an implementation of the `YesNoDialogBeanInfo` class.

This `BeanInfo` class specifies a number of pieces of information for our bean:

- An icon that can be used to represent the bean.
- A `BeanDescriptor` object, which includes a reference to a `Customizer` class for the bean. We'll see an implementation of this class later in the chapter.
- A list of the supported properties of the bean, along with a short description of each one. Some beanbox tools (but not Sun's *beanbox*) display these strings to the user in some useful way.
- A method that returns the most commonly customized property of the bean--this is called the "default" property.
- References to `PropertyEditor` classes for two of the properties. We'll see implementations of these property editor classes later in the chapter.
- A list of the methods supported by the bean, again with a short description of each one.

Besides specifying this information, a `BeanInfo` class can also provide information about the events it generates and specify a default event. The various `FeatureDescriptor` objects used to provide information about such things as properties and methods can also include other information not provided by `YesNoDialogBeanInfo`, such as a localized display name that is distinct from the programmatic name.

Example 10.5: The YesNoDialogBeanInfo Class

```
package oreilly.beans.yesno;
import java.beans.*;
import java.lang.reflect.*;
import java.awt.*;
/** The BeanInfo class for the YesNoDialog bean. */
public class YesNoDialogBeanInfo extends SimpleBeanInfo {
  /** Return an icon for the bean.  We should really check the kind argument
   *  to see what size icon the beanbox wants, but since we only have one
   *  icon to offer, we just return it and let the beanbox deal with it. */
  public Image getIcon(int kind) {
    return loadImage("YesNoDialogIcon.gif");
  }
  /** Return a descriptor for the bean itself.  It specifies a customizer
   *  for the bean class.  We could also add a description string here. */
  public BeanDescriptor getBeanDescriptor() {
    return new BeanDescriptor(YesNoDialog.class, YesNoDialogCustomizer.class);
  }
  /** This is a convenience routine for creating PropertyDescriptor objects. */
  public static PropertyDescriptor property(String name, String description)
       throws IntrospectionException
  {
    PropertyDescriptor p = new PropertyDescriptor(name, YesNoDialog.class);
    p.setShortDescription(description);
    return p;
  }
  /** This method returns an array of PropertyDescriptor objects that specify
   *  additional information about the properties supported by the bean.
   *  By explicitly specifying property descriptors, we are able to provide
   *  simple help strings for each property; these would not be available to
   *  the beanbox through simple introspection.  We are also able to register
   *  special property editors for two of the properties.
   */
  public PropertyDescriptor[] getPropertyDescriptors() {
    try {
      PropertyDescriptor[] props = {
        property("title", "The string that appears in the dialog title bar"),
        property("message", "The string that appears in the dialog body"),
        property("yesLabel", "The label for the 'Yes' button, if any"),
        property("noLabel", "The label for the 'No' button, if any"),
        property("cancelLabel", "The label for the 'Cancel' button, if any"),
        property("alignment", "The alignment of the message text"),
        property("font", "The font to use for message and buttons"),
        property("background", "The background color for the dialog"),
        property("foreground", "The text color for message and buttons")
      };
      props[1].setPropertyEditorClass(YesNoDialogMessageEditor.class);
      props[5].setPropertyEditorClass(YesNoDialogAlignmentEditor.class);
      return props;
    }
    catch (IntrospectionException e) {return super.getPropertyDescriptors(); }
  }
  /** The message property is most often customized; make it the default. */
  public int getDefaultPropertyIndex() { return 1; }
  /** This is a convenience method for creating MethodDescriptors.  Note that
   *  it assumes we are talking about methods with no arguments. */
  public static MethodDescriptor method(String name, String description)
       throws NoSuchMethodException, SecurityException {
    Method m = YesNoDialog.class.getMethod(name, new Class[] {});
    MethodDescriptor md = new MethodDescriptor(m);
    md.setShortDescription(description);
    return md;
  }
  /** This method returns an array of method descriptors for the supported
   *  methods of a bean. This allows us to provide useful description strings,
   *  but it also allows us to filter out non-useful methods like wait()
   *  and notify() that the bean inherits and which might otherwise be
   *  displayed by the beanbox.
   */
  public MethodDescriptor[] getMethodDescriptors() {
    try {
      MethodDescriptor[] methods = {
        method("display", "Pop up the dialog; make it visible")
      };
      return methods;
    }
    catch (Exception e) {
      return super.getMethodDescriptors();
    }
  }
}
```

10.6 Defining a Simple Property Editor

A bean can also provide an auxiliary `PropertyEditor` for use by a beanbox tool. `PropertyEditor` is a flexible interface that allows a bean to tell a beanbox how to display and edit the values of certain types of properties.

A beanbox tool always provides simple property editors for common property types, such as strings, numbers, fonts, and colors. If your bean has properties of a non-standard type, you should register a property editor for that type. The easiest way to "register" a property editor is through a simple naming convention. If your type is defined by the class *X*, the editor for it should be defined in the class `XEditor`. Alternately, you can register a property editor by calling the `PropertyEditorManager.registerEditor()` method, probably from the constructor of your `BeanInfo` class. If you call this method from the bean itself, the bean then depends on the property editor class, so the editor has to be bundled with the bean in applications.

In our `YesNoDialog` example, we don't define any new data types, but we still have two individual properties that need custom editors. In this case, we register the property editors for individual properties by specifying them in the `PropertyDescriptor` objects returned by the `getPropertyDescriptors()` method of our `BeanInfo` class.

The `PropertyEditor` interface can seem confusing at first. Its methods allow you to define three techniques for displaying the value of a property and two techniques for allowing the user to edit the value of a property. The value of a property can be displayed:

- As a string. If you define the `getAsText()` method, a beanbox can convert a property to a string and display that string to the user.
- As an enumerated value. If a property can only take on a value from a fixed set of values, you can define the `getTags()` method to allow a beanbox to display a dropdown menu of allowed values for the property.
- In a graphical display. If you define `paintValue()`, a beanbox can ask the property editor to display the value using some natural graphical format, such as a color swatch for colors. You also need to define `isPaintable()` to specify that a graphical format is supported.

The two editing techniques are:

- *String editing*. If you define the `setAsText()` method, a beanbox knows it can simply have the user type a value into a text field and pass that value to `setAsText()`. If your property editor defines `getTags()`, it should also define `setAsText()` so that a beanbox can set the property value using the individual tag values.

- *Custom editing*. If your property editor defines `getCustomEditor()`, a beanbox can call it to obtain some kind of AWT component that can be displayed in a dialog box and serve as a custom editor for the property. You also need to define `supportsCustomEditor()` to specify that custom editing is supported.

The `setValue()` method of a `PropertyEditor` is called to specify the current value of the property. It is this value that should be converted to a string or graphical representation by `getAsText()` or `paintValue()`.

A property editor is required to maintain a list of event listeners that are interested in changes to the value of the property. The `addPropertyChangeListener()` and `removePropertyChangeListener()` methods are standard event listener registration and removal methods. When a property editor changes the value of a property, either through `setAsText()` or through a custom editor, it must send a `PropertyChangeEvent` to all registered listeners.

`PropertyEditor` defines the `getJavaInitializationString()` for use by beanbox tools that generate Java code. This method should return a fragment of Java code that can be used to initialize a variable to the current property value.

Finally, a class that implements the `PropertyEditor` interface must have a no-argument constructor, so that it can be dynamically loaded and instantiated by a beanbox.

Most property editors can be much simpler than this detailed description would suggest. In many cases, you can subclass `PropertyEditorSupport` instead of implementing the `PropertyEditor` interface directly. This useful class provides no-op implementations of most `PropertyEditor` methods. It also implements the methods for adding and removing event listeners.

A property that has an enumerated value requires a simple property editor. The `alignment` property of the `YesNoDialog` bean is an example of this common type of property. The property is defined as an `int`, but it has only three legal values, defined by the constants `LEFT`, `CENTER`, and `RIGHT`. By default, a beanbox only knows that the property is an `int`, so it displays the property as a number and allows the user to enter any integer as a property value. Instead, we would like the beanbox to display one of the strings "left," "center," or "right" as the value, and allow the user to choose from these values when setting the property. This can be done with the `getTags()` and `setAsText()` methods of a property editor, as shown in Example 10.6.

This example creates the `YesNoDialogAlignmentEditor` class, which is registered as a `PropertyEditor` for the `alignment` property by the `YesNoDialogBeanInfo` class shown in Example 10.5 . The property editor subclasses `PropertyEditorSupport`, so it is relatively short. Notice that it passes `Integer` objects in the calls to `setValue()` that are made from the `setAsText()` method. You need to use wrapper objects for any primitive-type properties. The use of the `Integer` class is also apparent in the definition of `getJavaInitializationString()`. The `setValue()` method of `ProperyEditorSupport` handles notifying registered `PropertyChangeListener` objects about changes in the value of the property, so this simple property editor does not need to be aware of the existence of such listeners.

Example 10.6: The YesNoDialogAlignmentEditor Class

```
package oreilly.beans.yesno;
import java.beans.*;
import java.awt.*;
public class YesNoDialogAlignmentEditor extends PropertyEditorSupport {
  // These two methods allow the property to be edited in a dropdown list.
  // Return the list of value names for the enumerated type.
  public String[] getTags() {
    return new String[] { "left", "center", "right" };
  }
  // Convert each of those value names into the actual value.
  public void setAsText(String s) {
    if (s.equals("left")) setValue(new Integer(YesNoDialog.LEFT));
    else if (s.equals("center")) setValue(new Integer(YesNoDialog.CENTER));
    else if (s.equals("right")) setValue(new Integer(YesNoDialog.RIGHT));
    else throw new IllegalArgumentException(s);
  }
  // This is an important method for code generation.
  public String getJavaInitializationString() {
    switch(((Number)getValue()).intValue()) {
    default:
    case YesNoDialog.LEFT:   return "oreilly.beans.yesno.YesNoDialog.LEFT";
    case YesNoDialog.CENTER: return "oreilly.beans.yesno.YesNoDialog.CENTER";
    case YesNoDialog.RIGHT:  return "oreilly.beans.yesno.YesNoDialog.RIGHT";
    }
  }
}
```

10.7 Defining a Complex Property Editor

There is another `YesNoDialog` property value that requires a property editor. The `message` property of `YesNoDialog` can specify a multi-line message to be displayed in the dialog. This property requires a property editor because the *beanbox* program does not distinguish between single-line and multi-line string types and the `TextField` objects it uses for text input do not allow the user to enter multiple lines of text. For this reason, we define the `YesNoDialogMessageEditor` class and register it with the `PropertyDescriptor` for the message property, as shown in Example 10.5.

Example 10.7 shows the definition of this property editor. This is a more complex editor that supports the creation of a custom editor component and graphical display of the value. Note that this example implements `PropertyEditor` directly. `getCustomEditor()` returns an editor component for multi-line strings. Figure 10.2 shows this custom editor placed within a dialog box created by the *beanbox* program. Note that the **Done** button in this figure is part of the *beanbox* dialog, not part of the property editor itself.

Figure 10.2: A custom property editor dialog

[Graphic: Figure 10-2]

The `paintValue()` method is responsible for displaying the value of the `message` property. This multi-line value does not typically fit in the small rectangle of screen space allowed for the property, so `paintValue()` displays instructions for popping up the custom editor, which allows the user to inspect and edit the property value. (This example relies on the click-to-edit behavior of the *beanbox* program; this `paintValue()` method may not make sense when the bean is used in other beanbox tools.)

Example 10.7: The YesNoDialogMessageEditor Class

```
package oreilly.beans.yesno;
import java.beans.*;
import java.awt.*;
import java.awt.event.*;
/**
 * This class is a custom editor for the message property of the
 * YesNoDialog bean.  It is necessary because the default editor for
 * properties of type String does not allow multi-line strings to be entered.
 */
public class YesNoDialogMessageEditor implements PropertyEditor {
  protected String value;  // The value we will be editing.
  public void setValue(Object o) {  value = (String) o; }
  public Object getValue() { return value; }
  public void setAsText(String s) { value = s; }
  public String getAsText() { return value; }
  public String[] getTags() { return null; }  // not enumerated; no tags
  // Say that we allow custom editing.
  public boolean supportsCustomEditor() { return true; }
  // Return the custom editor.  This just creates and returns a TextArea
  // to edit the multi-line text.  But it also registers a listener on the
  // text area to update the value as the user types and to fire the
  // property change events that property editors are required to fire.
  public Component getCustomEditor() {
    final TextArea t = new TextArea(value);
    t.setSize(300, 150); // TextArea doesn't have a preferred size, so set one.
    t.addTextListener(new TextListener() {
      public void textValueChanged(TextEvent e) {
        value = t.getText();
        listeners.firePropertyChange(null, null, null);
      }
    });
    return t;
  }
  // Visual display of the value, for use with the custom editor.
  // Just print some instructions and hope they fit in the box.
  // This could be more sophisticated.
  public boolean isPaintable() { return true; }
  public void paintValue(Graphics g, Rectangle r) {
    g.setClip(r);
    g.drawString("Click to edit...", r.x+5, r.y+15);
  }
  // Important method for code generators.  Note that it
  // ought to add any necessary escape sequences.
  public String getJavaInitializationString() { return "\"" + value + "\""; }
  // This code uses the PropertyChangeSupport class to maintain a list of
  // listeners interested in the edits we make to the value.
  protected PropertyChangeSupport listeners = new PropertyChangeSupport(this);
  public void addPropertyChangeListener(PropertyChangeListener l) {
    listeners.addPropertyChangeListener(l);
  }
  public void removePropertyChangeListener(PropertyChangeListener l) {
    listeners.removePropertyChangeListener(l);
  }
}
```

PREVIOUS HOME NEXT
Defining a Simple Property Editor BOOK INDEX Defining a Bean Customizer

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

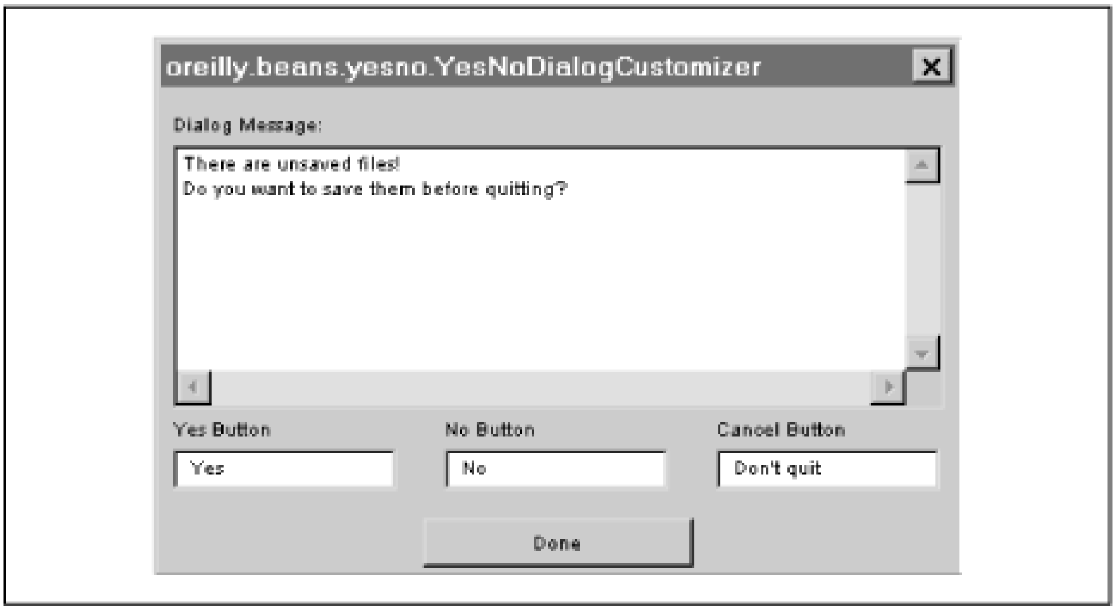
10.8 Defining a Bean Customizer

A bean may want to provide some way for the user of a beanbox program to customize its properties other than by setting them one at a time. A bean can do this by creating a `Customizer` class for itself, and registering the customizer class with the `BeanDescriptor` object returned by its `BeanInfo` class, as we saw in Example 10.5.

A customizer must be some kind of AWT component that is suitable for display in a dialog box created by the beanbox. Therefore, a customizer class is typically a subclass of `Panel`. In addition, a customizer must implement the `Customizer` interface. This interface consists of methods for adding and removing property change event listeners and a `setObject()` method that the beanbox calls to tell the customizer what bean object it is customizing. Whenever the user makes a change to the bean through the customizer, the customizer should send a `PropertyChangeEvent` to any interested listeners. Finally, like a property editor, a customizer must have a no-argument constructor, so it can easily be instantiated by a beanbox.

Example 10.8 shows a customizer for our `YesNoDialog` bean. This customizer displays a panel that has the same layout as a `YesNoDialog`, but it substitutes a `TextArea` object for the message display and three `TextField` objects for the three buttons that the dialog can display. These text entry areas allow the user to enter values for the `message`, `yesLabel`, `noLabel`, and `cancelLabel` properties. Figure 10.3 shows this customizer panel displayed within a dialog box created by the *beanbox* program. Again, note that the **Done** button is part of the *beanbox* dialog, not part of the customizer itself.

Figure 10.3: The customizer dialog for the YesNoDialog bean



Dialog Message:

There are unsaved files!
Do you want to save them before quitting?

Yes Button: Yes

No Button: No

Cancel Button: Don't quit

Example 10.8: The YesNoDialogCustomizer Class

```
package oreilly.beans.yesno;
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
/**
 * This class is a customizer for the YesNoDialog bean.  It displays a
 * TextArea and three TextFields where the user can enter the dialog message
 * and the labels for each of the three buttons.  It does not allow the
 * dialog title or other resources to be set.
 */
public class YesNoDialogCustomizer extends Panel
                                   implements Customizer, TextListener
{
  protected YesNoDialog bean;                 // The bean being customized.
  protected TextComponent message, fields[];  // Components used by customizer
  // Default constructor: YesNoDialogCustomizer() { super(); }
  // The bean box calls this method to tell us what object to customize.
  // This method will always be called before the customizer is displayed,
  // so it is safe to create the customizer GUI here.
  public void setObject(Object o) {
    bean = (YesNoDialog)o;   // Save the object we're customizing.
    // Put a label at the top of the panel.
    this.setLayout(new BorderLayout());
    this.add(new Label("Enter the message to appear in the dialog:"), "North");
    // And a big text area below it for entering the dialog message.
    message = new TextArea(bean.getMessage());
    message.addTextListener(this);
    // TextAreas don't know how big they want to be.  You must tell them.
    message.setSize(400, 200);
    this.add(message, "Center");
    // Then add a row of textfields for entering the button labels.
    Panel buttonbox = new Panel();                       // The row container.
    buttonbox.setLayout(new GridLayout(1, 0, 25, 10)); // Equally spaced items.
    this.add(buttonbox, "South");                        // Put row on bottom.
    // Now go create three TextFields to put in this row.  But actually
    // position a Label above each, so create a container for each
    // TextField+Label combination.
    fields = new TextComponent[3];             // Array of TextFields.
    String[] labels = new String[] {           // Labels for each.
      "Yes Button Label", "No Button Label", "Cancel Button Label"};
    String[] values = new String[] {           // Initial values of each.
      bean.getYesLabel(), bean.getNoLabel(), bean.getCancelLabel()};
    for(int i = 0; i < 3; i++) {
      Panel p = new Panel();                   // Create a container.
      p.setLayout(new BorderLayout());         // Give it a BorderLayout.
      p.add(new Label(labels[i]), "North");    // Put a label on the top.
      fields[i] = new TextField(values[i]);    // Create the text field.
      p.add(fields[i], "Center");              // Put it below the label.
      fields[i].addTextListener(this);         // Set the event listener.
      buttonbox.add(p);                        // Add container to row.
    }
  }
  // Add some space around the outside of the panel.
  public Insets getInsets() { return new Insets(10, 10, 10, 10); }
  // This is the method defined by the TextListener interface.  Whenever the
  // user types a character in the TextArea or TextFields, this will get
  // called.  It updates the appropriate property of the bean and fires a
  // property changed event, as all customizers are required to do.
  // Note that we are not required to fire an event for every keystroke.
  // Instead we could include an "Apply" button that would make all the
  // changes at once, with a single property changed event.
  public void textValueChanged(TextEvent e) {
    TextComponent t = (TextComponent)e.getSource();
    String s = t.getText();
    if (t == message) bean.setMessage(s);
    else if (t == fields[0]) bean.setYesLabel(s);
    else if (t == fields[1]) bean.setNoLabel(s);
    else if (t == fields[2]) bean.setCancelLabel(s);
    listeners.firePropertyChange(null, null, null);
  }
  // This code uses the PropertyChangeSupport class to maintain a list of
  // listeners interested in the edits we make to the bean.
  protected PropertyChangeSupport listeners = new PropertyChangeSupport(this);
  public void addPropertyChangeListener(PropertyChangeListener l) {
    listeners.addPropertyChangeListener(l);
  }
  public void removePropertyChangeListener(PropertyChangeListener l) {
    listeners.removePropertyChangeListener(l);
  }
```

```
}
```

3.6 Subclasses and Inheritance

The `Circle` class we've defined is good for abstract mathematical manipulation. For some applications this is exactly what we need. For other applications, we want to be able to manipulate circles *and* draw them on the screen. This means we need a new class, `GraphicCircle`, that has all the functionality of `Circle`, but also has the ability to be drawn.

We want to implement `GraphicCircle` so that it can make use of the code we've already written for `Circle`. One way to do that is the following:

```
public class GraphicCircle {
    // Here is the mathematical circle.
    public Circle c;
    // Here are the old methods.
    public double area() { return c.area(); }
    public double circumference() { return c.circumference(); }
    // The new graphic variables and methods go here.
    public Color outline, fill;
    public void draw(DrawWindow dw) { /* code omitted */ }
}
```

This approach would work, but it is not particularly elegant. The problem is that we have to write stubs for methods like `area()` and `circumference()` that have nothing to do with drawing circles. It would be nice if we didn't have to do this.

Extending a Class

In fact, we don't have to do it this way. Java allows us to define `GraphicCircle` as an extension, or *subclass* of `Circle`. Example 3.10 shows how. Note that this example assumes we have two other classes of objects defined: `Color`, which represents a color, and `DrawWindow`, a class that has the window into which drawing is done and that defines the primitive methods to do the drawing.

Example 3.10: Subclassing a Class

```
public class GraphicCircle extends Circle {
    // We automatically inherit the variables and methods of
    // Circle, so we only have to put the new stuff here.
    // We've omitted the GraphicCircle constructor, for now.
    Color outline, fill;
    public void draw(DrawWindow dw) {
        dw.drawCircle(x, y, r, outline, fill);
    }
}
```

The `extends` keyword tells Java that `GraphicCircle` is a subclass of `Circle`, and that it *inherits* the fields and methods of that class. [7] The definition of the `draw()` method shows variable inheritance--this method uses the `Circle` variables `x`, `y`, and `r` as if they were defined right in `GraphicCircle` itself.

> [7] Except for `private` fields and methods. We'll discuss `private` members of a class later. C++ programmers should note that `extends` is the Java equivalent of the `:` operator in C++--both indicate the superclass of a class.

`GraphicCircle` also inherits the methods of `Circle`. Thus, if we have a `GraphicCircle` object referred to by variable `gc`, we can say:

```
double area = gc.area();
```

This works just as if the `area()` method were defined in `GraphicCircle` itself.

Another feature of subclassing is that every `GraphicCircle` object is also a perfectly legal `Circle` object. Thus, if `gc` refers to a `GraphicCircle` object, we can assign it to a `Circle` variable, and we can forget all about its extra graphic capabilities: `Circle c = gc;`.

Final Classes

When a class is declared with the `final` modifier, it means that it cannot be extended or subclassed. `java.lang.System` is an example of a `final` class. Declaring a class `final` prevents unwanted extensions to the class. But it also allows the compiler to make some optimizations when invoking the methods of a class. We'll explore this in more detail when we talk about method overriding later in this chapter.

Superclasses, Object, and the Class Hierarchy

In our example, `GraphicCircle` is a subclass of `Circle`. We can also say that `Circle` is the *superclass* of `GraphicCircle`. The superclass of a class is specified in its `extends` clause:

```
public class GraphicCircle extends Circle { ... }
```

Every class you define has a superclass. If you do not specify the superclass with an `extends` clause, the superclass is the class `Object`. `Object` is a special class for a couple of reasons:

- It is the only class that does not have a superclass.

- The methods defined by `Object` can be called by any Java object.

Because every class has a superclass, classes in Java form a *class hierarchy*, which can be represented as a tree with `Object` at its root. Figure 3.1 shows a class hierarchy diagram which includes our `Circle` and `GraphicCircle` classes, as well as some of the standard classes from the Java API.

Figure 3.1: A class hierarchy diagram

[Graphic: Figure 3-1]

The complete class hierarchy for the Java API is diagrammed in the figures of Part V, *API Quick Reference*.

Subclass Constructors

In Example 3.10 we left out the constructor method for our new `GraphicCircle` class. Let's implement that now. Here's one way:

```
public GraphicCircle(double x, double y, double r,
                                Color outline, Color fill)
{
    this.x = x;
    this.y = y;
    this.r = r;
    this.outline = outline;
    this.fill = fill;
}
```

This constructor relies on the fact that `GraphicCircle` inherits all the variables of `Circle` and simply initializes those variables itself. But this duplicates the code of the `Circle` constructor, and if `Circle` did more elaborate initialization, it could become quite wasteful. Furthermore, if the `Circle` class had internal `private` fields (discussed later), we wouldn't be able to initialize them like this. What we need is a way of calling a `Circle` constructor from within our `GraphicCircle` constructor. Example 3.11 shows how we can do this.

Example 3.11: Invoking a Superclass's Constructor

```
public GraphicCircle(double x, double y, double r,
                                Color outline, Color fill)
{
    super(x, y, r);
    this.outline = outline;
    this.fill = fill;
}
```

`super` is a reserved word in Java. One of its uses is that shown in the example--to invoke the constructor method of a superclass. Its use is analogous to the use of the `this` keyword to invoke one constructor method of a class from within another. Using `super` to invoke a constructor is subject to the same restrictions as using `this` to invoke a constructor:

- `super` may only be used in this way within a constructor method.
- The call to the superclass constructor must appear as the first statement within the constructor method. It must appear even before variable declarations.

Constructor Chaining

When you define a class, Java guarantees that the class's constructor method is called whenever an instance of that class is created. It also guarantees that the constructor is called when an instance of any subclass is created. In order to guarantee this second point, Java must ensure that every constructor method calls its superclass constructor method. If the first statement in a constructor is not an explicit call to a constructor of the superclass with the `super` keyword, then Java implicitly inserts the call `super()`--that is, it calls the superclass constructor with no arguments. If the superclass does not have a constructor that takes no arguments, this causes a compilation error.

There is one exception to the rule that Java invokes `super()` implicitly if you do not do so explicitly. If the first line of a constructor, `C1`, uses the `this()` syntax to invoke another constructor, `C2`, of the class, Java relies on `C2` to invoke the superclass constructor, and does not insert a call to `super()` into `C1`. Of course, if `C2` itself uses `this()` to invoke a third constructor, `C2` does not call `super()` either, but somewhere along the chain, a constructor either explicitly or implicitly invokes the superclass constructor, which is what is required.

Consider what happens when we create a new instance of the `GraphicCircle` class. First, the `GraphicCircle` constructor shown in Example 3.11 is invoked. This constructor explicitly invokes a `Circle` constructor and that `Circle` constructor implicitly calls `super()` to invoke the constructor of its superclass, `Object`. The body of the `Object` constructor runs first, followed by the body of the `Circle` constructor, and finally followed by the body of the `GraphicCircle` constructor.

What this all means is that constructor calls are "chained"--any time an object is created, a sequence of constructor methods are invoked, from subclass to superclass on up to `Object` at the root of the class hierarchy. Because a superclass constructor is always invoked as the first statement of its subclass constructor, the body of the `Object` constructor always runs first, followed by the body of its subclass, and on down the class hierarchy to the class that is being instantiated.

The Default Constructor

There is one missing piece in the description of constructor chaining above. If a constructor does not invoke a superclass constructor, Java does so implicitly. But what if a class is declared without any constructor at all? In this case, Java implicitly adds a constructor to the class. This default constructor does nothing but invoke the superclass constructor.

For example, if we did not declare a constructor for the `GraphicCircle` class, Java would have implicitly inserted this constructor:

```
public GraphicCircle() { super(); }
```

Note that if the superclass, `Circle()`, did not declare a no-argument constructor, then this automatically inserted default constructor would cause a compilation error. If a class does not define a no-argument constructor, then all of its subclasses must define constructors that explicitly invoke the superclass constructor with the necessary arguments.

It can be confusing when Java implicitly calls a constructor or inserts a constructor definition into a class--something is happening that does not appear in your code! Therefore, it is good coding style, whenever you rely on an implicit superclass constructor call or on a default constructor, to insert a comment noting this fact. Your comments might look like those in the following example:

```
class A {
    int i;
    public A() {
        // Implicit call to super(); here.
        i = 3;
    }
}
class B extends A {
    // Default constructor: public B() { super(); }
}
```

If a class does not declare any constructor, it is given a `publicly` constructor by default. Classes that do not want to be publically instantiated, should declare a `protected` constructor to prevent the insertion of this `public` constructor. Classes that never want to be instantiated at all should define a `private` constructor.

Finalizer Chaining?

You might assume that since Java chains constructor methods that it also automatically chains the finalizer methods for an object. In other words, you may think that the finalizer method of a class automatically invokes the finalizer of its superclass. In fact, Java does *not* do this. In practice, finalizer methods are relatively rare, and the need for finalizer chaining rarely arises. If a class `B` with a finalizer method is a subclass of a class `A` with its own finalizer method, then `B`'s finalizer should be sure to invoke `A`'s finalizer, explicitly creating a chain of finalizers. This is a little tricky, since finalizers always have the same name (`finalize()`), and we haven't yet learned how to invoke a method in the superclass when that method is also defined in the subclass. We'll return to the issue of finalizer chaining when we learn how.

Shadowed Variables

Suppose that our `GraphicCircle` class has a new variable that specifies the resolution, in dots per inch, of the `DrawWindow` object in which it is going to be drawn. And further, suppose that it names that new variable `r`:

```
public class GraphicCircle extends Circle {
    Color outline, fill;
    float r;  // New variable.  Resolution in dots-per-inch.
    public GraphicCircle(double x, double y, double rad, Color o, Color f) {
        super(x, y, rad); outline = o; fill = f;
    }
    public void setResolution(float resolution) { r = resolution; }
    public void draw(DrawWindow dw) { dw.drawCircle(x, y, r, outline, fill); }
}
```

Now, with this resolution variable declared, when we use the variable `r` in the `GraphicCircle` class, we are no longer referring to the radius of the circle. The resolution variable `r` in `GraphicCircle` *shadows* the radius variable `r` in `Circle`. [8]

> [8] This is a contrived example, of course--we could simply rename the variable and avoid the issue. Typically we would rename the variable: variable shadowing is a necessary part of Java syntax, but is not a useful programming technique. Your code will be easier to understand if you avoid shadowed variables.

So, how can we refer to the radius variable defined in the `Circle` class when we need it? Recall that using a variable, such as `r`, in the class in which it is defined is shorthand for:

```
this.r  // Refers to the GraphicCircle resolution variable.
```

As you might guess, you can refer to a variable `r` defined in the superclass like this:

```
super.r  // Refers to the Circle radius variable.
```

Another way you can do this is to cast `this` to the appropriate class and then access the variable:

```
((Circle) this).r
```

This cast is exactly what the `super` keyword does when used like this. You can use this casting technique when you need to refer to a shadowed variable defined in a class that is not the immediate superclass. For example, if `C` is a subclass of `B`, which is a subclass of `A`, and class `C` shadows a variable `x` that is also defined in classes `A` and `B`, then you can refer to these different variables from class `C` as follows:

```
x                // Variable x in class C.
this.x           // Variable x in class C.
```

```
super.x           // Variable x in class B.
((B)this).x       // Variable x in class B.
((A)this).x       // Variable x in class A.
super.super.x     // Illegal; does not refer to x in class A.
```

Note that you cannot refer to a shadowed variable `x` in the superclass of a superclass with `super.super.x`. Java does not recognize this syntax.

Shadowed Methods?

Just as a variable defined in one class can shadow a variable with the same name in a superclass, you might expect that a method in one class could shadow a method with the same name (and same arguments) in a superclass. In a sense, they do: "shadowed" methods are called overridden methods. But method overriding is significantly different than variable shadowing; it is discussed in the sections that follow.

PREVIOUS	HOME	NEXT
Object Destruction	BOOK INDEX	Overriding Methods

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

3.7 Overriding Methods

When a class defines a method using the same name, return type, and arguments as a method in its superclass, the method in the class *overrides* the method in the superclass. When the method is invoked for an object of the class, it is the new definition of the method that is called, not the superclass' old definition.

Method overriding is an important and useful technique in object-oriented programming. Suppose we define a subclass `Ellipse` of our `Circle` class. [9] Then it would be important for `Ellipse` to override the `area()` and `circumference()` methods of `Circle`. `Ellipse` would have to implement new versions of these functions because the formulas that apply to circles don't work for ellipses.

> [9] This is admittedly a strange thing to do, since, mathematically, a circle is a kind of ellipse, and it is customary to derive a more specific class from a more general one. Nevertheless, it is a useful example here.

Before we go any further with the discussion of method overriding, be sure that you understand the difference between method overriding and method overloading, which we discussed earlier. As you probably recall, method overloading refers to the practice of defining multiple methods (in the same class) with the same name but with differing argument lists. This is very different from method overriding, and it is important not to get them confused!

Overriding Is Not Shadowing

Although Java treats the variables and methods of a class analogously in many ways, method overriding is not like variable shadowing at all: You can refer to shadowed variables simply by casting an object to the appropriate type. You cannot invoke overridden methods with this technique, however. Example 3.12 illustrates this crucial difference.

Example 3.12: Method Overriding versus Variable Shadowing

```
class A {
    int i = 1;
    int f() { return i; }
}
class B extends A {
    int i = 2;                              // Shadows variable i in class A.
    int f() { return -i; }                  // Overrides method f in class A.
}
public class override_test {
    public static void main(String args[]) {
        B b = new B();
        System.out.println(b.i);     // Refers to B.i; prints 2.
        System.out.println(b.f());   // Refers to B.f(); prints -2.
        A a = (A) b;                 // Cast b to an instance of class A.
        System.out.println(a.i);     // Now refers to A.i; prints 1;
        System.out.println(a.f());   // Still refers to B.f(); prints -2;
    }
}
```

While this difference between method overriding and variable shadowing may seem surprising at first, a little thought makes the purpose clear. Suppose we have a bunch of `Circle` and `Ellipse` (a subclass of `Circle`) objects that we are manipulating. To keep track of the circles and ellipses, we store them in an array of type `Circle[]`, casting all the `Ellipse` objects to `Circle` objects before we store them. Then, when we loop through the elements of this array, we don't have to know or care whether the element is actually a `Circle` or an `Ellipse`. What we do care very much about, however, is that the correct value is computed when we invoke the `area()` method of any element of the array. That is, we don't want to use the formula for the area of a circle when the object is actually an ellipse!

Seen in this context, it is not surprising at all that method overriding is handled differently by Java than variable shadowing.

final Methods

If a method is declared `final`, it means that the method declaration is the "final" one--that it cannot be overridden. `static` methods and `private` methods (which we haven't learned about yet) cannot be overridden either, nor can the methods of a `final` class. If a method cannot be overridden, the compiler may perform certain optimizations on it, as we'll see below.

Dynamic Method Lookup

If we have an array of `Circle` and `Ellipse` objects, how does the compiler know to call the `Circle area()` method or the `Ellipse area()` method for any given item in the array? The compiler does not know this; it can't. The compiler knows that it does not know, however, and produces code that uses "dynamic method lookup" at run-time. When the interpreter runs the code, it looks up the appropriate `area()` method to call for each of the objects. That is, when the interpreter interprets the expression s.area(), it dynamically looks for an area() method associated with the particular object referred to by the variable s. It does not simply use the area() method that is statically associated with the type of the variable s. [10]

[10] C++ programmers should note that dynamic method lookup is what C++ does for virtual functions. An important difference between Java and C++ is that Java does not have a virtual keyword; methods in Java are "virtual" by default.

Dynamic method lookup is fast, but it is not as fast as invoking a method directly. Fortunately, there are a number of cases in which Java does not need to use dynamic method lookup. static methods cannot be overridden, so they are always invoked directly. private methods (which we haven't learned about yet) are not inherited by subclasses and so cannot be overridden by subclasses; this means the Java compiler can safely invoke them without dynamic method lookup as well. final methods are invoked directly for the same reason: they cannot be overridden. Finally, when a method of a final class is invoked through an instance of the class or a subclass of it, then it, too, can be invoked without the overhead of dynamic lookup. These static, final, and private methods that can be invoked directly are also candidates for inlining--i.e., if the methods are short, the compiler may simply insert the method body into the code rather than inserting a call to the method.

Invoking an Overridden Method

We've seen the important differences between method overriding and variable shadowing. Nevertheless, the Java syntax for invoking an overridden method is very similar to the syntax for accessing a shadowed variable: both use the super keyword. Example 3.13 illustrates this.

Example 3.13: Invoking an Overridden Method

```
class A {
    int i = 1;
    int f() { return i; }        // A very simple method.
}
class B extends A {
    int i;                       // This variable shadows i in A.
    int f() {                    // This method overrides f() in A.
        i = super.i + 1;         // It retrieves A.i this way.
        return super.f() + i;    // And it invokes A.f() this way.
    }
}
```

Recall that when you use super to refer to a shadowed variable, it is the same as casting this to the superclass type and accessing the variable through that. On the other hand, using super to invoke an overridden method is not the same as casting this. In this case, super has the special purpose of turning off dynamic method lookup and invoking the specific method that the superclass defines or inherits.

In Example 3.13 we use `super` to invoke an overridden method that is actually defined in the immediate superclass. `super` also works perfectly well to invoke overridden methods that are defined further up the class hierarchy. This is because the overridden method is inherited by the immediate superclass, and so the `super` syntax does in fact refer to the correct method.

To make this more concrete, suppose class `A` defines method `f`, and that `B` is a subclass of `A`, and that `C` is a subclass of `B` that overrides method `f`. Then you can still use:

```
super.f()
```

to invoke the overridden method from within class `C`. This is because class `B` inherits method `f` from class `A`. If classes `A`, `B`, and `C` all define method `f`, however, then calling `super.f()` in class `C` invokes class `B`'s definition of the method. In this case, there is no way to invoke `A.f()` from within class `C`. `super.super.f()` is not legal Java syntax!

It is important to note that `super` can only be used to invoke overridden methods from within the class that does the overriding. With our `Circle` and `Ellipse` classes, for example, there is no way to write a program (with or without `super`) that invokes the `Circle` `area()` method on an object of type `Ellipse`. The only way to do this is to use `super` in a method within the `Ellipse` class.

Finally, note that this form of `super` does not have to occur in the first statement in a method, as it does when used to invoke a superclass constructor method.

Finalizer Chaining Revisited

Now that we've discussed method overriding and how to invoke an overridden method, we can return to the issue of the finalizer method that we left dangling earlier on.

In Java, constructor methods are automatically chained, but finalizer methods are not. If you define a `finalize()` method to free resources allocated by your class, you may be overriding a `finalize()` method in a superclass that frees resources allocated by that class. If your finalizer method does not invoke the superclass finalizer, the superclass finalizer never gets called, and resources are not deallocated when they should be.

To prevent this, you should be sure to invoke the superclass `finalize()` method. The best time to do this is usually after your `finalize()` method has done all of its deallocation. It is a good idea to add the following call:

```
super.finalize();
```

as the last line of all your finalizer methods. You should do this even if you know that none of your class's superclasses have finalizer methods, because future implementations of the class may include one.

3.8 Data Hiding and Encapsulation

We started this chapter by describing a class as "a collection of data and methods." One of the important object-oriented techniques that we haven't discussed so far is hiding the data within the class, and making it available only through the methods. This technique is often known as *encapsulation*, because it seals the class's data (and internal methods) safely inside the "capsule" of the class, where it can be accessed only by trusted users--i.e., by the methods of the class.

Why would you want to do this? The most important reason is to hide the internal implementation details of your class. If you prevent programmers from relying on those details, then you can safely modify the implementation without worrying that you will break existing code that uses the class.

Another reason for encapsulation is to protect your class against accidental or willful stupidity. A class often contains a number of variables that are interdependent and must be in a consistent state. If you allow a programmer (this may be you yourself) to manipulate those variables directly, she may change one variable without changing important related variables, thus leaving the class in an inconsistent state. If, instead, she had to call a method to change the variable, that method can be sure to do everything necessary to keep the state consistent.

Here's another way to think about encapsulation: When all of a class's variables are hidden, the class's methods define the only possible operations that can be performed on objects of that class. Once you have carefully tested and debugged your methods, you can be confident that the class will work as expected. On the other hand, if all the variables can be directly manipulated, the number of possibilities you have to test becomes unmanageable.

There are other reasons to hide data, too:

- Internal variables that are visible externally to the class just clutter up your class's API. Keeping visible variables to a minimum keeps your class tidy and elegant.
- If a variable is visible in your class, you have to document it. Save time by hiding it instead!

Visibility Modifiers

In most of our examples so far, you've probably noticed the `public` keyword being used. When applied to a class, it means that the class is visible everywhere. When applied to a method or variable, it means that the method or variable is visible everywhere.

To hide variables (or methods, for that matter) you just have to declare them `private`:

```
public class Laundromat {       // People can use this class.
    private Laundry[] dirty;    // They can't see this internal variable,
    public void wash() { ... }  // but they can use these public methods
    public void dry() { ... }   // to manipulate the internal variable.
}
```

A `private` field of a class is visible only in methods defined within that class. (Or, in Java 1.1, in classes defined within the class.) Similarly, a `private` method may only be invoked by methods within the class (or methods of classes within the class). Private members are not visible within subclasses, and are *not* inherited by subclasses as other members are. [11] Of course, non-`private` methods that invoke `private` methods internally are inherited and may be invoked by subclasses.

> [11] Every object does, of course, have its own copy of all fields of all superclasses, including the `private` fields. The methods defined by the object can never refer to or use the `private` fields of superclasses, however, and so we say that those fields are not inherited.

Besides `public` and `private`, Java has two other visibility levels: `protected` and the default visibility level, "package visibility," which applies if none of the `public`, `private`, and `protected` keywords are used.

A `protected` member of a class is visible within the class where it is defined, of course, and within all subclasses of the class, and also within all classes that are in the same package as that class. You should use `protected` visibility when you want to hide fields and methods from code that uses your class, but want those fields and methods to be fully accessible to code that extends your class.

The default package visibility is more restrictive than `protected`, but less restrictive than `private`. If a class member is not declared with any of the `public`, `private`, or `protected` keywords, then it is visible only within the class that defines it and within classes that are part of the same package. It is not visible to subclasses unless those subclasses are part of the same package.

A note about packages: A package is a group of related and possibly cooperating classes. All non-`private` members of all classes in the package are visible to all other classes in the package. This is okay because the classes are assumed to know about, and trust, each other. [12] The only time difficulty arises is when you write programs without a `package` statement. These classes are thrown into a default package with other `package`-less classes, and all their non-`private` members are visible throughout the package. (The default package usually consists of all classes in the current working directory.)

> [12] If you're a C++ programmer, you can say that classes within the same package are `friend`-ly to each other.

There is an important point to make about subclass access to `protected` members. A subclass inherits the `protected` members of its superclass, but it can only access those members through instances of itself, not directly in instances of the superclass. Suppose, for example, that `A`, `B`, and `C` are `public` classes, each defined in a different package, and that `a`, `b`, and `c` are instances of those classes. Let `B` be a subclass of `A`, and `C` be a subclass of `B`. Now, if `A` has a `protected` field `x`, then the class B inherits that field, and its methods can use `this.x`, `b.x` and `c.x`. But it cannot access `a.x`. Similarly, if `A` has a `protected` method `f()`, then the methods of class `B` can invoke `this.f()`, `b.f()`, and `c.f()`, but they cannot invoke `a.f()`.

Table 3.1 shows the circumstances under which class members of the various visibility types are accessible to other classes.

Table 3.1: Class Member Accessibility

Accessible to:	**public**	**protected**	**package**	**private**
Same class	yes	yes	yes	yes
Class in same package	yes	yes	yes	no
Subclass in different package	yes	yes	no	no
Non-subclass, different package	yes	no	no	no

The details of member visibility in Java can become quite confusing. Here are some simple rules of thumb for using visibility modifiers:

- Use `public` only for methods and constants that form part of the public API of the class. Certain very important or very frequently used fields may also be `public`, but it is common practice to make fields non-`public` and encapsulate them with `public` accessor methods.

- Use `protected` for fields and methods that aren't necessary to use the class, but that may be of interest to anyone creating a subclass as part of a different package.

- Use the default package visibility for fields and methods that you want to be hidden outside of the package, but which you want cooperating classes within the same package to have access to.

- Use `private` for fields and methods that are only used inside the class and should be hidden everywhere else.

Note that you can't take advantage of package visibility unless you use the `package` statement to group your related classes into packages. See Chapter 13, *Java Syntax*, *Java Syntax*, for a table that summarizes the Java visibility modifiers and other modifiers.

Data Access Methods

In the `Circle` example we've been using, we've declared the circle position and radius to be `public` fields. In fact, the `Circle` class is one where it may well make sense to keep those visible--it is a simple enough class, with no dependencies between the variables.

On the other hand, suppose we wanted to impose a maximum radius on objects of the `Circle` class. Then it would be better to hide the `r` variable so that it could not be set directly. Instead of a visible `r` variable, we'd implement a `setRadius()` method that verifies that the specified radius isn't too large and then sets the `r` variable internally. Example 3.14 shows how we might implement `Circle` with encapsulated data and a restriction on radius size. For convenience, we use `protected` fields for the radius and position variables. This means that subclasses of

`Circle`, or cooperating classes within the `shapes` package are able to access these variables directly. To any other classes, however, the variables are hidden. Also, note the `private` constant and method used to check whether a specified radius is legal. And finally, notice the `public` methods that allow you to set and query the values of the instance variables.

Example 3.14: Hiding Variables in the Circle Class

```
package shapes;              // Specify a package for the class.
public class Circle {        // Note that the class is still public!
    protected double x, y;   // Position is hidden, but visible to subclasses.
    protected double r;      // Radius is hidden, but visible to subclasses.
    private static final double MAXR = 100.0;    // Maximum radius (constant).
    private boolean check_radius(double r) { return (r <= MAXR); }
    // Public constructors
    public Circle(double x, double y, double r) {
        this.x = x; this.y = y;
        if (check_radius(r)) this.r = r;
        else this.r = MAXR;
    }
    public Circle(double r) { this(0.0, 0.0, r); }
    public Circle() { this(0.0, 0.0, 1.0); }
    // Public data access methods
    public void moveto(double x, double y) { this.x = x; this.y = y;}
    public void move(double dx, double dy) { x += dx;  y += dy; }
    public void setRadius(double r) { this.r = (check_radius(r))?r:MAXR; }
    // Declare these trivial methods final so we don't get dynamic
    // method lookup and so that they can be inlined by the compiler.
    public final double getX() { return x; };
    public final double getY() { return y; };
    public final double getRadius() { return r; };
}
```

3.9 Abstract Classes and Interfaces

In Example 3.14 we declared our `Circle` class to be part of a package named `shapes`. Suppose we plan to implement a number of shape classes: `Rectangle`, `Square`, `Ellipse`, `Triangle`, and so on. We'll give all of these shape classes our two basic `area()` and `circumference()` methods. Now, to make it easy to work with an array of shapes, it would be helpful if all our shape classes have a common superclass, `Shape`. We want `Shape` to encapsulate whatever features all our shapes have in common. In this case, they have in common the `area()` and `circumference()` methods. But our generic `Shape` class can't actually implement these methods, since it doesn't represent any actual shape. Java handles this case with abstract methods.

Abstract Methods

Java lets us define a method without implementing it by making the method `abstract`. An `abstract` method has no body; it simply has a signature definition followed by a semicolon. [13] Here are the rules about `abstract` methods, and the `abstract` classes that contain them:

[13] An `abstract` method in Java is something like a "pure virtual function" in C++ (i.e., a virtual function that is declared `= 0`). In C++, a class that contains a pure virtual function is called an "abstract class" and may not be instantiated. The same is true of Java classes that contain `abstract` methods.

- Any class with an `abstract` method is automatically `abstract` itself, and must be declared as such.
- A class may be declared `abstract` even if it has no `abstract` methods. This prevents it from being instantiated.
- An `abstract` class cannot be instantiated.
- A subclass of an `abstract` class can be instantiated if it overrides each of the `abstract` methods of its superclass and provides an implementation (i.e., a method body) for all of them.
- If a subclass of an `abstract` class does not implement all of the `abstract` methods it inherits, that subclass is itself `abstract`.

That description of the `abstract` keyword was pretty abstract! Example 3.15 is more concrete. It shows an `abstract Shape` class and two non-`abstract` subclasses of it.

Example 3.15: An Abstract Class and Subclasses

```
public abstract class Shape {
    public abstract double area();
    public abstract double circumference();
}
class Circle extends Shape {
    protected double r;
    protected static final double PI = 3.14159265358979323846;
    public Circle() { r = 1.0; }
    public Circle(double r) { this.r = r; }
    public double area() { return PI * r * r; }
    public double circumference() { return 2 * PI * r; }
    public double getRadius() { return r; }
}
class Rectangle extends Shape {
    protected double w, h;
    public Rectangle() { w = 0.0; h = 0.0; }
    public Rectangle(double w, double h) { this.w = w;  this.h = h; }
    public double area() { return w * h; }
    public double circumference() { return 2 * (w + h); }
    public double getWidth() { return w; }
    public double getHeight() { return h; }
}
```

Note that the `abstract` methods in `Shape` have a semicolon right after their parentheses. There are no curly braces, and no method body is defined. Using the classes defined in Example 3.15, we can now write code like this:

```
Shape[] shapes = new Shape[3];          // Create an array to hold shapes.
shapes[0] = new Circle(2.0);            // Fill in the array...
shapes[1] = new Rectangle(1.0, 3.0);
shapes[2] = new Rectangle(4.0, 2.0);
double total_area = 0;
for(int i = 0; i < shapes.length; i++)
    total_area += shapes[i].area();     // Compute the area of the shapes.
```

There are two important points to notice here:

- Subclasses of `Shape` can be assigned to elements of an array of `Shape`. No cast is necessary.

- You can invoke the `area()` and `circumference()` methods for `Shape` objects, even though `Shape` does not define a body for these methods, because `Shape` declared them `abstract`. If `Shape` did not declare them at all, the code would cause a compilation error.

Interfaces

Let's extend our `shapes` package further. Suppose we now want to implement a number of shapes that can be drawn on the screen. We could define an abstract `DrawableShape` class, and then implement various subclasses of it, such as `DrawableCircle`, `DrawableRectangle`, and so on. This would work fine.

But suppose we want our drawable shape types to support the `area()` and `circumference()` methods. We don't want to have to re-implement these methods, so we'd like to make `DrawableCircle` a subclass of `Circle`, for example, and `DrawableRectangle` a subclass of `Rectangle`. But classes in Java can only have one superclass. If `DrawableCircle` extends `Circle`, then it cannot also extend `DrawableShape`! [14]

> [14] C++ allows classes to have more than one superclass, using a technique known as "multiple inheritance." Multiple inheritance opens up a can of worms, so Java replaces it with what many believe is a more elegant solution.

Java's solution to this problem is called an *interface*. An interface looks a lot like an abstract class, except that it uses the keyword `interface` instead of the words `abstract` and `class`. Example 3.16 shows an interface named `Drawable`.

Example 3.16: An Interface Definition

```
public interface Drawable {
    public void setColor(Color c);
    public void setPosition(double x, double y);
    public void draw(DrawWindow dw);
}
```

While an `abstract` class may define some `abstract` methods and some non-`abstract` methods, all the methods defined within an interface are implicitly `abstract`. We've omitted the `abstract` keyword in this example, but it is perfectly legal to use it if you want to belabor the abstractness of interface methods. A further restriction on interfaces is that any variables declared in an interface must be `static` and `final`--that is, they must be constants.

So what can we do with an interface? Just as a class `extends` its superclass, it also optionally `implements` an interface. `implements` is a Java keyword that can appear in a class declaration following the `extends` clause. `implements` should be followed by the name of the interface that the class implements. In order to implement an interface, a class must first declare the interface in an `implements` clause, and then it must provide an implementation (i.e., a body) for all of the `abstract` methods of the interface. [15]

> [15] This is the real difference between multiple inheritance in C++ and interfaces in Java. In C++, a class can inherit method implementations from more than one superclass. In Java, a class can inherit actual implementations only from one superclass. It can inherit additional `abstract` methods from interfaces, but it must provide its own implementation of those methods. It is rare, however, to actually be able to use C++ multiple inheritance to inherit useful, non-trivial implementations from more than one class. The elegance and simplicity of Java's interface more than compensate for the inability to inherit implementations from more than one class.

Example 3.17 shows how we can define a `DrawableRectangle` class that extends our `Rectangle` class and implements the `Drawable` interface we defined in Example 3.16. The example assumes that a `Color` class and a `DrawWindow` class are defined elsewhere, and that `DrawWindow` knows how to convert floating-point coordinates to pixel coordinates and knows how to draw primitive shapes.

Example 3.17: Implementing an Interface

```
public class DrawableRectangle extends Rectangle implements Drawable {
    // New instance variables
    private Color c;
    private double x, y;
    // A constructor
    public DrawableRectangle(double w, double h) { super(w, h); }
    // Here are implementations of the Drawable methods.
    // We also inherit all the public methods of Rectangle.
    public void setColor(Color c) { this.c = c; }
    public void setPosition(double x, double y) { this.x = x; this.y = y; }
    public void draw(DrawWindow dw) {
        dw.drawRect(x, y, w, h, c);
    }
}
```

Using Interfaces

Suppose we implement `DrawableCircle` and `DrawableSquare` just as we implemented `DrawableRectangle` in Example 3.17. As we saw earlier, instances of these classes can be treated as instances of the abstract `Shape` class. They can also be treated as instances of the `Drawable` interface. Example 3.18 demonstrates this.

Example 3.18: Casting Objects to Their Interface Type

```
Shape[] shapes = new Shape[3];          // Create an array to hold shapes
Drawable[] drawables = new Drawable[3]; // and an array to hold drawables.
// Create some drawable shapes.
DrawableCircle dc = new DrawableCircle(1.1);
DrawableSquare ds = new DrawableSquare(2.5);
DrawableRectangle dr = new DrawableRectangle(2.3, 4.5);
// The shapes can be assigned to both arrays.
shapes[0] = dc;   drawables[0] = dc;
shapes[1] = ds;   drawables[1] = ds;
shapes[2] = dr;   drawables[2] = dr;
// Compute total area and draw the shapes by invoking
// the Shape and the Drawable abstract methods.
double total_area = 0;
for(int i = 0; i < shapes.length; i++) {
    total_area += shapes[i].area();    // Compute the area of the shapes.
    drawables[i].setPosition(i*10.0, i*10.0);
    drawables[i].draw(draw_window);    // Assume draw_window defined somewhere.
}
```

What this example demonstrates is that interfaces are data types in Java, just as classes are, and that when a class implements an interface, instances of that class can be assigned to variables of the interface type. Don't interpret this example to imply that you must assign a `DrawableRectangle` object to a `Drawable` variable before you can invoke the `draw()` method or that you must assign it to a `Shape` variable before you can invoke the `area()` method. `DrawableRectangle` defines `draw()` and inherits `area()` from its `Rectangle` superclass, and so you can always invoke these methods.

Implementing Multiple Interfaces

Suppose we wanted shapes that could be scaled to be larger or smaller. One way we could do this is by defining a `Scalable` interface and implementing subclasses of `DrawableRectangle` and the other classes. To do this, though, the new subclass would have to implement both the `Drawable` interface and the `Scalable` interface. This is no problem. You may specify a list of comma-separated interfaces in the `implements` clause of any class:

```
public class DrawableScalableRectangle extends DrawableRectangle
                implements Drawable, Scalable {
    // The methods of the Scalable interface must be implemented here.
}
```

When a class implements more than one interface, it means simply that it must provide an implementation for all of the abstract methods in all of its interfaces.

Constants in Interfaces

As we noted above, constants may appear in interface definitions. What does it mean to implement an interface that contains constants? It simply means that the class that implements the interface "inherits" the constants (in a sense) and can use them as if they were defined directly in the class. There is no need to prefix them with the name of the interface, and there is no need to provide an "implementation" of the constants:

```
class A { static final int CONSTANT1 = 3; }
interface B { static final int CONSTANT2 = 4; }
class C implements B {
    void f() {
        int i = A.CONSTANT1;  // Have to use the class name here.
        int j = CONSTANT2;    // No class name here, because we implement
    }                         // the interface that defines this constant.
}
```

When you have a set of constants used by more than one class within a package (for example, a port number and other protocol constants used by a client and server), it is convenient to define them in an interface that contains no abstract `methods`. Then, any class that wants to use those constants needs only to declare that it implements the interface.

Extending Interfaces

Interfaces can have sub-interfaces, just as classes can have subclasses. A sub-interface inherits all the abstract methods and constants of its super-interface, and may define new abstract methods and constants. Interfaces are different from classes in one very important way, however.

An interface can extend more than one interface at a time:

```
public interface Transformable extends Scalable, Rotateable, Reflectable { }
public interface DrawingObject extends Drawable, Transformable { }
public class Shape implements DrawingObject { ... }
```

An interface that extends more than one interface inherits all the abstract methods and constants from each of those interfaces, and may define its own additional abstract methods and constants. A class that implements such an interface must implement the abstract methods defined in the interface itself as well as all the abstract methods inherited from all the super-interfaces.

Marker Interfaces

Another technique that is sometimes useful is to define an interface that is entirely empty. A class can implement this interface to provide additional information about itself. The `Cloneable` interface in `java.lang` is an example of this type of "marker interface." It defines no methods, but serves simply to identify the class as one that will allow its internal state to be cloned by the `clone()` method of the `Object` class. In Java 1.1, `java.io.Serializable` is another such marker interface. You can test whether a class implements a marker interface (or any interface) using the `instanceof` operator.

PREVIOUS
Data Hiding and Encapsulation

HOME
BOOK INDEX

NEXT
C++ Features Not Found in Java

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

3.10 C++ Features Not Found in Java

Throughout this chapter, we've noted similarities and differences between Java and C++ in footnotes. Java shares enough concepts and features with C++ to make it an easy language for C++ programmers to pick up. There are several features of C++ that have no parallel in Java, however. In general, Java does not adopt those features of C++ that make the language significantly more complicated. These omissions from Java (or simplifications of C++) are described below.

C++ supports "multiple inheritance" of method implementations from more than one superclass at a time. While this seems like a very useful feature, adding it to the language actually turns out to introduce many complexities. The Java language designers chose to avoid the added complexity by using interfaces instead. Thus, a class in Java can only inherit method implementations from a single superclass, but it can inherit method declarations from any number of interfaces. In practice, this is not any particular hardship.

C++ supports (though not yet in a very standardized way) templates that allow you, for example, to implement a `Stack` class and then instantiate it as `Stack<int>` or `Stack<double>` to produce two separate types: a stack of integers and a stack of floating-point values. Java has no such facility. However, the fact that every class in Java is a subclass of `Object` means that every object can be cast to an instance of `Object`. Thus, in Java, it is often sufficient to define a data structure (such as a `Stack` class) that operates on `Object` values--the objects can be cast back to their actual type whenever necessary.

C++ allows you to define operators that perform arbitrary operations on instances of your classes. In effect, it allows you to extend the syntax of the language. This is a nifty feature, called operator overloading, that makes for very elegant examples. In practice, however, it tends to make code hard to understand. After much debate, the Java language designers decided to omit such "operator overloading" from the language. Note, though, that the use of the `+` operator for string concatenation in Java is at least reminiscent of operator overloading.

C++ allows you to define "conversion functions" for a class that automatically invoke an appropriate constructor method when a value is assigned to a variable of that class. This is simply a syntactic shortcut (similar to overriding the assignment operator) and is not included in Java.

In C++, objects are by default manipulated by value; you must use & to specify a variable or function argument that is automatically manipulated by reference. In Java, all objects are manipulated by reference, so there is no need for this & syntax.

PREVIOUS HOME NEXT
Abstract Classes and Interfaces BOOK INDEX Summary

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Examples From *Java in a Nutshell, Second Edition*

The Java programming examples linked below are from the book *Java in a Nutshell, Second Edition*, by David Flanagan, published by O'Reilly & Associates.

Although you can view the example source code online, by following the links below, I recommend that you download the complete set of examples so that you can work with them on your computer locally. They are available as a zip file or as a gzipped tar file.

Reporting Bugs

If you find any bugs in these examples, please send e-mail describing the bug to bookquestions@ora.com. If you have found a workaround to the problem, please include it.

Copyright

Example 1-2

Scribble.java: an applet of intermediate complexity, used as an example in the introductory chapter.

Example 2-3

throwtest.java: an application that demonstrates how to define, throw, and handle exceptions. This application doesn't do anything other than print out some text, but you might want to study it and play around with it to learn more about how exceptions work in Java. See the usage instructions in the source code.

Example 6-1

FirstApplet.java: the simplest possible applet. Displays "Hello World"

Example 6-2

SecondApplet.java: a fancier version of "Hello World"

Example 6-3

Scribble.java: a simple applet with user interaction. It allows the user to click and scribble in the window.

Example 6-4

ColorScribble.java: the scribble applet, with colors specified through applet parameters in an HTML file.

Example 6-5

Soundmap.java: An applet that displays an image, plays a sound, and demonstrates several other applet capabilities.

Example 7-1

Scribble1.java: a simple applet, using the Java 1.0 event model.

Example 7-2

Scribble2.java: the same applet, using the Java 1.1 event model.

Example 7-3

Scribble3.java: the applet using the Java 1.1 event model and inner classes.

Example 7-4

Scribble4.java: the applet using a low-level interface to the Java 1.1 event model.

Example 8-1

ScribbleFrame.java: a relatively long application that demonstrates many of the new AWT features of Java 1.1, and also demonstrates object serialization and data compression.

Example 9-2

IntList.java: a simple datatype that defines custom serialziation and de-serialization behavior for itself.

Example 10-1

MultiLineLabel.java: a custom AWT component and Java Bean that displays a specified string of text, using multiple lines, if the string contains newline characters.

Example 10-2

YesNoDialog.java: a bean that displays a dialog box.

Example 10-3

AnswerEvent.java: an event type used by the bean.

Example 10-4

AnswerListener.java: the event listener interface used by the bean

Example 10-5

YesNoDialogBeanInfo.java: a BeanInfo class for the bean.

Example 10-6

YesNoDialogAlignmentEditor.java: a property editor class for one of the bean's properties.

Example 10-7

YesNoDialogMessageEditor.java: a property editor class for another of the bean's properties.

Example 10-8

YesNoDialogCustomizer.java: a customizer class for the bean.

Example 11-1

ConvertEncoding.java: an application that converts a file from one character encoding to another.

Example 11-3

Portfolio.java: a dummy stock portfolio program that demonstrates internationalization of dates, times and numbers.

Example 11-4

SimpleMenu.java: a convenience class for simple creation of localized menus using ResourceBundles.

Example 11-5

Menus.properties Menus_en_GB.properties Menus_fr.properties: property files that specify default, British, and French resource bundles for simple menu creation.

Example 11-6

LocalizedError.java: a class that displays a localized error message fora given exception object, using the MessageFormat class.

Example 11-7

Errors.properties: a sample property file used by the previous example

Example 12-1

ShowClass.java: a program that uses the Reflection API to show the fields and methods of a class.

Example 12-2

UniversalActionListener.java: an ActionListener implementation that uses reflection to invoke a named method of an object.

```
// This example is from the book "Java in a Nutshell, Second Edition".
// Written by David Flanagan.  Copyright (c) 1997 O'Reilly & Associates.
// You may distribute this source code for non-commercial purposes only.
// You may study, modify, and use this example for any purpose, as long as
// this notice is retained.  Note that this example is provided "as is",
// WITHOUT WARRANTY of any kind either expressed or implied.

import java.applet.*;
import java.awt.*;

public class Scribble extends Applet {
  private int last_x, last_y;                // Store the last mouse position.
  private Color current_color = Color.black; // Store the current color.
  private Button clear_button;               // The clear button.
  private Choice color_choices;              // The color dropdown list.

  // This method is called to initialize the applet.
  // Applets don't have a main() method.
  public void init() {
    // Set the background color
    this.setBackground(Color.white);

    // Create a button and add it to the applet.  Set the button's colors
    clear_button = new Button("Clear");
    clear_button.setForeground(Color.black);
    clear_button.setBackground(Color.lightGray);
    this.add(clear_button);

    // Create a menu of colors and add it to the applet.
    // Also set the menus's colors and add a label.
    color_choices = new Choice();
    color_choices.addItem("black");
    color_choices.addItem("red");
    color_choices.addItem("yellow");
    color_choices.addItem("green");
    color_choices.setForeground(Color.black);
    color_choices.setBackground(Color.lightGray);
    this.add(new Label("Color: "));
    this.add(color_choices);
  }

  // This method is called when the user clicks the mouse to start a scribble.
  public boolean mouseDown(Event e, int x, int y)
  {
    last_x = x; last_y = y;
    return true;
  }

  // This method is called when the user drags the mouse.
  public boolean mouseDrag(Event e, int x, int y)
  {
    Graphics g = this.getGraphics();
    g.setColor(current_color);
    g.drawLine(last_x, last_y, x, y);
    last_x = x;
    last_y = y;
    return true;
  }

  // This method is called when the user clicks the button or chooses a color
  public boolean action(Event event, Object arg) {
    // If the Clear button was clicked on, handle it.
    if (event.target == clear_button) {
      Graphics g = this.getGraphics();
      Rectangle r = this.bounds();
      g.setColor(this.getBackground());
      g.fillRect(r.x, r.y, r.width, r.height);
      return true;
    }
    // Otherwise if a color was chosen, handle that
    else if (event.target == color_choices) {
      if (arg.equals("black")) current_color = Color.black;
      else if (arg.equals("red")) current_color = Color.red;
      else if (arg.equals("yellow")) current_color = Color.yellow;
      else if (arg.equals("green")) current_color = Color.green;
      return true;
    }
    // Otherwise, let the superclass handle it.
    else return super.action(event, arg);
  }
}
```

```
// This example is from the book "Java in a Nutshell, Second Edition".
// Written by David Flanagan.  Copyright (c) 1997 O'Reilly & Associates.
// You may distribute this source code for non-commercial purposes only.
// You may study, modify, and use this example for any purpose, as long as
// this notice is retained.  Note that this example is provided "as is",
// WITHOUT WARRANTY of any kind either expressed or implied.

// Here we define some exception types of our own.
// Exception classes generally have constructors but no data or
// other methods.  All these do is to call their superclass constructors.
class MyException extends Exception {
    public MyException() { super(); }
    public MyException(String s) { super(s); }
}
class MyOtherException extends Exception {
    public MyOtherException() { super(); }
    public MyOtherException(String s) { super(s); }
}
class MySubException extends MyException {
    public MySubException() { super(); }
    public MySubException(String s) { super(s); }
}

// This class demonstrates defining, throwing and handling exceptions.
// Try invoking it in the following ways and try to understand the
// output:
//    java throwtest
//    java throwtest one
//    java throwtest 0
//    java throwtest 1
//    java throwtest 99
//    java throwtest 2
//    java throwtest 3
public class throwtest {
    // This is the main() method.  Note that it uses two
    // catch clauses to handle two standard Java exceptions.
    public static void main(String argv[]) {
        int i;

        // First, convert our argument to an integer
        // Make sure we have an argument and that it is convertible.
        try {
            i = Integer.parseInt(argv[0]);
        }
        catch (ArrayIndexOutOfBoundsException e) { // argv is empty
            System.out.println("Must specify an argument");
            return;
        }
        catch (NumberFormatException e) { // argv[0] isn't an integer
            System.out.println("Must specify an integer argument.");
            return;
        }

        // Now, pass that integer to method a().
        a(i);
    }

    // This method invokes b(), which is declared to throw
    // one type of exception.  We handle that one exception.
    public static void a(int i) {
        try {
            b(i);
        }
        catch (MyException e) {                              // Point 1.
            // Here we handle MyException and
            // its subclass MyOtherException
            if (e instanceof MySubException)
                System.out.print("MySubException: ");
            else
                System.out.print("MyException: ");
            System.out.println(e.getMessage());
            System.out.println("Handled at point 1");
        }
    }

    // This method invokes c(), and handles one of the
    // two exception types that that method can throw.  The other
    // exception type is not handled, and is propagated up
    // and declared in this method's throws clause.
    // This method also has a finally clause to finish up
    // the work of its try clause.  Note that the finally clause
    // is executed after a local catch clause, but before a
    // a containing catch clause or one in an invoking procedure.
    public static void b(int i) throws MyException {
        int result;
        try {
            System.out.print("i = " + i);
            result = c(i);
            System.out.print(" c(i) = " + result);
        }
        catch (MyOtherException e) {                          // Point 2
            // Handle MyOtherException exceptions:
            System.out.println("MyOtherException: " + e.getMessage());
            System.out.println("Handled at point 2");
        }
        finally {
            // Terminate the output we printed above with a newline.
            System.out.print("\n");
        }
    }

    // This method computes a value or throws an exception.
    // The throws clause only lists two exceptions, because
    // one of the exceptions thrown is a subclass of another.
    public static int c(int i) throws MyException, MyOtherException {
        switch (i) {
            case 0: // processing resumes at point 1 above
                throw new MyException("input too low");
            case 1: // processing resumes at point 1 above
                throw new MySubException("input still too low");
            case 99:// processing resumes at point 2 above
                throw new MyOtherException("input too high");
            default:
                return i*i;
        }
    }
}
```

```
// This example is from the book "Java in a Nutshell, Second Edition".
// Written by David Flanagan.  Copyright (c) 1997 O'Reilly & Associates.
// You may distribute this source code for non-commercial purposes only.
// You may study, modify, and use this example for any purpose, as long as
// this notice is retained.  Note that this example is provided "as is",
// WITHOUT WARRANTY of any kind either expressed or implied.

import java.applet.*;   // Don't forget this import statement!
import java.awt.*;          // Or this one for the graphics!

public class FirstApplet extends Applet {
  // This method displays the applet.
  // The Graphics class is how you do all drawing in Java.
  public void paint(Graphics g) {
    g.drawString("Hello World", 25, 50);
  }
}
```

```
// This example is from the book "Java in a Nutshell, Second Edition".
// Written by David Flanagan.  Copyright (c) 1997 O'Reilly & Associates.
// You may distribute this source code for non-commercial purposes only.
// You may study, modify, and use this example for any purpose, as long as
// this notice is retained.  Note that this example is provided "as is",
// WITHOUT WARRANTY of any kind either expressed or implied.

import java.applet.*;
import java.awt.*;

public class SecondApplet extends Applet {
  static final String message = "Hello World";
  private Font font;

  // One-time initialization for the applet
  // Note: no constructor defined.
  public void init() {
    font = new Font("Helvetica", Font.BOLD, 48);
  }

  // Draw the applet whenever necessary.  Do some fancy graphics.
  public void paint(Graphics g) {
    // The pink oval
    g.setColor(Color.pink);
    g.fillOval(10, 10, 330, 100);

    // The red outline. Java doesn't support wide lines, so we
    // try to simulate a 4-pixel wide line by drawing four ovals.
    g.setColor(Color.red);
    g.drawOval(10,10, 330, 100);
    g.drawOval(9, 9, 332, 102);
    g.drawOval(8, 8, 334, 104);
    g.drawOval(7, 7, 336, 106);

    // The text
    g.setColor(Color.black);
    g.setFont(font);
    g.drawString(message, 40, 75);
  }
}
```

```java
// This example is from the book "Java in a Nutshell, Second Edition".
// Written by David Flanagan.  Copyright (c) 1997 O'Reilly & Associates.
// You may distribute this source code for non-commercial purposes only.
// You may study, modify, and use this example for any purpose, as long as
// this notice is retained.  Note that this example is provided "as is",
// WITHOUT WARRANTY of any kind either expressed or implied.

import java.applet.*;
import java.awt.*;

public class Scribble extends Applet {
  private int last_x = 0, last_y = 0;  // Fields to store a point in.

  // Called when the user clicks.
  public boolean mouseDown(Event e, int x, int y) {
    last_x = x; last_y = y;            // Remember the location of the click.
    return true;
  }

  // Called when the mouse moves with the button down
  public boolean mouseDrag(Event e, int x, int y)  {
    Graphics g = getGraphics();        // Get a Graphics to draw with.
    g.drawLine(last_x, last_y, x, y);  // Draw a line from last point to this.
    last_x = x; last_y = y;            // And update the saved location.
    return true;
  }
}
```

```java
// This example is from the book "Java in a Nutshell, Second Edition".
// Written by David Flanagan.  Copyright (c) 1997 O'Reilly & Associates.
// You may distribute this source code for non-commercial purposes only.
// You may study, modify, and use this example for any purpose, as long as
// this notice is retained.  Note that this example is provided "as is",
// WITHOUT WARRANTY of any kind either expressed or implied.

import java.applet.*;
import java.awt.*;

public class ColorScribble extends Scribble {
  // Read in two color parameters and set the colors.
  public void init() {
    super.init();
    Color foreground = getColorParameter("foreground");
    Color background = getColorParameter("background");
    if (foreground != null) this.setForeground(foreground);
    if (background != null) this.setBackground(background);
  }

  // Read the specified parameter.  Interpret it as a hexadecimal
  // number of the form RRGGBB and convert it to a color.
  protected Color getColorParameter(String name) {
    String value = this.getParameter(name);
    try { return new Color(Integer.parseInt(value, 16)); }
    catch (Exception e) { return null; }
  }

  // Return information suitable for display in an About dialog box.
  public String getAppletInfo() {
    return "ColorScribble v. 0.02.  Written by David Flanagan.";
  }

  // Return info about the supported parameters.  Web browsers and applet
  // viewers should display this information, and may also allow users to
  // set the parameter values.
  public String[][] getParameterInfo() { return info; }

  // Here's the information that getParameterInfo() returns.
  // It is an array of arrays of strings describing each parameter.
  // Format: parameter name, parameter type, parameter description
  private String[][] info = {
    {"foreground", "hexadecimal color value", "foreground color"},
    {"background", "hexadecimal color value", "background color"}
  };
}
```

```java
// This example is from the book "Java in a Nutshell, Second Edition".
// Written by David Flanagan.  Copyright (c) 1997 O'Reilly & Associates.
// You may distribute this source code for non-commercial purposes only.
// You may study, modify, and use this example for any purpose, as long as
// this notice is retained.  Note that this example is provided "as is",
// WITHOUT WARRANTY of any kind either expressed or implied.

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.util.*;

/**
 * A Java applet that simulates a client-side imagemap.
 * Plays a sound whenever the user clicks on one of the hyperlinks.
 */
public class Soundmap extends Applet {
  protected Image image;      // The image to display.
  protected Vector rects;     // A list of rectangles in it.
  protected AudioClip sound;  // A sound to play on user clicks in a rectangle.

  /** Initialize the applet */
  public void init() {
    // Look up the name of the image, relative to a base URL, and load it.
    // Note the use of three Applet methods in this one line.
    image = this.getImage(this.getDocumentBase(), this.getParameter("image"));

    // Lookup and parse a list of rectangular areas and the URLs they map to.
    // The convenience routine getRectangleParameter() is defined below.
    rects = new Vector();
    ImagemapRectangle r;
    for(int i = 0; (r = getRectangleParameter("rect" + i)) != null; i++)
      rects.addElement(r);

    // Look up a sound to play when the user clicks one of those areas.
    sound = this.getAudioClip(this.getDocumentBase(),
                              this.getParameter("sound"));

    // Specify an "event listener" object to respond to mouse button
    // presses and releases.  Note that this is the Java 1.1 event model.
    // Note that it also uses a Java 1.1 inner class, defined below.
    this.addMouseListener(new Listener());
  }

  /** Called when the applet is being unloaded from the system.
   * We use it here to "flush" the image we no longer need. This may
   * result in memory and other resources being freed more quickly. */
  public void destroy() { image.flush(); }

  /** To display the applet, we simply draw the image. */
  public void paint(Graphics g) { g.drawImage(image, 0, 0, this); }

  /** We override this method so that it doesn't clear the background
   * before calling paint().  No clear is necessary, since paint() overwrites
   * everything with an image.  Causes less flickering this way. */
  public void update(Graphics g) { paint(g); }

  /** Parse a comma-separated list of rectangle coordinates and a URL.
   * Used to read the imagemap rectangle definitions from applet parameters */
  protected ImagemapRectangle getRectangleParameter(String name) {
    int x, y, w, h;
    URL url;
    String value = this.getParameter(name);
    if (value == null) return null;

    try {
      StringTokenizer st = new StringTokenizer(value, ",");
      x = Integer.parseInt(st.nextToken());
      y = Integer.parseInt(st.nextToken());
      w = Integer.parseInt(st.nextToken());
      h = Integer.parseInt(st.nextToken());
      url = new URL(this.getDocumentBase(), st.nextToken());
    }
    catch (NoSuchElementException e) { return null; }
    catch (NumberFormatException e) { return null; }
    catch (MalformedURLException e) { return null; }

    return new ImagemapRectangle(x, y, w, h, url);
  }

  /**
   * An instance of this inner class is used to respond to mouse events
   */
  class Listener extends MouseAdapter {
    /** The rectangle that the mouse was pressed in. */
    private ImagemapRectangle lastrect;

    /** Called when a mouse button is pressed. */
    public void mousePressed(MouseEvent e) {
      // On button down, check if we're inside one of the specified rectangles.
      // If so, highlight the rectangle, display a message, and play a sound.
      // The utility routine findrect() is defined below.
      ImagemapRectangle r = findrect(e);
      if (r == null) return;
      Graphics g = Applet.this.getGraphics();
      g.setXORMode(Color.red);
      g.drawRect(r.x, r.y, r.width, r.height);  // highlight rectangle
      Applet.this.showStatus("To: " + r.url);   // display URL
      sound.play();                             // play the sound
      lastrect = r;   // Remember the rectangle so it can be un-highlighted.
    }

    /** Called when a mouse button is released. */
    public void mouseReleased(MouseEvent e) {
      // When the button is released, unhighlight the rectangle.  If the
      // mouse is still inside it, ask the browser to go to the URL.
      if (lastrect != null) {
        Graphics g = Applet.this.getGraphics();
        g.setXORMode(Color.red);
        g.drawRect(lastrect.x, lastrect.y, lastrect.width, lastrect.height);
        Applet.this.showStatus("");   // Clear the message.
        ImagemapRectangle r = findrect(e);
        if ((r != null) && (r == lastrect))  // If still in the same rectangle
          Applet.this.getAppletContext().showDocument(r.url); // Go to the URL
        lastrect = null;
      }
    }

    /** Find the rectangle we're inside. */
    protected ImagemapRectangle findrect(MouseEvent e) {
      int i, x = e.getX(), y = e.getY();
      for(i = 0; i < rects.size(); i++)  {
        ImagemapRectangle r = (ImagemapRectangle) rects.elementAt(i);
        if (r.contains(x, y)) return r;
      }
      return null;
    }
  }

  /**
   * A helper class.  Just like java.awt.Rectangle, but with a URL field.
   * Note the use of a nested toplevel class for neatness.
   */
  static class ImagemapRectangle extends Rectangle {
    URL url;
    public ImagemapRectangle(int x, int y, int w, int h, URL url) {
      super(x, y, w, h);
      this.url = url;
    }
  }
}
```

```java
// This example is from the book "Java in a Nutshell, Second Edition".
// Written by David Flanagan.  Copyright (c) 1997 O'Reilly & Associates.
// You may distribute this source code for non-commercial purposes only.
// You may study, modify, and use this example for any purpose, as long as
// this notice is retained.  Note that this example is provided "as is",
// WITHOUT WARRANTY of any kind either expressed or implied.

import java.applet.*;
import java.awt.*;

/** A simple applet using the Java 1.0 event handling model */
public class Scribble1 extends Applet {
  private int lastx, lasty;    // Remember last mouse coordinates.
  Button clear_button;         // The Clear button.
  Graphics g;                  // A Graphics object for drawing.

  /** Initialize the button and the Graphics object */
  public void init() {
    clear_button = new Button("Clear");
    this.add(clear_button);
    g = this.getGraphics();
  }
  /** Respond to mouse clicks */
  public boolean mouseDown(Event e, int x, int y) {
    lastx = x; lasty = y;
    return true;
  }
  /** Respond to mouse drags */
  public boolean mouseDrag(Event e, int x, int y) {
    g.setColor(Color.black);
    g.drawLine(lastx, lasty, x, y);
    lastx = x; lasty = y;
    return true;
  }
  /** Respond to key presses */
  public boolean keyDown(Event e, int key) {
    if ((e.id == Event.KEY_PRESS) && (key == 'c')) {
      clear();
      return true;
    }
    else return false;
  }
  /** Respond to Button clicks */
  public boolean action(Event e, Object arg) {
    if (e.target == clear_button) {
      clear();
      return true;
    }
    else return false;
  }
  /** convenience method to erase the scribble */
  public void clear() {
    g.setColor(this.getBackground());
    g.fillRect(0, 0, bounds().width, bounds().height);
  }
}
```

```
// This example is from the book "Java in a Nutshell, Second Edition".
// Written by David Flanagan.  Copyright (c) 1997 O'Reilly & Associates.
// You may distribute this source code for non-commercial purposes only.
// You may study, modify, and use this example for any purpose, as long as
// this notice is retained.  Note that this example is provided "as is",
// WITHOUT WARRANTY of any kind either expressed or implied.

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Scribble2 extends Applet
                      implements MouseListener,  MouseMotionListener {
  private int last_x, last_y;

  public void init() {
    // Tell this applet what MouseListener and MouseMotionListener
    // objects to notify when mouse and mouse motion events occur.
    // Since we implement the interfaces ourself, our own methods are called.
    this.addMouseListener(this);
    this.addMouseMotionListener(this);
  }

  // A method from the MouseListener interface.  Invoked when the
  // user presses a mouse button.
  public void mousePressed(MouseEvent e) {
    last_x = e.getX();
    last_y = e.getY();
  }

  // A method from the MouseMotionListener interface.  Invoked when the
  // user drags the mouse with a button pressed.
  public void mouseDragged(MouseEvent e) {
    Graphics g = this.getGraphics();
    int x = e.getX(), y = e.getY();
    g.drawLine(last_x, last_y, x, y);
    last_x = x; last_y = y;
  }

  // The other, unused methods of the MouseListener interface.
  public void mouseReleased(MouseEvent e) {;}
  public void mouseClicked(MouseEvent e) {;}
  public void mouseEntered(MouseEvent e) {;}
  public void mouseExited(MouseEvent e) {;}

  // The other method of the MouseMotionListener interface.
  public void mouseMoved(MouseEvent e) {;}
}
```

```java
// This example is from the book "Java in a Nutshell, Second Edition".
// Written by David Flanagan.  Copyright (c) 1997 O'Reilly & Associates.
// You may distribute this source code for non-commercial purposes only.
// You may study, modify, and use this example for any purpose, as long as
// this notice is retained.  Note that this example is provided "as is",
// WITHOUT WARRANTY of any kind either expressed or implied.

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Scribble3 extends Applet {
  int last_x, last_y;

  public void init() {
    // Define, instantiate and register a MouseListener object.
    this.addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent e) {
        last_x = e.getX();
        last_y = e.getY();
      }
    });

    // Define, instantiate and register a MouseMotionListener object.
    this.addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent e) {
        Graphics g = getGraphics();
        int x = e.getX(), y = e.getY();
        g.setColor(Color.black);
        g.drawLine(last_x, last_y, x, y);
        last_x = x; last_y = y;
      }
    });

    // Create a clear button
    Button b = new Button("Clear");
    // Define, instantiate, and register a listener to handle button presses
    b.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {  // clear the scribble
        Graphics g = getGraphics();
        g.setColor(getBackground());
        g.fillRect(0, 0, getSize().width, getSize().height);
      }
    });
    // And add the button to the applet
    this.add(b);
  }
}
```

```java
// This example is from the book "Java in a Nutshell, Second Edition".
// Written by David Flanagan.  Copyright (c) 1997 O'Reilly & Associates.
// You may distribute this source code for non-commercial purposes only.
// You may study, modify, and use this example for any purpose, as long as
// this notice is retained.  Note that this example is provided "as is",
// WITHOUT WARRANTY of any kind either expressed or implied.

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Scribble4 extends Applet {
  private int lastx, lasty;

  /** Tell the system we're interested in mouse events, mouse motion events,
   *  and keyboard events.  This is required or events won't be sent.
   */
  public void init() {
    this.enableEvents(AWTEvent.MOUSE_EVENT_MASK |
                      AWTEvent.MOUSE_MOTION_EVENT_MASK |
                      AWTEvent.KEY_EVENT_MASK);
    this.requestFocus();  // Ask for keyboard focus so we get key events
  }

  /** Invoked when a mouse event of some type occurs */
  public void processMouseEvent(MouseEvent e) {
    if (e.getID() == MouseEvent.MOUSE_PRESSED) {  // check the event type
      lastx = e.getX(); lasty = e.getY();
    }
    else super.processMouseEvent(e); // pass unhandled events to superclass
  }

  /** Invoked when a mouse motion event occurs */
  public void processMouseMotionEvent(MouseEvent e) {
    if (e.getID() == MouseEvent.MOUSE_DRAGGED) {  // check type
      int x = e.getX(), y = e.getY();
      Graphics g = this.getGraphics();
      g.drawLine(lastx, lasty, x, y);
      lastx = x; lasty = y;
    }
    else super.processMouseMotionEvent(e);
  }

  /** Called on key events:  clear the screen when 'c' is typed */
  public void processKeyEvent(KeyEvent e) {
    if ((e.getID() == KeyEvent.KEY_TYPED) && (e.getKeyChar() == 'c')) {
      Graphics g = this.getGraphics();
      g.setColor(this.getBackground());
      g.fillRect(0, 0, this.getSize().width, this.getSize().height);
    }
    else super.processKeyEvent(e);  // pass unhandled events to our superclass
  }
}
```

```java
// This example is from the book "Java in a Nutshell, Second Edition".
// Written by David Flanagan.  Copyright (c) 1997 O'Reilly & Associates.
// You may distribute this source code for non-commercial purposes only.
// You may study, modify, and use this example for any purpose, as long as
// this notice is retained.  Note that this example is provided "as is",
// WITHOUT WARRANTY of any kind either expressed or implied.

import java.awt.*;               // ScrollPane, PopupMenu, MenuShortcut, etc.
import java.awt.datatransfer.*;  // Clipboard, Transferable, DataFlavor, etc.
import java.awt.event.*;         // New event model.
import java.io.*;                // Object serialization streams.
import java.util.zip.*;          // Data compression/decompression streams.
import java.util.Vector;         // To store the scribble in.
import java.util.Properties;     // To store printing preferences in.

/**
 * This class places a Scribble component in a ScrollPane container,
 * puts the ScrollPane in a window, and adds a simple pulldown menu system.
 * The menu uses menu shortcuts.  Events are handled with anonymous classes.
 */
public class ScribbleFrame extends Frame {
  /** A very simple main() method for our program. */
  public static void main(String[] args) { new ScribbleFrame(); }

  /** Remember # of open windows so we can quit when last one is closed */
  protected static int num_windows = 0;

  /** Create a Frame, Menu, and ScrollPane for the scribble component */
  public ScribbleFrame() {
    super("ScribbleFrame");                  // Create the window.
    num_windows++;                           // Count it.

    ScrollPane pane = new ScrollPane();      // Create a ScrollPane.
    pane.setSize(300, 300);                  // Specify its size.
    this.add(pane, "Center");                // Add it to the frame.
    Scribble scribble;
    scribble = new Scribble(this, 500, 500); // Create a bigger scribble area.
    pane.add(scribble);                      // Add it to the ScrollPane.

    MenuBar menubar = new MenuBar();         // Create a menubar.
    this.setMenuBar(menubar);                // Add it to the frame.
    Menu file = new Menu("File");            // Create a File menu.
    menubar.add(file);                       // Add to menubar.

    // Create three menu items, with menu shortcuts, and add to the menu.
    MenuItem n, c, q;
    file.add(n = new MenuItem("New Window", new MenuShortcut(KeyEvent.VK_N)));
    file.add(c = new MenuItem("Close Window",new MenuShortcut(KeyEvent.VK_W)));
    file.addSeparator();                     // Put a separator in the menu
    file.add(q = new MenuItem("Quit", new MenuShortcut(KeyEvent.VK_Q)));

    // Create and register action listener objects for the three menu items.
    n.addActionListener(new ActionListener() {     // Open a new window
      public void actionPerformed(ActionEvent e) { new ScribbleFrame(); }
    });
    c.addActionListener(new ActionListener() {     // Close this window.
      public void actionPerformed(ActionEvent e) { close(); }
    });
    q.addActionListener(new ActionListener() {     // Quit the program.
      public void actionPerformed(ActionEvent e) { System.exit(0); }
    });
    // Another event listener, this one to handle window close requests.
    this.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) { close(); }
    });

    // Set the window size and pop it up.
    this.pack();
    this.show();
  }

  /** Close a window.  If this is the last open window, just quit. */
  void close() {
    if (--num_windows == 0) System.exit(0);
    else this.dispose();
  }
}

/**
 * This class is a custom component that supports scribbling.  It also has
 * a popup menu that allows the scribble color to be set and provides access
 * to printing, cut-and-paste, and file loading and saving facilities.
 * Note that it extends Component rather than Canvas, making it "lightweight."
 */
class Scribble extends Component implements ActionListener {
  protected short last_x, last_y;                // Coordinates of last click.
  protected Vector lines = new Vector(256,256);  // Store the scribbles.
  protected Color current_color = Color.black;   // Current drawing color.
  protected int width, height;                   // The preferred size.
  protected PopupMenu popup;                     // The popup menu.
  protected Frame frame;                         // The frame we are within.

  /** This constructor requires a Frame and a desired size */
  public Scribble(Frame frame, int width, int height) {
    this.frame = frame;
    this.width = width;
    this.height = height;

    // We handle scribbling with low-level events, so we must specify
    // which events we are interested in.
    this.enableEvents(AWTEvent.MOUSE_EVENT_MASK);
    this.enableEvents(AWTEvent.MOUSE_MOTION_EVENT_MASK);

    // Create the popup menu using a loop.  Note the separation of menu
    // "action command" string from menu label.  Good for internationalization.
    String[] labels = new String[] {
      "Clear", "Print", "Save", "Load", "Cut", "Copy", "Paste" };
    String[] commands = new String[] {
      "clear", "print", "save", "load", "cut", "copy", "paste" };
    popup = new PopupMenu();                   // Create the menu
    for(int i = 0; i < labels.length; i++) {
      MenuItem mi = new MenuItem(labels[i]);   // Create a menu item.
      mi.setActionCommand(commands[i]);        // Set its action command.
      mi.addActionListener(this);              // And its action listener.
      popup.add(mi);                           // Add item to the popup menu.
    }
    Menu colors = new Menu("Color");           // Create a submenu.
    popup.add(colors);                         // And add it to the popup.
    String[] colornames = new String[] { "Black", "Red", "Green", "Blue"};
    for(int i = 0; i < colornames.length; i++) {
      MenuItem mi = new MenuItem(colornames[i]);  // Create the submenu items
      mi.setActionCommand(colornames[i]);         // in the same way.
      mi.addActionListener(this);
      colors.add(mi);
    }
    // Finally, register the popup menu with the component it appears over
    this.add(popup);
  }

  /** Specifies big the component would like to be.  It always returns the
   *  preferred size passed to the Scribble() constructor */
  public Dimension getPreferredSize() { return new Dimension(width, height); }

  /** This is the ActionListener method invoked by the popup menu items */
  public void actionPerformed(ActionEvent event) {
    // Get the "action command" of the event, and dispatch based on that.
    // This method calls a lot of the interesting methods in this class.
    String command = event.getActionCommand();
    if (command.equals("clear")) clear();
    else if (command.equals("print")) print();
    else if (command.equals("save")) save();
    else if (command.equals("load")) load();
    else if (command.equals("cut")) cut();
    else if (command.equals("copy")) copy();
    else if (command.equals("paste")) paste();
    else if (command.equals("Black")) current_color = Color.black;
    else if (command.equals("Red")) current_color = Color.red;
    else if (command.equals("Green")) current_color = Color.green;
    else if (command.equals("Blue")) current_color = Color.blue;
  }

  /** Draw all the saved lines of the scribble, in the appropriate colors */
  public void paint(Graphics g) {
    for(int i = 0; i < lines.size(); i++) {
      Line l = (Line)lines.elementAt(i);
      g.setColor(l.color);
      g.drawLine(l.x1, l.y1, l.x2, l.y2);
    }
  }

  /**
   * This is the low-level event-handling method called on mouse events
   * that do not involve mouse motion.  Note the use of isPopupTrigger()
   * to check for the platform-dependent popup menu posting event, and of
   * the show() method to make the popup visible.  If the menu is not posted,
   * then this method saves the coordinates of a mouse click or invokes
   * the superclass method.
   */
  public void processMouseEvent(MouseEvent e) {
    if (e.isPopupTrigger())                               // If popup trigger,
      popup.show(this, e.getX(), e.getY());               // pop up the menu.
    else if (e.getID() == MouseEvent.MOUSE_PRESSED) {
      last_x = (short)e.getX(); last_y = (short)e.getY(); // Save position.
    }
    else super.processMouseEvent(e);  // Pass other event types on.
  }

  /**
   * This method is called for mouse motion events.  It adds a line to the
   * scribble, on screen, and in the saved representation
   */
  public void processMouseMotionEvent(MouseEvent e) {
    if (e.getID() == MouseEvent.MOUSE_DRAGGED) {
      Graphics g = getGraphics();                     // Object to draw with.
      g.setColor(current_color);                      // Set the current color.
      g.drawLine(last_x, last_y, e.getX(), e.getY()); // Draw this line
      lines.addElement(new Line(last_x, last_y,       // and save it, too.
                                (short) e.getX(), (short)e.getY(),
                                current_color));
      last_x = (short) e.getX();  // Remember current mouse coordinates.
      last_y = (short) e.getY();
    }
    else super.processMouseMotionEvent(e);  // Important!
  }

  /** Clear the scribble.  Invoked by popup menu */
  void clear() {
    lines.removeAllElements();   // Throw out the saved scribble
    repaint();                   // and redraw everything.
  }

  /** Print out the scribble.  Invoked by popup menu. */
  void print() {
    // Obtain a PrintJob object.  This posts a Print dialog.
    // printprefs (created below) stores user printing preferences.
    Toolkit toolkit = this.getToolkit();
    PrintJob job = toolkit.getPrintJob(frame, "Scribble", printprefs);

    // If the user clicked Cancel in the print dialog, then do nothing.
    if (job == null) return;

    // Get a Graphics object for the first page of output.
    Graphics page = job.getGraphics();

    // Check the size of the scribble component and of the page.
    Dimension size = this.getSize();
    Dimension pagesize = job.getPageDimension();

    // Center the output on the page.  Otherwise it would be
    // be scrunched up in the upper-left corner of the page.
    page.translate((pagesize.width - size.width)/2,
                   (pagesize.height - size.height)/2);

    // Draw a border around the output area, so it looks neat.
    page.drawRect(-1, -1, size.width+1, size.height+1);

    // Set a clipping region so our scribbles don't go outside the border.
    // On-screen this clipping happens automatically, but not on paper.
    page.setClip(0, 0, size.width, size.height);

    // Print this Scribble component.  By default this will just call paint().
    // This method is named print(), too, but that is just coincidence.
    this.print(page);

    // Finish up printing.
    page.dispose();   // End the page--send it to the printer.
    job.end();        // End the print job.
  }

  /** This Properties object stores the user print dialog settings. */
  private static Properties printprefs = new Properties();

  /**
   * The DataFlavor used for our particular type of cut-and-paste data.
   * This one will transfer data in the form of a serialized Vector object.
   * Note that in Java 1.1.1, this works intra-application, but not between
   * applications.  Java 1.1.1 inter-application data transfer is limited to
   * the pre-defined string and text data flavors.
   */
  public static final DataFlavor dataFlavor =
      new DataFlavor(Vector.class, "ScribbleVectorOfLines");

  /**
   * Copy the current scribble and store it in a SimpleSelection object
   * (defined below).  Then put that object on the clipboard for pasting.
   */
  public void copy() {
    // Get system clipboard
    Clipboard c = this.getToolkit().getSystemClipboard();
    // Copy and save the scribble in a Transferable object
    SimpleSelection s = new SimpleSelection(lines.clone(), dataFlavor);
    // Put that object on the clipboard
    c.setContents(s, s);
  }

  /** Cut is just like a copy, except we erase the scribble afterwards */
  public void cut() { copy(); clear();  }

  /**
   * Ask for the Transferable contents of the system clipboard, then ask that
   * object for the scribble data it represents.  If either step fails, beep!
   */
  public void paste() {
    Clipboard c = this.getToolkit().getSystemClipboard();  // Get clipboard.
    Transferable t = c.getContents(this);                  // Get its contents.
    if (t == null) {              // If there is nothing to paste, beep.
      this.getToolkit().beep();
      return;
    }
    try {
      // Ask for clipboard contents to be converted to our data flavor.
      // This will throw an exception if our flavor is not supported.
      Vector newlines = (Vector) t.getTransferData(dataFlavor);
      // Add all those pasted lines to our scribble.
      for(int i = 0; i < newlines.size(); i++)
        lines.addElement(newlines.elementAt(i));
      // And redraw the whole thing
      repaint();
    }
    catch (UnsupportedFlavorException e) {
      this.getToolkit().beep();   // If clipboard has some other type of data
    }
    catch (Exception e) {
      this.getToolkit().beep();   // Or if anything else goes wrong...
    }
  }

  /**
   * This nested class implements the Transferable and ClipboardOwner
   * interfaces used in data transfer.  It is a simple class that remembers a
   * selected object and makes it available in only one specified flavor.
   */
  static class SimpleSelection implements Transferable, ClipboardOwner {
    protected Object selection;    // The data to be transferred.
    protected DataFlavor flavor;   // The one data flavor supported.
    public SimpleSelection(Object selection, DataFlavor flavor) {
      this.selection = selection;  // Specify data.
      this.flavor = flavor;        // Specify flavor.
    }

    /** Return the list of supported flavors.  Just one in this case */
    public DataFlavor[] getTransferDataFlavors() {
      return new DataFlavor[] { flavor };
    }
    /** Check whether we support a specified flavor */
    public boolean isDataFlavorSupported(DataFlavor f) {
      return f.equals(flavor);
    }
    /** If the flavor is right, transfer the data (i.e. return it) */
    public Object getTransferData(DataFlavor f)
         throws UnsupportedFlavorException {
      if (f.equals(flavor)) return selection;
      else throw new UnsupportedFlavorException(f);
    }

    /** This is the ClipboardOwner method.  Called when the data is no
     *  longer on the clipboard.  In this case, we don't need to do much. */
    public void lostOwnership(Clipboard c, Transferable t) {
      selection = null;
    }
  }

  /**
   * Prompt the user for a filename, and save the scribble in that file.
   * Serialize the vector of lines with an ObjectOutputStream.
   * Compress the serialized objects with a GZIPOutputStream.
   * Write the compressed, serialized data to a file with a FileOutputStream.
   * Don't forget to flush and close the stream.
   */
  public void save() {
    // Create a file dialog to query the user for a filename.
    FileDialog f = new FileDialog(frame, "Save Scribble", FileDialog.SAVE);
    f.show();                        // Display the dialog and block.
    String filename = f.getFile();   // Get the user's response
    if (filename != null) {          // If user didn't click "Cancel".
      try {
        // Create the necessary output streams to save the scribble.
        FileOutputStream fos = new FileOutputStream(filename); // Save to file
        GZIPOutputStream gzos = new GZIPOutputStream(fos);     // Compressed
        ObjectOutputStream out = new ObjectOutputStream(gzos); // Save objects
        out.writeObject(lines);      // Write the entire Vector of scribbles
        out.flush();                 // Always flush the output.
        out.close();                 // And close the stream.
      }
      // Print out exceptions.  We should really display them in a dialog...
      catch (IOException e) { System.out.println(e); }
    }
  }

  /**
   * Prompt for a filename, and load a scribble from that file.
   * Read compressed, serialized data with a FileInputStream.
   * Uncompress that data with a GZIPInputStream.
   * Deserialize the vector of lines with a ObjectInputStream.
   * Replace current data with new data, and redraw everything.
   */
  public void load() {
    // Create a file dialog to query the user for a filename.
    FileDialog f = new FileDialog(frame, "Load Scribble", FileDialog.LOAD);
```

```
    f.show();                         // Display the dialog and block.
    String filename = f.getFile();    // Get the user's response
    if (filename != null) {           // If user didn't click "Cancel".
      try {
        // Create necessary input streams
        FileInputStream fis = new FileInputStream(filename); // Read from file
        GZIPInputStream gzis = new GZIPInputStream(fis);     // Uncompress
        ObjectInputStream in = new ObjectInputStream(gzis);  // Read objects
        // Read in an object.  It should be a vector of scribbles
        Vector newlines = (Vector)in.readObject();
        in.close();                    // Close the stream.
        lines = newlines;              // Set the Vector of lines.
        repaint();                     // And redisplay the scribble.
      }
      // Print out exceptions.  We should really display them in a dialog...
      catch (Exception e) { System.out.println(e); }
    }
  }

  /** A class to store the coordinates and color of one scribbled line.
   *  The complete scribble is stored as a Vector of these objects */
  static class Line implements Serializable {
    public short x1, y1, x2, y2;
    public Color color;
    public Line(short x1, short y1, short x2, short y2, Color c) {
      this.x1 = x1; this.y1 = y1; this.x2 = x2; this.y2 = y2; this.color = c;
    }
  }
}
```

```java
// This example is from the book "Java in a Nutshell, Second Edition".
// Written by David Flanagan.  Copyright (c) 1997 O'Reilly & Associates.
// You may distribute this source code for non-commercial purposes only.
// You may study, modify, and use this example for any purpose, as long as
// this notice is retained.  Note that this example is provided "as is",
// WITHOUT WARRANTY of any kind either expressed or implied.

import java.io.*;

/** A simple class that implements a growable array or ints, and knows
 *  how to serialize itself as efficiently as a non-growable array. */
public class IntList implements Serializable
{
  private int[] nums = new int[8]; // An array to store the numbers.
  private transient int size = 0;  // Index of next unused element of nums[].

  /** Return an element of the array */
  public int elementAt(int index) throws ArrayIndexOutOfBoundsException {
    if (index >= size) throw new ArrayIndexOutOfBoundsException(index);
    else return nums[index];
  }

  /** Add an int to the array, growing the array if necessary */
  public void add(int x) {
    if (nums.length == size) resize(nums.length*2); // Grow array, if needed.
    nums[size++] = x;                               // Store the int in it.
  }

  /** An internal method to change the allocated size of the array */
  protected void resize(int newsize) {
    int[] oldnums = nums;
    nums = new int[newsize];                     // Create a new array.
    System.arraycopy(oldnums, 0, nums, 0, size); // Copy array elements.
  }

  /** Get rid of unused array elements before serializing the array */
  private void writeObject(ObjectOutputStream out) throws IOException {
    if (nums.length > size) resize(size);  // Compact the array.
    out.defaultWriteObject();              // Then write it out normally.
  }

  /** Compute the transient size field after deserializing the array */
  private void readObject(ObjectInputStream in)
          throws IOException, ClassNotFoundException {
    in.defaultReadObject();                // Read the array normally.
    size = nums.length;                    // Restore the transient field.
  }
}
```

```
// This example is from the book "Java in a Nutshell, Second Edition".
// Written by David Flanagan.  Copyright (c) 1997 O'Reilly & Associates.
// You may distribute this source code for non-commercial purposes only.
// You may study, modify, and use this example for any purpose, as long as
// this notice is retained.  Note that this example is provided "as is",
// WITHOUT WARRANTY of any kind either expressed or implied.

package oreilly.beans.yesno;
import java.awt.*;
import java.util.*;

/**
 * A custom component that displays multiple lines of text with specified
 * margins and alignment.  In Java 1.1, we could extend Component instead
 * of Canvas, making this a more efficient "Lightweight component"
 */
public class MultiLineLabel extends Canvas {
  // User-specified attributes
  protected String label;             // The label, not broken into lines
  protected int margin_width;         // Left and right margins
  protected int margin_height;        // Top and bottom margins
  protected int alignment;            // The alignment of the text.
  public static final int LEFT = 0, CENTER = 1, RIGHT = 2; // alignment values
  // Computed state values
  protected int num_lines;            // The number of lines
  protected String[] lines;           // The label, broken into lines
  protected int[] line_widths;        // How wide each line is
  protected int max_width;            // The width of the widest line
  protected int line_height;          // Total height of the font
  protected int line_ascent;          // Font height above baseline
  protected boolean measured = false; // Have the lines been measured?

  // Here are five versions of the constructor.
  public MultiLineLabel(String label, int margin_width,
                        int margin_height, int alignment) {
    this.label = label;                 // Remember all the properties.
    this.margin_width = margin_width;
    this.margin_height = margin_height;
    this.alignment = alignment;
    newLabel();                         // Break the label up into lines.
  }
  public MultiLineLabel(String label, int margin_width, int margin_height) {
    this(label, margin_width, margin_height, LEFT);
  }
  public MultiLineLabel(String label, int alignment) {
    this(label, 10, 10, alignment);
  }
  public MultiLineLabel(String label) { this(label, 10, 10, LEFT); }
  public MultiLineLabel() { this(""); }

  // Methods to set and query the various attributes of the component.
  // Note that some query methods are inherited from the superclass.
  public void setLabel(String label) {
    this.label = label;
    newLabel();               // Break the label into lines.
    measured = false;         // Note that we need to measure lines.
    repaint();                // Request a redraw.
  }
  public void setFont(Font f) {
    super.setFont(f);         // Tell our superclass about the new font.
    measured = false;         // Note that we need to remeasure lines.
    repaint();                // Request a redraw.
  }
  public void setForeground(Color c) {
    super.setForeground(c);   // Tell our superclass about the new color.
    repaint();                // Request a redraw (size is unchanged).
  }
  public void setAlignment(int a) { alignment = a; repaint(); }
  public void setMarginWidth(int mw) { margin_width = mw; repaint(); }
  public void setMarginHeight(int mh) { margin_height = mh; repaint(); }
  public String getLabel() { return label; }
  public int getAlignment() { return alignment; }
  public int getMarginWidth() { return margin_width; }
  public int getMarginHeight() { return margin_height; }

  /**
   * This method is called by a layout manager when it wants to
   * know how big we'd like to be.  In Java 1.1, getPreferredSize() is
   * the preferred version of this method.  We use this deprecated version
   * so that this component can interoperate with 1.0 components.
   */
  public Dimension preferredSize() {
    if (!measured) measure();
    return new Dimension(max_width + 2*margin_width,
                         num_lines * line_height + 2*margin_height);
  }

  /**
   * This method is called when the layout manager wants to know
   * the bare minimum amount of space we need to get by.
   * For Java 1.1, we'd use getMinimumSize().
   */
  public Dimension minimumSize() { return preferredSize(); }

  /**
   * This method draws the label (same method that applets use).
   * Note that it handles the margins and the alignment, but that
   * it doesn't have to worry about the color or font--the superclass
   * takes care of setting those in the Graphics object we're passed.
   */
  public void paint(Graphics g) {
    int x, y;
    Dimension size = this.size();  // use getSize() in Java 1.1
    if (!measured) measure();
    y = line_ascent + (size.height - num_lines * line_height)/2;
    for(int i = 0; i < num_lines; i++, y += line_height) {
      switch(alignment) {
      default:
      case LEFT:    x = margin_width; break;
      case CENTER:  x = (size.width - line_widths[i])/2; break;
      case RIGHT:   x = size.width - margin_width - line_widths[i]; break;
      }
      g.drawString(lines[i], x, y);
    }
  }

  /** This internal method breaks a specified label up into an array of lines.
   *  It uses the StringTokenizer utility class. */
  protected synchronized void newLabel() {
    StringTokenizer t = new StringTokenizer(label, "\n");
    num_lines = t.countTokens();
    lines = new String[num_lines];
```

```
    line_widths = new int[num_lines];
    for(int i = 0; i < num_lines; i++) lines[i] = t.nextToken();
  }

  /** This internal method figures out how the font is, and how wide each
   *  line of the label is, and how wide the widest line is. */
  protected synchronized void measure() {
    FontMetrics fm = this.getToolkit().getFontMetrics(this.getFont());
    line_height = fm.getHeight();
    line_ascent = fm.getAscent();
    max_width = 0;
    for(int i = 0; i < num_lines; i++) {
      line_widths[i] = fm.stringWidth(lines[i]);
      if (line_widths[i] > max_width) max_width = line_widths[i];
    }
    measured = true;
  }
}
```

```java
// This example is from the book "Java in a Nutshell, Second Edition".
// Written by David Flanagan.  Copyright (c) 1997 O'Reilly & Associates.
// You may distribute this source code for non-commercial purposes only.
// You may study, modify, and use this example for any purpose, as long as
// this notice is retained.  Note that this example is provided "as is",
// WITHOUT WARRANTY of any kind either expressed or implied.

package oreilly.beans.yesno;      // Put this bean in its own private package.
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class YesNoDialog {
  // Properties of the bean.
  protected String message, title;
  protected String yesLabel, noLabel, cancelLabel;
  protected int alignment;
  protected Font font = new Font("Serif", Font.PLAIN, 12);
  protected Color background = SystemColor.control;
  protected Color foreground = SystemColor.controlText;

  // Constants for the alignment property.
  public static final int LEFT = MultiLineLabel.LEFT;
  public static final int RIGHT = MultiLineLabel.RIGHT;
  public static final int CENTER = MultiLineLabel.CENTER;

  // Methods to query all of the bean properties.
  public String getMessage() { return message; }
  public String getTitle() { return title; }
  public String getYesLabel() { return yesLabel; }
  public String getNoLabel() { return noLabel; }
  public String getCancelLabel() { return cancelLabel; }
  public int getAlignment() { return alignment; }
  public Font getFont() { return font; }
  public Color getBackground() { return background; }
  public Color getForeground() { return foreground; }

  // Methods to set all of the bean properties.
  public void setMessage(String m) { message = m; }
  public void setTitle(String t) { title=t; }
  public void setYesLabel(String l) { yesLabel = l; }
  public void setNoLabel(String l) { noLabel = l; }
  public void setCancelLabel(String l) { cancelLabel = l; }
  public void setAlignment(int a) { alignment = a; }
  public void setFont(Font f) { font = f; }
  public void setBackground(Color bg) { background = bg; }
  public void setForeground(Color fg) { foreground = fg; }

  /** This field holds a list of registered ActionListeners.
   *  Vector is internally synchronized to prevent race conditions */
  protected Vector listeners = new Vector();

  /** Register an action listener to be notified when a button is pressed */
  public void addAnswerListener(AnswerListener l) {
    listeners.addElement(l);
  }

  /** Remove an Answer listener from our list of interested listeners */
  public void removeAnswerListener(AnswerListener l) {
    listeners.removeElement(l);
  }
```

```
/** Send an event to all registered listeners */
public void fireEvent(AnswerEvent e) {
  // Make a copy of the list and fire the events using that copy.
  // This means that listeners can be added or removed from the original
  // list in response to this event.  We ought to be able to just use an
  // enumeration for the vector, but that doesn't copy the list internally.
  Vector list = (Vector) listeners.clone();
  for(int i = 0; i < list.size(); i++) {
    AnswerListener listener = (AnswerListener)list.elementAt(i);
    switch(e.getID()) {
    case AnswerEvent.YES: listener.yes(e); break;
    case AnswerEvent.NO:  listener.no(e); break;
    case AnswerEvent.CANCEL: listener.cancel(e); break;
    }
  }
}

/** The no-argument bean constructor, with default property values */
public YesNoDialog() {
  this("Question", "Your\nMessage\nHere", "Yes", "No", "Cancel", LEFT);
}

/** A constructor for programmers using this class "by hand" */
public YesNoDialog(String title, String message,
                   String yesLabel, String noLabel, String cancelLabel,
                   int alignment) {
  this.title = title;
  this.message = message;
  this.yesLabel = yesLabel;
  this.noLabel = noLabel;
  this.cancelLabel = cancelLabel;
  this.alignment = alignment;
}

/** This method makes the bean display the dialog box */
public void display() {
  // Create a frame with the specified title.  It would be nice to
  // use a Dialog, but that needs to be passed a Frame argument, and
  // the BDK beanbox tool only seems to work with no-argument methods.
  final Frame frame = new Frame(title);

  // Specify a LayoutManager for it.
  frame.setLayout(new BorderLayout(15, 15));

  // Specify font and colors, if any are specified.
  if (font != null) frame.setFont(font);
  if (background != null) frame.setBackground(background);
  if (foreground != null) frame.setForeground(foreground);

  // Put the message label in the middle of the window.
  frame.add("Center", new MultiLineLabel(message, 20, 20, alignment));

  // Create an action listener for use by the buttons of the dialog.
  // When a button is pressed, this listener first closes the dialog box.
  // Then, it creates an AnswerEvent object that corresponds to the
  // button that was pressed, and send that new event to all registered
  // listeners, using the fireEvent() method defined above.
  ActionListener listener = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      frame.dispose();     // pop down window
      if (listeners != null) {      // notify any registered listeners
        String cmd = e.getActionCommand();
        int type;
        if (cmd.equals("yes")) type = AnswerEvent.YES;
        else if (cmd.equals("no")) type = AnswerEvent.NO;

        else type = AnswerEvent.CANCEL;
        fireEvent(new AnswerEvent(YesNoDialog.this, type));
      }
    }
  };

  // Create a panel for the dialog buttons and put it at the bottom
  // of the dialog.  Specify a FlowLayout layout manager for it.
  Panel buttonbox = new Panel();
  buttonbox.setLayout(new FlowLayout(FlowLayout.CENTER, 25, 15));
  frame.add("South", buttonbox);

  // Create each specified button, specifying the action listener
  // and action command for each, and adding them to the buttonbox
  if ((yesLabel != null) && (yesLabel.length() > 0)) {
    Button yes = new Button(yesLabel);        // Create button.
    yes.setActionCommand("yes");              // Set action command.
    yes.addActionListener(listener);          // Set listener.
    buttonbox.add(yes);                       // Add button to the panel.
  }
  if ((noLabel != null) && (noLabel.length() > 0)) {
    Button no = new Button(noLabel);
    no.setActionCommand("no");
    no.addActionListener(listener);
    buttonbox.add(no);
  }
  if ((cancelLabel != null) && (cancelLabel.length() > 0)) {
    Button cancel = new Button(cancelLabel);
    cancel.setActionCommand("cancel");
    cancel.addActionListener(listener);
    buttonbox.add(cancel);
  }

  // Finally, set the dialog to its preferred size and display it.
  frame.pack();
  frame.show();
}

/**
 * A main method that demonstrates how to use this class, and allows testing
 */
public static void main(String[] args) {
  // Create an instance of InfoDialog, with title and message specified:
  YesNoDialog d =
    new YesNoDialog("YesNoDialog Test",
                    "There are unsaved files.\n" +
                    "Do you want to save them before quitting?",
                    "Yes, save and quit",
                    "No, quit without saving",
                    "Cancel; don't quit",
                    YesNoDialog.CENTER);

  // Register an action listener for the dialog.  This one just prints
  // the results out to the console.
  d.addAnswerListener(new AnswerListener() {
    public void yes(AnswerEvent e) { System.out.println("Yes"); }
    public void no(AnswerEvent e) { System.out.println("No"); }
    public void cancel(AnswerEvent e) { System.out.println("Cancel"); }
    });

    // Now pop the dialog up.  It will pop itself down automatically.
    d.display();
  }
}
```

```
// This example is from the book "Java in a Nutshell, Second Edition".
// Written by David Flanagan.  Copyright (c) 1997 O'Reilly & Associates.
// You may distribute this source code for non-commercial purposes only.
// You may study, modify, and use this example for any purpose, as long as
// this notice is retained.  Note that this example is provided "as is",
// WITHOUT WARRANTY of any kind either expressed or implied.

package oreilly.beans.yesno;

public class AnswerEvent extends java.util.EventObject {
  protected int id;
  public static final int YES = 0, NO = 1, CANCEL = 2;
  public AnswerEvent(Object source, int id) {
    super(source);
    this.id = id;
  }
  public int getID() { return id; }
}
```

```
// This example is from the book "Java in a Nutshell, Second Edition".
// Written by David Flanagan.  Copyright (c) 1997 O'Reilly & Associates.
// You may distribute this source code for non-commercial purposes only.
// You may study, modify, and use this example for any purpose, as long as
// this notice is retained.  Note that this example is provided "as is",
// WITHOUT WARRANTY of any kind either expressed or implied.

package oreilly.beans.yesno;

public interface AnswerListener extends java.util.EventListener {
  public void yes(AnswerEvent e);
  public void no(AnswerEvent e);
  public void cancel(AnswerEvent e);
}
```

```
// This example is from the book "Java in a Nutshell, Second Edition".
// Written by David Flanagan.  Copyright (c) 1997 O'Reilly & Associates.
// You may distribute this source code for non-commercial purposes only.
// You may study, modify, and use this example for any purpose, as long as
// this notice is retained.  Note that this example is provided "as is",
// WITHOUT WARRANTY of any kind either expressed or implied.

package oreilly.beans.yesno;
import java.beans.*;
import java.lang.reflect.*;
import java.awt.*;

/** The BeanInfo class for the YesNoDialog bean */
public class YesNoDialogBeanInfo extends SimpleBeanInfo {

  /** Return an icon for the bean.  We should really check the kind argument
   *  to see what size icon the beanbox wants, but since we only have one
   *  icon to offer, we just return it and let the beanbox deal with it */
  public Image getIcon(int kind) {
    return loadImage("YesNoDialogIcon.gif");
  }

  /** Return a descriptor for the bean itself.  It specifies a customizer
   *  for the bean class.  We could also add a description string here */
  public BeanDescriptor getBeanDescriptor() {
    return new BeanDescriptor(YesNoDialog.class, YesNoDialogCustomizer.class);
  }

  /** This is a convenience routine for creating PropertyDescriptor objects */
  public static PropertyDescriptor property(String name, String description)
       throws IntrospectionException
  {
    PropertyDescriptor p = new PropertyDescriptor(name, YesNoDialog.class);
    p.setShortDescription(description);
    return p;
  }

  /** This method returns an array of PropertyDescriptor objects that specify
   *  additional information about the properties supported by the bean.
   *  By explicitly specifying property descriptors, we are able to provide
   *  simple help strings for each property; these would not be available to
   *  the beanbox through simple introspection.  We are also able to register
   *  special property editors for two of the properties
   */
  public PropertyDescriptor[] getPropertyDescriptors() {
    try {
      PropertyDescriptor[] props = {
        property("title", "The string that appears in the dialog title bar"),
        property("message", "The string that appears in the dialog body"),
        property("yesLabel", "The label for the 'Yes' button, if any"),
        property("noLabel", "The label for the 'No' button, if any"),
        property("cancelLabel", "The label for the 'Cancel' button, if any"),
        property("alignment", "The alignment of the message text"),
        property("font", "The font to use for message and buttons"),
        property("background", "The background color for the dialog"),
        property("foreground", "The text color for message and buttons")
      };

      props[1].setPropertyEditorClass(YesNoDialogMessageEditor.class);
      props[5].setPropertyEditorClass(YesNoDialogAlignmentEditor.class);

      return props;
    }
    catch (IntrospectionException e) {return super.getPropertyDescriptors(); }
  }

  /** The message property is most often customized; make it the default */
  public int getDefaultPropertyIndex() { return 1; }

  /** This is a convenience method for creating MethodDescriptors.  Note that
   *  it assumes we are talking about methods with no arguments */
  public static MethodDescriptor method(String name, String description)
       throws NoSuchMethodException, SecurityException {
    Method m = YesNoDialog.class.getMethod(name, new Class[] {});
    MethodDescriptor md = new MethodDescriptor(m);
    md.setShortDescription(description);
    return md;
  }

  /** This method returns an array of method descriptors for the supported
   *  methods of a bean. This allows us to provide useful description strings,
   *  but it also allows us to filter out non-useful methods like wait()
   *  and notify() that the bean inherits and which might otherwise be
   *  displayed by the beanbox.
   */
  public MethodDescriptor[] getMethodDescriptors() {
    try {
      MethodDescriptor[] methods = {
        method("display", "Pop up the dialog; make it visible")
      };
      return methods;
    }
    catch (Exception e) {
      return super.getMethodDescriptors();
    }
  }
}
```

```
// This example is from the book "Java in a Nutshell, Second Edition".
// Written by David Flanagan.  Copyright (c) 1997 O'Reilly & Associates.
// You may distribute this source code for non-commercial purposes only.
// You may study, modify, and use this example for any purpose, as long as
// this notice is retained.  Note that this example is provided "as is",
// WITHOUT WARRANTY of any kind either expressed or implied.

package oreilly.beans.yesno;
import java.beans.*;
import java.awt.*;

public class YesNoDialogAlignmentEditor extends PropertyEditorSupport {
  // These two methods allow the property to be edited in a dropdown list.
  // Return the list of value names for the enumerated type.
  public String[] getTags() {
    return new String[] { "left", "center", "right" };
  }

  // Convert each of those value names into the actual value.
  public void setAsText(String s) {
    if (s.equals("left")) setValue(new Integer(YesNoDialog.LEFT));
    else if (s.equals("center")) setValue(new Integer(YesNoDialog.CENTER));
    else if (s.equals("right")) setValue(new Integer(YesNoDialog.RIGHT));
    else throw new IllegalArgumentException(s);
  }

  // This is an important method for code generation.
  public String getJavaInitializationString() {
    switch(((Number)getValue()).intValue()) {
    default:
    case YesNoDialog.LEFT:   return "oreilly.beans.yesno.YesNoDialog.LEFT";
    case YesNoDialog.CENTER: return "oreilly.beans.yesno.YesNoDialog.CENTER";
    case YesNoDialog.RIGHT:  return "oreilly.beans.yesno.YesNoDialog.RIGHT";
    }
  }
}
```

```java
// This example is from the book "Java in a Nutshell, Second Edition".
// Written by David Flanagan.  Copyright (c) 1997 O'Reilly & Associates.
// You may distribute this source code for non-commercial purposes only.
// You may study, modify, and use this example for any purpose, as long as
// this notice is retained.  Note that this example is provided "as is",
// WITHOUT WARRANTY of any kind either expressed or implied.

package oreilly.beans.yesno;
import java.beans.*;
import java.awt.*;
import java.awt.event.*;

/**
 * This class is a custom editor for the message property of the
 * YesNoDialog bean.  It is necessary because the default editor for
 * properties of type String does not allow multi-line strings to be entered.
 */
public class YesNoDialogMessageEditor implements PropertyEditor {
  protected String value;  // The value we will be editing.

  public void setValue(Object o) {  value = (String) o; }
  public Object getValue() { return value; }
  public void setAsText(String s) { value = s; }
  public String getAsText() { return value; }
  public String[] getTags() { return null; }  // not enumerated; no tags

  // Say that we allow custom editing.
  public boolean supportsCustomEditor() { return true; }

  // Return the custom editor.  This just creates and returns a TextArea
  // to edit the multi-line text.  But it also registers a listener on the
  // text area to update the value as the user types and to fire the
  // property change events that property editors are required to fire.
  public Component getCustomEditor() {
    final TextArea t = new TextArea(value);
    t.setSize(300, 150); // TextArea doesn't have a preferred size, so set one
    t.addTextListener(new TextListener() {
      public void textValueChanged(TextEvent e) {
        value = t.getText();
        listeners.firePropertyChange(null, null, null);
      }
    });
    return t;
  }

  // Visual display of the value, for use with the custom editor.
  // Just print some instructions and hope they fit in the in the box.
  // This could be more sophisticated.
  public boolean isPaintable() { return true; }
  public void paintValue(Graphics g, Rectangle r) {
    g.setClip(r);
    g.drawString("Click to edit...", r.x+5, r.y+15);
  }

  // Important method for code generators.  Note that it
  // ought to add any necessary escape sequences.
  public String getJavaInitializationString() { return "\"" + value + "\""; }

  // This code uses the PropertyChangeSupport class to maintain a list of
  // listeners interested in the edits we make to the value.
  protected PropertyChangeSupport listeners = new PropertyChangeSupport(this);
  public void addPropertyChangeListener(PropertyChangeListener l) {
    listeners.addPropertyChangeListener(l);
  }
  public void removePropertyChangeListener(PropertyChangeListener l) {
    listeners.removePropertyChangeListener(l);
  }
}
```

```
// This example is from the book "Java in a Nutshell, Second Edition".
// Written by David Flanagan.  Copyright (c) 1997 O'Reilly & Associates.
// You may distribute this source code for non-commercial purposes only.
// You may study, modify, and use this example for any purpose, as long as
// this notice is retained.  Note that this example is provided "as is",
// WITHOUT WARRANTY of any kind either expressed or implied.

package oreilly.beans.yesno;

import java.awt.*;
import java.awt.event.*;
import java.beans.*;

/**
 * This class is a customizer for the YesNoDialog bean.  It displays a
 * TextArea and three TextFields where the user can enter the dialog message
 * and the labels for each of the three buttons.  It does not allow the
 * dialog title or other resources to be set.
 */
public class YesNoDialogCustomizer extends Panel
                                   implements Customizer, TextListener
{
  protected YesNoDialog bean;                 // The bean being customized
  protected TextComponent message, fields[];  // Components used by customizer

  // Default constructor: YesNoDialogCustomizer() { super(); }

  // The bean box calls this method to tell us what object to customize.
  // This method will always be called before the customizer is displayed,
  // so it is safe to create the customizer GUI here.
  public void setObject(Object o) {
    bean = (YesNoDialog)o;   // save the object we're customizing

    // Put a label at the top of the panel.
    this.setLayout(new BorderLayout());
    this.add(new Label("Enter the message to appear in the dialog:"), "North");

    // And a big text area below it for entering the dialog message.
    message = new TextArea(bean.getMessage());
    message.addTextListener(this);
    // TextAreas don't know how big they want to be.  You must tell them.
    message.setSize(400, 200);
    this.add(message, "Center");

    // Then add a row of textfields for entering the button labels.
    Panel buttonbox = new Panel();                     // The row container
    buttonbox.setLayout(new GridLayout(1, 0, 25, 10)); // Equally spaced items
    this.add(buttonbox, "South");                      // Put row on bottom

    // Now go create three TextFields to put in this row.  But actually
    // position a Label above each, so create an container for each
    // TextField+Label combination.
    fields = new TextComponent[3];           // Array of TextFields.
    String[] labels = new String[] {         // Labels for each.
      "Yes Button Label", "No Button Label", "Cancel Button Label"};
    String[] values = new String[] {         // Initial values of each.
      bean.getYesLabel(), bean.getNoLabel(), bean.getCancelLabel()};
    for(int i = 0; i < 3; i++) {
      Panel p = new Panel();                 // Create a container.
      p.setLayout(new BorderLayout());       // Give it a BorderLayout.
      p.add(new Label(labels[i]), "North");  // Put a label on the top.
      fields[i] = new TextField(values[i]);  // Create the text field.
      p.add(fields[i], "Center");             // Put it below the label.
      fields[i].addTextListener(this);        // Set the event listener.
      buttonbox.add(p);                       // Add container to row.
    }
  }

  // Add some space around the outside of the panel.
  public Insets getInsets() { return new Insets(10, 10, 10, 10); }

  // This is the method defined by the TextListener interface.  Whenever the
  // user types a character in the TextArea or TextFields, this will get
  // called.  It updates the appropriate property of the bean and fires a
  // property changed event, as all customizers are required to do.
  // Note that we are not required to fire an event for every keystroke.
  // Instead we could include an "Apply" button that would make all the
  // changes at once, with a single property changed event.
  public void textValueChanged(TextEvent e) {
    TextComponent t = (TextComponent)e.getSource();
    String s = t.getText();
    if (t == message) bean.setMessage(s);
    else if (t == fields[0]) bean.setYesLabel(s);
    else if (t == fields[1]) bean.setNoLabel(s);
    else if (t == fields[2]) bean.setCancelLabel(s);
    listeners.firePropertyChange(null, null, null);
  }

  // This code uses the PropertyChangeSupport class to maintain a list of
  // listeners interested in the edits we make to the bean.
  protected PropertyChangeSupport listeners = new PropertyChangeSupport(this);
  public void addPropertyChangeListener(PropertyChangeListener l) {
    listeners.addPropertyChangeListener(l);
  }
  public void removePropertyChangeListener(PropertyChangeListener l) {
    listeners.removePropertyChangeListener(l);
  }
}
```

```
// This example is from the book "Java in a Nutshell, Second Edition".
// Written by David Flanagan.  Copyright (c) 1997 O'Reilly & Associates.
// You may distribute this source code for non-commercial purposes only.
// You may study, modify, and use this example for any purpose, as long as
// this notice is retained.  Note that this example is provided "as is",
// WITHOUT WARRANTY of any kind either expressed or implied.

import java.io.*;

/** A program to convert from one character encoding to another */
public class ConvertEncoding {
  public static void main(String[] args) {
    String from = null, to = null;
    String infile = null, outfile = null;
    for(int i = 0; i < args.length; i++) {  // Parse command-line arguments.
      if (i == args.length-1) usage();      // All legal args require another.
      if (args[i].equals("-from")) from = args[++i];
      else if (args[i].equals("-to")) to = args[++i];
      else if (args[i].equals("-in")) infile = args[++i];
      else if (args[i].equals("-out")) outfile = args[++i];
      else usage();
    }

    try { convert(infile, outfile, from, to); }  // Attempt conversion.
    catch (Exception e) {                        // Handle possible exceptions.
      LocalizedError.display(e);  // Defined at the end of this chapter.
      System.exit(1);
    }
  }

  public static void usage() {
    System.err.println("Usage: java ConvertEncoding <options>\n" +
                       "Options:\n\t-from <encoding>\n\t-to <encoding>\n\t" +
                       "-in <file>\n\t-out <file>");
    System.exit(1);
  }

  public static void convert(String infile, String outfile,
                             String from, String to)
       throws IOException, UnsupportedEncodingException
  {
    // Set up byte streams.
    InputStream in;
    if (infile != null) in = new FileInputStream(infile);
    else in = System.in;
    OutputStream out;
    if (outfile != null) out = new FileOutputStream(outfile);
    else out = System.out;

    // Use default encoding if no encoding is specified.
    if (from == null) from = System.getProperty("file.encoding");
    if (to == null) to = System.getProperty("file.encoding");

    // Set up character streams.
    Reader r = new BufferedReader(new InputStreamReader(in, from));
    Writer w = new BufferedWriter(new OutputStreamWriter(out, to));

    // Copy characters from input to output.  The InputStreamReader converts
    // from the input encoding to Unicode, and the OutputStreamWriter converts
    // from Unicode to the output encoding.  Characters that cannot be
    // represented in the output encoding are output as '?'
    char[] buffer = new char[4096];
    int len;
    while((len = r.read(buffer)) != -1)  // Read a block of input.
      w.write(buffer, 0, len);           // And write it out.
    r.close();                           // Close the input.
    w.flush();                           // Flush and close output.
    w.close();
  }
}
```

```java
// This example is from the book "Java in a Nutshell, Second Edition".
// Written by David Flanagan.  Copyright (c) 1997 O'Reilly & Associates.
// You may distribute this source code for non-commercial purposes only.
// You may study, modify, and use this example for any purpose, as long as
// this notice is retained.  Note that this example is provided "as is",
// WITHOUT WARRANTY of any kind either expressed or implied.

import java.text.*;
import java.util.Date;

/**
 * A partial implementation of a hypothetical stock portfolio class.
 * We use it only to demonstrate number and date internationalization.
 */
public class Portfolio {
  EquityPosition[] positions;
  Date lastQuoteTime = new Date();

  public void print() {
    // Obtain NumberFormat and DateFormat objects to format our data.
    NumberFormat number = NumberFormat.getInstance();
    NumberFormat price = NumberFormat.getCurrencyInstance();
    NumberFormat percent = NumberFormat.getPercentInstance();
    DateFormat shortdate = DateFormat.getDateInstance(DateFormat.SHORT);
    DateFormat fulldate = DateFormat.getDateTimeInstance(DateFormat.LONG,
                                                         DateFormat.LONG);

    // Print some introductory data.
    System.out.println("Portfolio value at " +
                       fulldate.format(lastQuoteTime) + ":");
    System.out.println("Symbol\tShares\tBought On\tAt\t" +
                       "Quote\tChange");

    // Then display the table using the format() methods of the Format objects.
    for(int i = 0; i < positions.length; i++) {
      System.out.print(positions[i].name + "\t");
      System.out.print(number.format(positions[i].shares) + "\t");
      System.out.print(shortdate.format(positions[i].purchased) + "\t");
      System.out.print(price.format(positions[i].bought) + "\t");
      System.out.print(price.format(positions[i].current) + "\t");
      double change =
        (positions[i].current - positions[i].bought)/positions[i].bought;
      System.out.println(percent.format(change));
    }
  }
  static class EquityPosition {
    String name;             // Name of the stock.
    int shares;              // Number of shares held.
    Date purchased;          // When purchased.
    double bought, current;  // Purchase price and current price (per share).
    EquityPosition(String n, int s, Date when, double then, double now) {
      name = n; shares = s; purchased = when; bought = then; current = now;
    }
  }
}
```

```java
// This example is from the book "Java in a Nutshell, Second Edition".
// Written by David Flanagan.  Copyright (c) 1997 O'Reilly & Associates.
// You may distribute this source code for non-commercial purposes only.
// You may study, modify, and use this example for any purpose, as long as
// this notice is retained.  Note that this example is provided "as is",
// WITHOUT WARRANTY of any kind either expressed or implied.

import java.awt.*;
import java.awt.event.*;
import java.util.Locale;
import java.util.ResourceBundle;
import java.util.MissingResourceException;

/** A convenience class to automatically create localized menu panes */
public class SimpleMenu {
  /** The convenience method that creates menu panes */
  public static Menu create(String bundlename,
                            String menuname, String[] itemnames,
                            ActionListener listener, boolean popup) {
    // Get the resource bundle used for this menu.
    ResourceBundle b = ResourceBundle.getBundle(bundlename);

    // Get the menu title from the bundle.  Use name as default label.
    String menulabel;
    try { menulabel = b.getString(menuname + ".label"); }
    catch(MissingResourceException e) { menulabel = menuname; }

    // Create the menu pane.
    Menu m;
    if (popup) m = new PopupMenu(menulabel);
    else m = new Menu(menulabel);

    // For each named item in the menu.
    for(int i = 0; i < itemnames.length; i++) {
      // Look up the label for the item, using name as default.
      String itemlabel;
      try { itemlabel=b.getString(menuname + "." + itemnames[i] + ".label"); }
      catch (MissingResourceException e) { itemlabel = itemnames[i]; }

      // Look up a shortcut for the item, and create the menu shortcut, if any.
      String shortcut;
      try{shortcut = b.getString(menuname + "." + itemnames[i]+".shortcut"); }
      catch (MissingResourceException e) { shortcut = null; }
      MenuShortcut ms = null;
      if (shortcut != null) ms = new MenuShortcut(shortcut.charAt(0));

      // Create the menu item.
      MenuItem mi;
      if (ms != null) mi = new MenuItem(itemlabel, ms);
      else mi = new MenuItem(itemlabel);

      // Register an action listener and command for the item.
      if (listener != null) {
        mi.addActionListener(listener);
        mi.setActionCommand(itemnames[i]);
      }

      // Add the item to the menu.
      m.add(mi);
    }

    // Return the automatically created localized menu.
    return m;
  }

  /** A simple test program for the above code */
  public static void main(String[] args) {
    // Set the default locale based on the command-line args.
    if (args.length == 2) Locale.setDefault(new Locale(args[0], args[1]));

    Frame f = new Frame("SimpleMenu Test");  // Create a window.
    MenuBar menubar = new MenuBar();         // Create a menubar.
    f.setMenuBar(menubar);                   // Add menubar to window.

    // Create a menu using our convenience routine (and the default locale).
    Menu colors = SimpleMenu.create("Menus", "colors",
                                    new String[] { "red", "green", "blue" },
                                    null, false);

    menubar.add(colors);                     // Add the menu to the menubar.
    f.setSize(300, 150);                     // Set the window size.
    f.show();                                // Pop the window up.
  }
}
```

```
# The file Menus.properties is the default "Menus" resource bundle.
# As an American programmer, I made my own locale the default.
colors.label=Colors
colors.red.label=Red
colors.red.shortcut=R
colors.green.label=Green
colors.green.shortcut=G
colors.blue.label=Blue
colors.blue.shortcut=B
```

```
# This is the file Menus_en_GB.properties.  It is the resource bundle for
# British English.  Note that it overrides only a single resource definition
# and simply inherits the rest from the default (American) bundle.
colors.label=Colours
```

```
# This is the file Menus_fr.properties.  It is the resource bundle for all
# French-speaking locales.  It overrides most, but not all, of the resources
# in the default bundle.
colors.label=Couleurs
colors.red.label=Rouge
colors.green.label=Vert
colors.green.shortcut=V
colors.blue.label=Bleu
```

```java
// This example is from the book "Java in a Nutshell, Second Edition".
// Written by David Flanagan.  Copyright (c) 1997 O'Reilly & Associates.
// You may distribute this source code for non-commercial purposes only.
// You may study, modify, and use this example for any purpose, as long as
// this notice is retained.  Note that this example is provided "as is",
// WITHOUT WARRANTY of any kind either expressed or implied.

import java.text.*;
import java.io.*;
import java.util.*;

/**
 * A convenience class that can display a localized exception message
 * depending on the class of the exception.  It uses a MessageFormat,
 * and passes five arguments that the localized message may include:
 *   {0}: the message included in the exception or error.
 *   {1}: the full class name of the exception or error.
 *   {2}: a guess at what file the exception was caused by.
 *   {3}: a line number in that file.
 *   {4}: the current date and time.
 * Messages are looked up in a ResourceBundle with the basename
 * "Errors", using a the full class name of the exception object as
 * the resource name.  If no resource is found for a given exception
 * class, the superclasses are checked.
 */
public class LocalizedError {
  public static void display(Throwable error) {
    ResourceBundle bundle;
    // Try to get the resource bundle.
    // If none, print the error in a non-localized way.
    try { bundle = ResourceBundle.getBundle("Errors"); }
    catch (MissingResourceException e) {
      error.printStackTrace(System.err);
      return;
    }

    // Look up a localized message resource in that bundle, using the
    // classname of the error (or its superclasses) as the resource name.
    // If no resource was found, display the error without localization.
    String message = null;
    Class c = error.getClass();
    while((message == null) && (c != Object.class)) {
      try { message = bundle.getString(c.getName()); }
      catch (MissingResourceException e) { c = c.getSuperclass(); }
    }
    if (message == null) { error.printStackTrace(System.err);  return; }

    // Try to figure out the filename and line number of the
    // exception.  Output the error's stack trace into a string, and
    // use the heuristic that the first line number that appears in
    // the stack trace is after the first or  second colon.  We assume that
    // this stack frame is the first one the programmer has any control
    // over, and so report it as the location of the exception.
    String filename = "";
    int linenum = 0;
    try {
      StringWriter sw = new StringWriter();    // Output stream into a string.
      PrintWriter out = new PrintWriter(sw);   // PrintWriter wrapper.
      error.printStackTrace(out);              // Print stacktrace.
      String trace = sw.toString();            // Get it as a string.
      int pos = trace.indexOf(':');            // Look for first colon.
      if (error.getMessage() != null)          // If the error has a message
        pos = trace.indexOf(':', pos+1);       // look for second colon.
      int pos2 = trace.indexOf(')', pos);      // Look for end of line number.
      linenum = Integer.parseInt(trace.substring(pos+1,pos2));  // Get linenum.
      pos2 = trace.lastIndexOf('(', pos);      // Back to start of filename.
      filename = trace.substring(pos2+1, pos); // Get filename.
    }
    catch (Exception e) { ; }                  // Ignore exceptions.

    // Set up an array of arguments to use with the message
    String errmsg = error.getMessage();
    Object[] args = {
      ((errmsg!= null)?errmsg:""), error.getClass().getName(),
      filename, new Integer(linenum), new Date()
    };
    // Finally, display the localized error message, using
    // MessageFormat.format() to substitute the arguments into the message.
    System.out.println(MessageFormat.format(message, args));
  }
}
```

```
#
# This is the file Errors.properties
# One property for each class of exceptions that our program might
# report.  Note the use of backslashes to continue long lines onto the
# next.  Also note the use of \n and \t for newlines and tabs
#
java.io.FileNotFoundException: \
Error: File "{0}" not found\n\t\
Error occurred at line {3} of file "{2}"\n\tat {4}

java.io.UnsupportedEncodingException: \
Error: Specified encoding not supported\n\t\
Error occurred at line {3} of file "{2}"\n\tat {4,time} on {4,date}

java.io.CharConversionException:\
Error: Character conversion failure.  Input data is not in specified format.

# A generic resource.  Display a message for any error or exception that
# is not handled by a more specific resource.
java.lang.Throwable:\
Error: {1}: {0}\n\t\
Error occurred at line {3} of file "{2}"\n\t{4,time,long} {4,date,long}
```

```
// This example is from the book "Java in a Nutshell, Second Edition".
// Written by David Flanagan.  Copyright (c) 1997 O'Reilly & Associates.
// You may distribute this source code for non-commercial purposes only.
// You may study, modify, and use this example for any purpose, as long as
// this notice is retained.  Note that this example is provided "as is",
// WITHOUT WARRANTY of any kind either expressed or implied.

import java.lang.reflect.*;

/** A program that displays a class synopsis for the named class */
public class ShowClass {
  /** The main method.  Print info about the named class */
  public static void main(String[] args) throws ClassNotFoundException {
    Class c = Class.forName(args[0]);
    print_class(c);
  }

  /** Display the modifiers, name, superclass and interfaces of a class
   *  or interface. Then go and list all constructors, fields, and methods. */
  public static void print_class(Class c)
  {
    // Print modifiers, type (class or interface), name and superclass.
    if (c.isInterface()) {
      // The modifiers will include the "interface" keyword here...
      System.out.print(Modifier.toString(c.getModifiers()) +  c.getName());
    }
    else
      System.out.print(Modifier.toString(c.getModifiers()) + " class " +
                       c.getName() +
                       " extends " + c.getSuperclass().getName());

    // Print interfaces or super-interfaces of the class or interface.
    Class[] interfaces = c.getInterfaces();
    if ((interfaces != null) && (interfaces.length > 0)) {
      if (c.isInterface()) System.out.println(" extends ");
      else System.out.print(" implements ");
      for(int i = 0; i < interfaces.length; i++) {
        if (i > 0) System.out.print(", ");
        System.out.print(interfaces[i].getName());
      }
    }

    System.out.println(" {");            // Begin class member listing.

    // Now look up and display the members of the class.
    System.out.println(" // Constructors");
    Constructor[] constructors = c.getDeclaredConstructors();
    for(int i = 0; i < constructors.length; i++)      // Display constructors.
      print_method_or_constructor(constructors[i]);

    System.out.println(" // Fields");
    Field[] fields = c.getDeclaredFields();           // Look up fields.
    for(int i = 0; i < fields.length; i++)            // Display them.
      print_field(fields[i]);

    System.out.println(" // Methods");
    Method[] methods = c.getDeclaredMethods();        // Look up methods.
    for(int i = 0; i < methods.length; i++)           // Display them.
      print_method_or_constructor(methods[i]);

    System.out.println("}");             // End class member listing.
  }
  /** Return the name of an interface or primitive type, handling arrays. */
  public static String typename(Class t) {
    String brackets = "";
    while(t.isArray()) {
      brackets += "[]";
      t = t.getComponentType();
    }
    return t.getName() + brackets;
  }

  /** Return a string version of modifiers, handling spaces nicely. */
  public static String modifiers(int m) {
    if (m == 0) return "";
    else return Modifier.toString(m) + " ";
  }

  /** Print the modifiers, type, and name of a field */
  public static void print_field(Field f) {
    System.out.println("  " +
                       modifiers(f.getModifiers()) +
                       typename(f.getType()) + " " + f.getName() + ";");
  }

  /** Print the modifiers, return type, name, parameter types and exception
   *  type of a method or constructor.  Note the use of the Member interface
   *  to allow this method to work with both Method and Constructor objects */
  public static void print_method_or_constructor(Member member) {
    Class returntype=null, parameters[], exceptions[];
    if (member instanceof Method) {
      Method m = (Method) member;
      returntype = m.getReturnType();
      parameters = m.getParameterTypes();
      exceptions = m.getExceptionTypes();
    } else {
      Constructor c = (Constructor) member;
      parameters = c.getParameterTypes();
      exceptions = c.getExceptionTypes();
    }

    System.out.print("  " + modifiers(member.getModifiers()) +
                     ((returntype!=null)? typename(returntype)+" " : "") +
                     member.getName() + "(");
    for(int i = 0; i < parameters.length; i++) {
      if (i > 0) System.out.print(", ");
      System.out.print(typename(parameters[i]));
    }
    System.out.print(")");
    if (exceptions.length > 0) System.out.print(" throws ");
    for(int i = 0; i < exceptions.length; i++) {
      if (i > 0) System.out.print(", ");
      System.out.print(typename(exceptions[i]));
    }
    System.out.println(";");
  }
}
```

```
// This example is from the book "Java in a Nutshell, Second Edition".
// Written by David Flanagan.  Copyright (c) 1997 O'Reilly & Associates.
// You may distribute this source code for non-commercial purposes only.
// You may study, modify, and use this example for any purpose, as long as
// this notice is retained.  Note that this example is provided "as is",
// WITHOUT WARRANTY of any kind either expressed or implied.

import java.awt.event.*;
import java.lang.reflect.*;
import java.awt.*;   // Only used for the test program below.

public class UniversalActionListener implements ActionListener {
  protected Object target;
  protected Object arg;
  protected Method m;

  public UniversalActionListener(Object target, String methodname, Object arg)
       throws NoSuchMethodException, SecurityException
  {
    this.target = target;                            // Save the target object.
    this.arg = arg;                                  // And method argument.

    // Now look up and save the Method to invoke on that target object
    Class c, parameters[];
    c = target.getClass();                           // The Class object.
    if (arg == null) parameters = new Class[0];      // Method parameter.
    else parameters = new Class[] { arg.getClass() };
    m = c.getMethod(methodname, parameters);         // Find matching method.
  }

  public void actionPerformed(ActionEvent event) {
    Object[] arguments;
    if (arg == null) arguments = new Object[0];      // Set up arguments.
    else arguments = new Object[] { arg };
    try { m.invoke(target, arguments); }             // And invoke the method.
    catch (IllegalAccessException e) {               // Should never happen.
      System.err.println("UniversalActionListener: " + e);
    } catch (InvocationTargetException e) {          // Should never happen.
      System.err.println("UniversalActionListener: " + e);
    }
  }

  // A simple test program for the UniversalActionListener
  public static void main(String[] args) throws NoSuchMethodException {
    Frame f = new Frame("UniversalActionListener Test");// Create window.
    f.setLayout(new FlowLayout());                      // Set layout manager.
    Button b1 = new Button("tick");                     // Create buttons.
    Button b2 = new Button("tock");
    Button b3 = new Button("Close Window");
    f.add(b1); f.add(b2); f.add(b3);                    // Add them to window.

    // Specify what the buttons do.  Invoke a named method with
    // the UniversalActionListener object.
    b1.addActionListener(new UniversalActionListener(b1, "setLabel", "tock"));
    b1.addActionListener(new UniversalActionListener(b2, "setLabel", "tick"));
    b1.addActionListener(new UniversalActionListener(b3, "hide", null));
    b2.addActionListener(new UniversalActionListener(b1, "setLabel", "tick"));
    b2.addActionListener(new UniversalActionListener(b2, "setLabel", "tock"));
    b2.addActionListener(new UniversalActionListener(b3, "show", null));
    b3.addActionListener(new UniversalActionListener(f, "dispose", null));

    f.pack();                                             // Set window size.
    f.show();                                              // And pop it up.
  }
}
```

Index

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Symbols and Numbers

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

HOME

HOME

Index

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

HOME

Index

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

HOME

HOME

Index

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Index

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Index

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

HOME

HOME

Index

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

HOME

HOME

Index

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

HOME

HOME

Index

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

HOME

HOME

Index

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

HOME

Index

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

HOME

HOME

Index

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

HOME

Index

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

HOME

Index

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

HOME

Index

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

HOME

Index

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

HOME

Index

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

HOME

HOME

Index

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

HOME

Index

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Index

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

HOME

Index

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

HOME

Index

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

HOME

HOME

Index

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

HOME

Index

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Index

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

HOME

HOME

Index

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

HOME

Index

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

6.3 Synchronization

Every thread has a life of its own. Normally, a thread goes about its business without any regard for what other threads in the application are doing. Threads may be time-sliced, which means they can run in arbitrary spurts and bursts as directed by the operating system. On a multiprocessor system, it is even possible for many different threads to be running simultaneously on different CPUs. This section is about coordinating the activities of two or more threads, so they can work together and not collide in their use of the same address space.

Java provides a few simple structures for synchronizing the activities of threads. They are all based on the concept of *monitors*, a widely used synchronization scheme developed by C.A.R. Hoare. You don't have to know the details about how monitors work to be able to use them, but it may help you to have a picture in mind.

A monitor is essentially a lock. The lock is attached to a resource that many threads may need to access, but that should be accessed by only one thread at a time. It's not unlike a public restroom at a gas station. If the resource is not being used, the thread can acquire the lock and access the resource. By the same token, if the restroom is unlocked, you can enter and lock the door. When the thread is done, it relinquishes the lock, just as you unlock the door and leave it open for the next person. However, if another thread already has the lock for the resource, all other threads have to wait until the current thread finishes and releases the lock, just as if the restroom is locked when you arrive, you have to wait until the current occupant is done and unlocks the door.

Fortunately, Java makes the process of synchronizing access to resources quite easy. The language handles setting up and acquiring locks; all you have to do is specify which resources require locks.

Serializing Methods

The most common need for synchronization among threads in Java is to serialize their access to some resource, namely an object. In other words, synchronization makes sure only one thread at a time can perform certain activities that manipulate an object. In Java, every object has a lock associated with it. To be more specific, every class and every instance of a class has its own lock. The `synchronized` keyword marks places where a thread must acquire the lock before proceeding.

For example, say we implemented a `SpeechSynthesizer` class that contains a `say()` method. We don't want multiple threads calling `say()` at the same time or we wouldn't be able to understand anything being said. So we mark the `say()` method as synchronized, which means that a thread has to acquire the lock on the `SpeechSynthesizer` object before it can speak:

```
class SpeechSynthesizer {

    synchronized void say( String words ) {
        // Speak
    }
}
```

Because `say()` is an instance method, a thread has to acquire the lock on the particular `SpeechSynthesizer` instance it is using before it can invoke the `say()` method. When `say()` has completed, it gives up the lock, which allows the next waiting thread to acquire the lock and run the method. Note that it doesn't matter whether the thread is owned by the `SpeechSynthesizer` itself or some other object; every thread has to acquire the same lock, that of the `SpeechSynthesizer` instance. If `say()` were a class (static) method instead of an instance method, we could still mark it as synchronized. But in this case as there is no instance object involved, the lock would be on the class object itself.

Often, you want to synchronize multiple methods of the same class, so that only one of the methods modifies or examines parts of the class at a time. All static synchronized methods in a class use the same class object lock. By the same token, all instance methods in a class use the same instance object lock. In this way, Java can guarantee that only one of a set of synchronized methods is running at a time. For example, a `SpreadSheet` class might contain a number of instance variables that represent cell values, as well as some methods that manipulate the cells in a row:

```
class SpreadSheet {

    int cellA1, cellA2, cellA3;

    synchronized int sumRow() {
        return cellA1 + cellA2 + cellA3;
    }

    synchronized void setRow( int a1, int a2, int a3 ) {
        cellA1 = a1;
        cellA2 = a2;
        cellA3 = a3;
    }
...
}
```

In this example, both methods `setRow()` and `sumRow()` access the cell values. You can see that problems might arise if one thread were changing the values of the variables in `setRow()` at the same moment another thread was reading the values in `sumRow()`. To prevent this, we have marked both methods as synchronized. When threads are synchronized, only one will be run at a time. If a thread is in the middle of executing `setRow()` when another thread calls `sumRow()`, the second thread waits until the first one is done executing `setRow()` before it gets to run `sumRow()`. This synchronization allows us to preserve the consistency of the `SpreadSheet`. And the best part is that all of this locking and waiting is handled by Java; it's transparent to the programmer.

In addition to synchronizing entire methods, the `synchronized` keyword can be used in a special construct to guard arbitrary blocks of code. In this form it also takes an explicit argument that specifies the object for which it is to acquire a lock:

```
synchronized ( myObject ) {
    // Functionality that needs to be synced
    ...
    }
```

The code block above can appear in any method. When it is reached, the thread has to acquire the lock on `myObject` before proceeding. In this way, we can have methods (or parts of methods) in different classes synchronized the same as methods in the same class.

A synchronized method is, therefore, equivalent to a method with its statements synchronized on the current object. Thus:

```
synchronized void myMethod () {
    ...
}
```

is equivalent to:

```
void myMethod () {
    synchronized ( this ) {
        ...
    }
}
```

wait( ) and notify( )

With the `synchronized` keyword, we can serialize the execution of complete methods and blocks of code. The `wait()` and `notify()` methods of the `Object` class extend this capability. Every object in Java is a subclass of `Object`, so every object inherits these methods. By using `wait()` and `notify()`, a thread can give up its hold on a lock at an arbitrary point, and then wait for another thread to give it back before continuing. All of the coordinated activity still happens inside of synchronized blocks, and still only one thread is executing at a given time.

By executing `wait()` from a synchronized block, a thread gives up its hold on the lock and goes to sleep. A thread might do this if it needs to wait for something to happen in another part of the application, as you'll see shortly. Later, when the necessary event happens, the thread that is running it calls `notify()` from a block synchronized on the same object. Now the first thread wakes up and begins trying to acquire the lock again.

When the first thread manages to reacquire the lock, it continues from the point it left off. However, the thread that waited may not get the lock immediately (or perhaps ever). It depends on when the second thread eventually releases the lock, and which thread manages to snag it next. Note also, that the first thread won't wake up from the `wait()` unless another thread calls `notify()`. There is an overloaded version of `wait()`, however, that allows us to specify a timeout period. If another thread doesn't call `notify()` in the specified period, the waiting thread automatically wakes up.

Let's look at a simple scenario to see what's going on. In the following example, we'll assume there are three threads--one waiting to execute each of the three synchronized methods of the `MyThing` class. We'll call them the `waiter`, `notifier`, and `related` threads, respectively. Here's a code fragment to illustrate:

```
class MyThing {

    synchronized void waiterMethod() {
        // Do some stuff

        // Now we need to wait for notifier to do something
        wait();

        // Continue where we left off
    }

    synchronized void notifierMethod() {
        // Do some stuff

        // Notify waiter that we've done it
        notify();
        // Do more things
    }

    synchronized void relatedMethod() {
        // Do some related stuff
    }
```

Let's assume `waiter` gets through the gate first and begins executing `waiterMethod()`. The two other threads are initially blocked, trying to acquire the lock for the `MyThing` object. When `waiter` executes the `wait()` method, it relinquishes its hold on the lock and goes to sleep. Now there are now two viable threads waiting for the lock. Which thread gets it depends on several factors, including chance and the priorities of the threads. (We'll discuss thread scheduling in the next section.)

Let's say that `notifier` is the next thread to acquire the lock, so it begins to run. `waiter` continues to sleep and `related` languishes, waiting for its turn. When `notifier` executes the call to `notify()`, Java prods the `waiter` thread, effectively telling it something has changed. `waiter` then wakes up and rejoins `related` in vying for the `MyThing` lock. Note that it doesn't actually receive the lock; it just changes from saying "leave me alone" to "I want the lock."

At this point, `notifier` still owns the lock and continues to hold it until it leaves its synchronized method (or perhaps executes a `wait()` itself). When it finally completes, the other two methods get to fight over the lock. `waiter` would like to continue executing `waiterMethod()` from the point it left off, while `unrelated`, which has been patient, would like to get started. We'll let you choose your own ending for the story.

For each call to `notify()`, Java wakes up just one method that is asleep in a `wait()` call. If there are multiple threads waiting, Java picks the first thread on a first-in, first-out basis. The `Object` class also provides a `notifyAll()` call to wake up all waiting threads. In most cases, you'll probably want to use `notifyAll()` rather than `notify()`. Keep in mind that `notify()` really means "Hey, something related to this object has changed. The condition you are waiting for may have changed, so check it again." In general, there is no reason to assume only one thread at a time is interested in the change or able to act upon it. Different threads might look upon whatever has changed in different ways.

Often, our `waiter` thread is waiting for a particular condition to change and we will want to sit in a loop like the following:

```
...
while ( condition != true )
    wait();
...
```

Other synchronized threads call `notify()` or `notifyAll()` when they have modified the environment so that `waiter` can check the condition again. This is the civilized alternative to polling and sleeping, as you'll see the following example.

The Message Passer

Now we'll illustrate a classic interaction between two threads: a `Producer` and a `Consumer`. A producer thread creates messages and places them into a queue, while a consumer reads them out and displays them. To be realistic, we'll give the queue a maximum depth. And to make things really interesting, we'll have our consumer thread be lazy and run much slower than the producer. This means that `Producer` occasionally has to stop and wait for `Consumer` to catch up. The example below shows the `Producer` and `Consumer` classes.

```
import java.util.Vector;

class Producer extends Thread {
    static final int MAXQUEUE = 5;
    private Vector messages = new Vector();

    public void run() {
        try {
            while ( true ) {
                putMessage();
                sleep( 1000 );
            }
        }
        catch( InterruptedException e ) { }
    }

    private synchronized void putMessage()
        throws InterruptedException {

        while ( messages.size() == MAXQUEUE )
            wait();
        messages.addElement( new java.util.Date().toString() );
        notify();
    }

    // Called by Consumer
    public synchronized String getMessage()
        throws InterruptedException {
        notify();
        while ( messages.size() == 0 )
            wait();
        String message = (String)messages.firstElement();
        messages.removeElement( message );
        return message;
    }
}

class Consumer extends Thread {
    Producer producer;

    Consumer(Producer p) {
        producer = p;
    }

    public void run() {
        try {
            while ( true ) {
                String message = producer.getMessage();
                System.out.println("Got message: " + message);
                sleep( 2000 );
            }
        }
        catch( InterruptedException e ) { }
    }

    public static void main(String args[]) {
        Producer producer = new Producer();
        producer.start();
        new Consumer( producer ).start();
    }
}
```

For convenience, we have included a `main()` method that runs the complete example in the `Consumer` class. It creates a `Consumer` that is tied to a `Producer` and starts the two classes. You can run the example as follows:

```
% java Consumer
```

The output is the time-stamp messages created by the `Producer`:

```
Got message: Sun Dec 19 03:35:55 CST 1996
Got message: Sun Dec 19 03:35:56 CST 1996
Got message: Sun Dec 19 03:35:57 CST 1996
...
```

The time stamps initially show a spacing of one second, although they appear every two seconds. Our `Producer` runs faster than our `Consumer`. `Producer` would like to generate a new message every second, while `Consumer` gets around to reading and displaying a message only every two seconds. Can you see how long it will take the message queue to fill up? What will happen when it does?

Let's look at the code. We are using a few new tools here. `Producer` and `Consumer` are subclasses of `Thread`. It would have been a better design decision to have `Producer` and `Consumer` implement the `Runnable` interface, but we took the slightly easier path and subclassed `Thread`. You should find it fairly simple to use the other technique; you might try it as an exercise.

The `Producer` and `Consumer` classes pass messages through an instance of a `java.util.Vector` object. We haven't discussed the `Vector` class yet, but you can think of this one as a queue where we add and remove elements in first-in, first-out order. See Chapter 7, *Basic Utility Classes* for more information about the `Vector` class.

The important activity is in the synchronized methods: `putMessage()` and `getMessage()`. Although one of the methods is used by the `Producer` thread and the other by the `Consumer` thread, they both live in the `Producer` class because they have to be synchronized on the same object to work together. Here they both implicitly use the `Producer` object's lock. If the queue is empty, the `Consumer` blocks in a call in the `Producer`, waiting for another message.

Another design option would implement the `getMessage()` method in the `Consumer` class and use a synchronized code block to explicitly synchronize on the `Producer` object. In either case, synchronizing on the `Producer` is important because it allows us to have multiple `Consumer` objects that feed on the same `Producer`.

`putMessage()`'s job is to add a new message to the queue. It can't do this if the queue is already full, so it first checks the number of elements in `messages`. If there is room, it stuffs in another time stamp. If the queue is at its limit however, `putMessage()` has to wait until there's space. In this situation, `putMessage()` executes a `wait()` and relies on the consumer to call `notify()` to wake it up after a message has been read. Here we have `putMessage()` testing the condition in a loop. In this simple example, the test probably isn't necessary; we could assume that when `putMessage()` wakes up, there is a free spot. However, this test is another example of good programming practice. Before it finishes, `putMessage()` calls `notify()` itself to prod any `Consumer` that might be waiting on an empty queue.

`getMessage()` retrieves a message for the `Consumer`. It enters a loop like the `Producer`'s, waiting for the queue to have at least one element before proceeding. If the queue is empty, it executes a `wait()` and expects the producer to call `notify()` when more items are available. Notice that `getMessage()` makes its own unconditional call to `notify()`. This is a somewhat lazy way of keeping the `Producer` on its toes, so that the queue should generally be full. Alternatively, `getMessage()` might test to see if the queue had fallen below a low water mark before waking up the producer.

Now let's add another Consumer to the scenario, just to make things really interesting. Most of the necessary changes are in the Consumer class; the example below shows the code for the modified class.

```
class Consumer extends Thread {
    Producer producer;
        String name;

    Consumer(String name, Producer producer) {
        this.producer = producer;
        this.name = name;
    }

    public void run() {
        try {
            while ( true ) {
                String message = producer.getMessage();
                System.out.println(name + " got message: " + message);
                sleep( 2000 );
            }
        }
        catch( InterruptedException e ) { }
    }

    public static void main(String args[]) {
        Producer producer = new Producer();
        producer.start();

        // Start two this time
        new Consumer( "One", producer ).start();
        new Consumer( "Two", producer ).start();
    }
}
```

The Consumer constructor now takes a string name, to identify each consumer. The run() method uses this name in the call to println() to identify which consumer received the message.

The only modification to make in the Producer code is to change the call to notify() in putMessage() to a call to notifyAll(). Now, instead of the consumer and producer playing tag with the queue, we can have many players waiting on the condition of the queue to change. We might have a number of consumers waiting for a message, or we might have the producer waiting for a consumer to take a message. Whenever the condition of the queue changes, we prod all of the waiting methods to reevaluate the situation by calling `notifyAll()`. Note, however, that we don't need to change the call to `notify()` in `getMessage()`. If a `Consumer` thread is waiting for a message to appear in the queue, it's not possible for the `Producer` to be simultaneously waiting because the queue is full.

Here is some sample output when there are two consumers running, as in the `main()` method shown above:

```
One got message: Wed Mar 20 20:00:01 CST 1996
Two got message: Wed Mar 20 20:00:02 CST 1996
One got message: Wed Mar 20 20:00:03 CST 1996
Two got message: Wed Mar 20 20:00:04 CST 1996
One got message: Wed Mar 20 20:00:05 CST 1996
Two got message: Wed Mar 20 20:00:06 CST 1996
One got message: Wed Mar 20 20:00:07 CST 1996
Two got message: Wed Mar 20 20:00:08 CST 1996
...
```

We see nice, orderly alternation between the two consumers, as a result of the calls to `sleep()` in the various methods. Interesting things would happen, however, if we were to remove all of the calls to `sleep()` and let things run at full speed. The threads would compete and their behavior would depend on whether or not the system is using time slicing. On a time-sliced system, there should be a fairly random distribution between the two consumers, while on a nontime-sliced system, a single consumer could monopolize the messages. And since you're probably wondering about time slicing, let's talk about thread priority and scheduling.

PREVIOUS Threading Applets | HOME BOOK INDEX | NEXT Scheduling and Priority

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

6.2 Threading Applets

Applets are embeddable Java applications that are expected to be able to start and stop themselves on command. Unlike threads, applets can be started and stopped any number of times. A Java-enabled Web browser normally starts an applet when the applet is displayed and stops it when the user moves to another page or scrolls the applet out of view. In general, we would like an applet to cease its nonessential activity when it is stopped, and resume it when started again. (See Chapter 10, *Understand the Abstract Windowing Toolkit* for a complete discussion of applets.)

In this section, we will build `UpdateApplet`, a simple base class for an `Applet` that maintains a `Thread` to automatically update its display at regular intervals. `UpdateApplet` handles the basic starting and stopping behavior for us, as shown below.

```
public class UpdateApplet extends java.applet.Applet
    implements Runnable {

    private Thread updateThread;
    int updateInterval = 1000;

    public void run() {
        while ( true ) {
            try {
                Thread.sleep( updateInterval );
            }
            catch (InterruptedException e ) { }

            repaint();
        }
    }

    public void start() {
        if ( updateThread == null ) {
            updateThread = new Thread(this);
            updateThread.start();
        }
    }

    public void stop() {
        if ( updateThread != null ) {
            updateThread.stop();
            updateThread = null;
        }
    }
}
```

UpdateApplet is a Runnable object that alternately sleeps and calls its repaint() method. It has two other public methods: start() and stop(). These are methods of the Applet class we are overriding; do not confuse them with the similarly named methods of the Thread class. These start() and stop() methods are called by the Java environment to tell the applet when it should and should not be running.

UpdateApplet illustrates an environmentally friendly way to deal with threads in a simple applet. UpdateApplet kills its thread each time the applet is stopped and recreates it if the applet is restarted. When UpdateApplet's start() method is called, we first check to make sure there is no currently executing updateThread. We then create one to begin our execution. When our applet is subsequently stopped, we kill the thread by invoking its stop() method and throw away the reference by setting it to null. Setting updateThread to null serves both to allow the garbage collector to clean up the dead Thread object, and to indicate to UpdateApplet's start() method that the thread is gone.

In truth, an Applet's start() and stop() methods are guaranteed to be called in sequence. As a result, we shouldn't have to check for the existence of updateThread in start() (it should always be null). However, it's good programming practice to perform the test. If we didn't, and for some reason stop() were to fail at its job, we might inadvertently start a lot of threads.

With UpdateApplet doing all of the work for us, we can now create the world's simplest clock applet with just a few lines of code. Figure 6.3 shows our Clock. (This might be a good one to run on your Java wrist watch.)

```
public class Clock extends UpdateApplet {
    public void paint( java.awt.Graphics g ) {
        g.drawString( new java.util.Date().toString(), 10, 25 );
    }
}
```

Figure 6.3: The clock applet

[Graphic: Figure 6-3]

The `java.util.Date().toString()` sequence simply creates a string that contains the current time; we'll see where this code comes from in Chapter 7, *Basic Utility Classes*.

Our `Clock` applet provides a good example of a simple thread; we don't mind throwing it away and subsequently rebuilding it if the user should happen to wander on and off of our Web page a few times. But what if the task that our thread handles isn't so simple? What if, for instance, we have to open a socket and establish a connection with another system? One solution is to use `Thread`'s `suspend()` and `resume()` methods, as I'll show you momentarily.

Now if you've been wondering why we've been using `stop()` to kill the thread, rather than using the `suspend()` and `resume()` methods all along, here's the explanation you've been waiting for. The problem with applets is that we have no control over how a user navigates Web pages. For example, say a user scrolls our applet out of view, and we use `suspend()` to suspend the applet. Now we have no way of ensuring that the user will bring the applet back into view before moving on to another page. And actually, the same situation would occur if the user simply moves on to another page and never comes back.

If we call `suspend()`, we'd really like to make sure we call `resume()` at a later date, or we'll end up leaving the thread hanging in permanent suspense. But we have no way of knowing if the applet will ever be restarted, so just putting a call to `resume()` in the applet's `start()` method won't work. Leaving the suspended thread around forever might not hurt us, but it's not good programming practice to be wasteful. What we need is a way to guarantee we can clean up our mess if the applet is never used again. What to do?

There is a solution for this dilemma, but in many cases, like with our simple `Clock`, it's just easier to use `stop()`, with a subsequent call to `start()` if necessary. In cases where it is expensive to set up and tear down a thread, we could make the following modifications to `UpdateApplet`:

```
public void start() {
    if ( updateThread == null ) {
        updateThread = new Thread(this);
        updateThread.start();
    }
    else
        updateThread.resume();
}

public void stop() {
    updateThread.suspend();

public void destroy() {
    if ( updateThread != null ) {
        updateThread.stop();
        updateThread = null;
    }
}
```

These modifications change `UpdateApplet` so that it suspends and restarts its `updateThread`, rather than killing and recreating it. The new `start()` method creates the thread and calls `start()` if `updateThread` is `null`; otherwise it assumes that the thread has been suspended, so it calls `resume()`. The applet's `stop()` method simply suspends the thread by calling `suspend()`.

What's new here is the `destroy()` method. This is another method that `UpdateApplet` inherits from the `Applet` class. The method is called by the Java environment when the applet is going to be removed (often from a cache). It provides a place where we can free up any resources the applet is holding. This is the perfect place to cut the suspense and clean up after our thread. In our `destroy()` method, we check to see that the thread exists, and if it does, we call `stop()` to kill it and set its reference to `null`.

PREVIOUS | HOME | NEXT
Introducing Threads | BOOK INDEX | Synchronization

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

5.2 Vectors

The Vector class implements a variable-length array that can hold any kind of object. Like an array, the elements in a Vector are accessed with an integer index. However, unlike an array, the size of a Vector can grow and shrink as needed to accommodate a changing number of objects. Vector provides methods to add and remove elements, as well as ways to search for objects in a Vector and iterate through all of the objects.

You can create a Vector object using the constructor that takes no arguments.

```
Vector v = new Vector()
```

This constructor creates an empty Vector with an initial capacity of 10. The capacity of a Vector specifies how many objects it can contain before more space must be allocated. You can improve the performance of a Vector by setting its initial capacity to a more appropriate value when you create it. For example, if you know that you are going to be storing close to 100 objects in a Vector, you could set the initial capacity as follows:

```
Vector v = new Vector(100)
```

It can be time-consuming for a Vector to increase its capacity, so it is better to set the initial capacity based on a rough estimate of the number of objects a Vector will contain than to simply use the default capacity.

The capacity increment of a Vector specifies how much more space is allocated each time the Vector needs to increase its capacity. If you do not specify a capacity increment when you create a Vector, it uses the default value of 0, which causes the Vector to double in size every time it needs to increase its capacity. Doubling in size is a good way for a Vector to become large enough quickly when you have no idea what size it needs to be. However, if you do have a rough idea of the final size of a Vector, specifying a positive capacity increment is less wasteful of memory. For example, if you know that you will be putting 100 or so objects in a Vector, you could create it as follows:

```
Vector v = new Vector(110, 20)
```

Once you have created an empty `Vector` object, you can put object references in it using the `addElement()` and `insertElementAt()` methods. The `addElement()` method adds an element to the end of a `Vector`. The following code fragment shows the use of the `addElement()` method:

```
Vector v = new Vector();
v.addElement("abc");
v.addElement("jkl");
v.addElement("xyz");
```

The `insertElementAt()` method inserts a new element into a `Vector` before a given position, so it can be used to insert an element at any position in a `Vector` except the last. Like arrays, `Vector` objects are indexed starting at 0. Here's how to insert an object at the beginning of the `Vector` object created above:

```
v.insertElementAt("123", 0);
```

The `size()` method returns the number of elements in a `Vector` object.

After you have added some elements to a `Vector` object, you can retrieve elements with a number of different methods. For example, the `elementAt()` method fetches the object at the specified position in the `Vector`, while the `firstElement()` and `lastElement()` methods return the first and last objects in the `Vector`, respectively. Finally, the `elements()` method returns an `Enumeration` object that accesses the elements in the `Vector` object.

The `setElementAt()` method allows you to change the object stored at a specified position in the `Vector`, while the `removeElementAt()` method removes the object at a specified position from the `Vector`. The `removeElement()` method takes an object reference as an argument and removes the first element in the `Vector` that refers to the given object, if there is such an element. You can also remove all of the elements from the `Vector` using the `removeAllElements()` method.

The `Vector` class also provides some methods for searching the contents of a `Vector` object. For example, the `contains()` method returns `true` if a `Vector` contains a reference to a specified object. The `indexOf()` and `lastIndexOf()` methods return the positions of the first and last elements, respectively, in a `Vector` that match a specified object.

PREVIOUS HOME NEXT
Enumerations BOOK INDEX Stacks

JAVA IN A NUTSHELL | JAVA LANG REF | JAVA AWT REF | JAVA FUND CLASSES REF | EXPLORING JAVA

Credits

The Books

Java in a Nutshell
By David Flanagan
1-56592-262-X, 628 pages.
2nd Edition, May 1997

Java Language Reference
By Mark Grand
1-56592-326-X, 450 pages (est.)
2nd Edition July 1997 (est.)

Java AWT Reference
By John Zukowski
1-56592-240-9, 1074 pages
1st Edition April 1997

Java Fundamental Classes Reference
By Mark Grand and Jonathan Knudsen
1-56592-241-7, 1136 pages (est.)
1st Edition June 1997 (est.)

Exploring Java
By Patrick Niemeyer & Joshua Peck
1-56592-184-271-9, 500 pages (est.)
2nd Edition July 1997 (est.)

The Java Reference Library CD-ROM

Product Manager
Lorrie LeJeune
Technical Development
Jessica Perry Hekman and Norman Walsh
Creative Director
Edie Freedman
Conversion Specialist
Erik Ray
Index

Seth Maislin