



Beginning

JavaScript®

3rd Edition

Paul Wilton, Jeremy McPeak



Updates, source code, and Wrox technical support at www.wrox.com

Beginning JavaScript®

Third Edition

Paul Wilton
Jeremy McPeak



Wiley Publishing, Inc.

Beginning JavaScript®

Third Edition

Beginning JavaScript®

Third Edition

Paul Wilton
Jeremy McPeak



Wiley Publishing, Inc.

Beginning JavaScript®, Third Edition

Published by
Wiley Publishing, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Copyright © 2007 by Wiley Publishing, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-0-470-05151-1

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Library of Congress Cataloging-in-Publication Data:

Wilton, Paul, 1969–

Beginning JavaScript / Paul Wilton, Jeremy McPeak.—3rd ed.
p. cm.

ISBN 978-0-470-05151-1 (paper/website)

1. JavaScript (Computer program language) 2. World Wide Web. 3. Web servers. I. McPeak, Jeremy, 1979– II. Title.

QA76.73.J39W55 2007

005.13'3—dc22

2007008102

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Legal Department, Wiley Publishing, Inc., 10475 Crosspoint Blvd., Indianapolis, IN 46256, (317) 572-3447, fax (317) 572-4355, or online at <http://www.wiley.com/go/permissions>.

LIMIT OF LIABILITY/DISCLAIMER OF WARRANTY: THE PUBLISHER AND THE AUTHOR MAKE NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS WORK AND SPECIFICALLY DISCLAIM ALL WARRANTIES, INCLUDING WITHOUT LIMITATION WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE. NO WARRANTY MAY BE CREATED OR EXTENDED BY SALES OR PROMOTIONAL MATERIALS. THE ADVICE AND STRATEGIES CONTAINED HEREIN MAY NOT BE SUITABLE FOR EVERY SITUATION. THIS WORK IS SOLD WITH THE UNDERSTANDING THAT THE PUBLISHER IS NOT ENGAGED IN RENDERING LEGAL, ACCOUNTING, OR OTHER PROFESSIONAL SERVICES. IF PROFESSIONAL ASSISTANCE IS REQUIRED, THE SERVICES OF A COMPETENT PROFESSIONAL PERSON SHOULD BE SOUGHT. NEITHER THE PUBLISHER NOR THE AUTHOR SHALL BE LIABLE FOR DAMAGES ARISING HEREFROM. THE FACT THAT AN ORGANIZATION OR WEBSITE IS REFERRED TO IN THIS WORK AS A CITATION AND/OR A POTENTIAL SOURCE OF FURTHER INFORMATION DOES NOT MEAN THAT THE AUTHOR OR THE PUBLISHER ENDORSES THE INFORMATION THE ORGANIZATION OR WEBSITE MAY PROVIDE OR RECOMMENDATIONS IT MAY MAKE. FURTHER, READERS SHOULD BE AWARE THAT INTERNET WEBSITES LISTED IN THIS WORK MAY HAVE CHANGED OR DISAPPEARED BETWEEN WHEN THIS WORK WAS WRITTEN AND WHEN IT IS READ.

For general information on our other products and services please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Trademarks: Wiley, the Wiley logo, Wrox, the Wrox logo, Programmer to Programmer, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. JavaScript is a registered trademark of Sun Microsystems Incorporated. All other trademarks are the property of their respective owners. Wiley Publishing, Inc., is not associated with any product or vendor mentioned in this book.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

In memory of my mum, June Wilton, who in 2006 lost her brave battle against cancer. She was always very proud of me and my books and showed my books to anyone and everyone she happened to meet however briefly and whether they wanted to see them or not! She's very much missed.

—Paul Wilton

To my parents, Jerry and Judy, for their love and support.

—Jeremy McPeak

About the Authors

Paul Wilton started as a Visual Basic applications programmer at the Ministry of Defence in the UK, then found himself pulled into the Net. Having joined an Internet development company, he spent three years helping create Internet solutions. He's now running his own successful and rapidly growing company developing online holiday property reservation systems.

Jeremy McPeak began tinkering with web development as a hobby in 1998. Currently working in the IT department of a school district, Jeremy has experience developing web solutions with JavaScript, PHP, and C#. He has written several online articles covering topics such as XSLT, WebForms, and C#. He is also co-author of *Professional Ajax*. Jeremy can be reached through his web site at www.wdonline.com.

Acknowledgments

Firstly a big thank-you for her support to my partner Beci, who, now that the book's finished, will get to see me for more than 10 minutes a week.

I'd also like to say a very big thank you to Tom Dinse, who has been a great editor to work with and has done amazing work on the book.

Thanks also to Jim Minatel for making this book happen, and also his support in what has for me been yet another challenging and difficult year.

Many thanks to everyone who's supported and encouraged me over my many years of writing books. Your help will always be remembered.

Finally, pats and treats to my German shepherd Katie, who does an excellent job of warding off disturbances from door-to-door salespeople.

—Paul Wilton

Credits

Senior Acquisitions Editor

Jim Minatel

Senior Development Editor

Tom Dinse

Technical Editor

Alexei Gorkov

Production Editor

Travis Henderson

Copy Editor

S. B. Kleinman

Editorial Manager

Mary Beth Wakefield

Production Manager

Tim Tate

Vice President and Executive Group Publisher

Richard Swadley

Vice President and Executive Publisher

Joseph B. Wikert

Graphics and Production Specialists

Joyce Haughey

Jennifer Mayberry

Alicia B. South

Quality Control Technician

Melanie Hoffman

Project Coordinator

Erin Smith

Proofreading and Indexing

Laura L. Bowman

Slivoskey Indexing Services

Anniversary Logo Design

Richard Pacifico

Contents

Acknowledgments	ix
Introduction	xix
Chapter 1: Introduction to JavaScript and the Web	1
Introduction to JavaScript	1
What Is JavaScript?	1
JavaScript and the Web	2
Why Choose JavaScript?	3
What Can JavaScript Do for Me?	4
Tools Needed to Create JavaScript Web Applications	4
The <script> Tag and Your First Simple JavaScript Program	6
A Brief Look at Browsers and Compatibility Problems	12
Introducing the “Who Wants To Be A Billionaire” Trivia Quiz	13
Ideas Behind the Coding of the Trivia Quiz	16
What Functionality to Add and Where?	18
Summary	18
Chapter 2: Data Types and Variables	21
Types of Data in JavaScript	21
Numerical Data	22
Text Data	22
Boolean Data	24
Variables — Storing Data in Memory	24
Declaring Variables and Giving Them Values	25
Assigning Variables with the Value of Other Variables	28
Setting Up Your Browser for Errors	30
Displaying Errors in Firefox	30
Displaying Errors in Internet Explorer	31
What Happens When You Get an Error	32
Using Data — Calculations and Basic String Manipulation	35
Numerical Calculations	35
Operator Precedence	39
Basic String Operations	42
Mixing Numbers and Strings	44
Data Type Conversion	45
Dealing with Strings That Won’t Convert	48

Contents

Arrays	49
A Multi-Dimensional Array	54
The “Who Wants To Be A Billionaire” Trivia Quiz — Storing the Questions Using Arrays	58
Summary	61
Exercise Questions	62
Question 1	62
Question 2	62
 Chapter 3: Decisions, Loops, and Functions	 63
 Decision Making — The if and switch Statements	 64
Comparison Operators	64
The if Statement	66
Logical Operators	70
Multiple Conditions Inside an if Statement	73
else and else if	77
Comparing Strings	78
The switch Statement	79
Looping — The for and while Statements	84
The for Loop	84
The for...in Loop	87
The while Loop	88
The do...while loop	90
The break and continue Statements	91
Functions	92
Creating Your Own Functions	92
Variable Scope and Lifetime	96
The Trivia Quiz — Building One of the Basic Functions	97
Summary	100
Exercise Questions	102
Question 1	102
Question 2	102
Question 3	103
Question 4	103
 Chapter 4: JavaScript — An Object-Based Language	 105
 Object-Based Programming	 105
A Brief Introduction to Objects	105
Objects in JavaScript	106
Using JavaScript Objects	107
Primitives and Objects	110

The JavaScript Native Objects	111
String Objects	111
The Math Object	122
Number Object	129
Array Objects	130
Date Objects	137
JavaScript Classes	146
Summary	156
Exercise Questions	157
Question 1	157
Question 2	157
Question 3	157
 Chapter 5: Programming the Browser	 159
Introduction to the Browser Objects	160
The window Object — Our Window onto the Page	161
The history Object	163
The location Object	163
The navigator Object	164
The screen Object	164
The document Object — The Page Itself	165
Connecting Code to Web Page Events	169
Browser Version Checking Examples	175
Summary	185
Exercise Questions	186
Question 1	186
Question 2	186
Question 3	186
 Chapter 6: HTML Forms — Interacting with the User	 187
HTML Forms	188
Other Form Object Properties and Methods	190
HTML Elements in Forms	191
Common Properties and Methods	193
Button Form Elements	194
Text Elements	197
textarea Element	204
Check Boxes and Radio Buttons	207
The select Elements	215

Contents

The Trivia Quiz	227
Creating the Form	228
Creating the Answer Radio Buttons	229
Summary	233
Exercises	235
Question 1	235
Question2	235
 Chapter 7: Windows and Frames	 237
 Frames and the window Object	 238
Coding Between Frames	241
Code Access Between Frames	247
Opening New Windows	257
Opening a New Browser Window	257
Scripting Between Windows	264
Moving and Resizing Windows	269
Security	270
Trivia Quiz	271
Creating the New Trivia Quiz	272
Summary	287
Exercise Questions	289
Question 1	289
Question 2	289
 Chapter 8: String Manipulation	 291
 Additional String Methods	 292
The split() Method	292
The replace() Method	295
The search() Method	296
The match() Method	296
Regular Expressions	297
Simple Regular Expressions	297
Regular Expressions: Special Characters	300
Covering All Eventualities	308
Grouping Regular Expressions	308
The String Object — split(), replace(), search(), and match() Methods	312
The split() Method	312
The replace() Method	314
The search() Method	318
The match() Method	318
Using the RegExp Object's Constructor	321

The Trivia Quiz	322
Summary	329
Exercise Questions	330
Question 1	330
Question 2	331
Question 3	331
 Chapter 9: Date, Time, and Timers	 333
World Time	334
Setting and Getting a Date Object's UTC Date and Time	339
Timers in a Web Page	347
One-Shot Timer	348
Setting a Timer that Fires at Regular Intervals	352
The Trivia Quiz	354
Summary	360
Exercise Questions	360
Question 1	360
Question 2	360
 Chapter 10: Common Mistakes, Debugging, and Error Handling	 361
I Can't Believe I Just Did That: Some Common Mistakes	361
1: Undefined Variables	362
2: Case Sensitivity	363
3: Incorrect Number of Closing Braces	364
4: Missing Plus Signs During Concatenation	364
5: Equals Rather than Is Equal To	365
6: Incorrect Number of Closing Parentheses	365
7: Using a Method as a Property and Vice Versa	366
Microsoft Script Debugger	367
Obtaining the Script Debugger	367
Installing the Script Debugger	368
Using the Script Debugger	370
Firefox Debugging with Venkman	388
Error Handling	393
Preventing Errors	393
The try...catch Statements	394
Summary	404
Exercise Questions	405
Question 1	405
Question 2	405

Contents

Chapter 11: Storing Information: Cookies **407**

Baking Your First Cookie	407
A Fresh-Baked Cookie	408
The Cookie String	415
Creating a Cookie	418
Getting a Cookie's Value	422
Cookie Limitations	428
Cookie Security and IE6 and IE7	429
Summary	434
Exercise Questions	435
Question 1	435
Question 2	435

Chapter 12: Introduction to Dynamic HTML **437**

Cross-Browser Issues	437
Events	438
CSS: A Primer	458
Adding Some Style to HTML	459
Dynamic HTML	470
Accessing Elements	470
Changing Appearances	470
Positioning and Moving Content	478
Example: Animated Advertisement	483
Summary	488
Exercise Questions	488
Question 1	488
Question 2	488

Chapter 13: Dynamic HTML in Modern Browsers **489**

Why Do We Need Web Standards?	490
The Web Standards	492
HTML	492
ECMAScript	492
XML	493
XHTML	494
The Document Object Model	495
The DOM Standard	496
Differences Between the DOM and the BOM	497
Representing the HTML Document as a Tree Structure	497
The DOM Objects	501

DOM Properties and Methods	502
The DOM Event Model	521
DHTML Example: Internet Explorer 5+	526
The IE Event Model	526
Building a DHTML Toolbar	528
DHTML Example: The Toolbar in Firefox and Opera	540
Creating Cross-Browser DHTML	544
Summary	548
Exercise Questions	548
Question 1	548
Question 2	549
 Chapter 14: JavaScript and XML	 551
What Can XML Do for Me?	551
The Basics of XML	552
Understanding XML Syntax	552
Creating an XML Document	557
Document Type Definition	559
Creating the First DTD File	559
Bring on the Data	561
Altering the Look of XML	566
Style Sheets and XML	566
Extensible Stylesheet Language	568
Manipulating XML with JavaScript	576
Retrieving an XML File in IE	577
Determining When the XML File Has Loaded	579
Retrieving an XML File in Firefox and Opera	580
Determining When the File is Loaded	580
Cross-Browser XML File Retrieval	581
Displaying a Daily Message	582
Summary	592
Exercise Questions	593
Question 1	593
Question 2	593
 Chapter 15: Using ActiveX and Plug-Ins with JavaScript	 595
Checking for and Embedding Plug-ins in Firefox	596
Adding a Plug-in to the Page	596
Checking for and Installing Plug-ins in Firefox	598

Contents

Checking for and Embedding ActiveX Controls on Internet Explorer	601
Adding an ActiveX Control to the Page	602
Installing an ActiveX Control	606
Using Plug-ins and ActiveX Controls	607
Testing Your “No Plug-in or ActiveX Control” Redirection Script	614
Potential Problems	614
Summary	617
Exercise Question	618
Question 1	618
 Chapter 16: Ajax and Remote Scripting	 619
What Is Remote Scripting?	619
What Can It Do?	619
Ajax	621
Browser Support	621
Ajax with JavaScript: The XMLHttpRequest Object	622
Cross-Browser Issues	622
Using the XMLHttpRequest Object	626
Asynchronous Requests	627
Creating a Remote Scripting Class	629
The HttpRequest Constructor	629
Creating the Methods	631
The Full Code	632
Creating a Smarter Form with XMLHttpRequest	634
Requesting Information	634
The Received Data	635
Before You Begin	635
Creating a Smarter Form with an IFrame	644
The Server Response	645
Things to Watch Out For	652
The Same-Origin Policy	652
ActiveX	652
Usability Concerns	653
Summary	654
Exercise Questions	655
Question 1	655
Question 2	655
 Appendix A: Exercise Solutions	 657
 Index	 727

Introduction

JavaScript is a scripting language that enables you to enhance static web applications by providing dynamic, personalized, and interactive content. This improves the experience of visitors to your site and makes it more likely that they will visit again. You must have seen the flashy drop-down menus, moving text, and changing content that are now widespread on web sites—they are enabled through JavaScript. Supported by all the major browsers, JavaScript is the language of choice on the Web. It can even be used outside web applications—to automate administrative tasks, for example.

This book aims to teach you all you need to know to start experimenting with JavaScript: what it is, how it works, and what you can do with it. Starting from the basic syntax, you'll move on to learn how to create powerful web applications. Don't worry if you've never programmed before—this book will teach you all you need to know, step by step. You'll find that JavaScript can be a great introduction to the world of programming; with the knowledge and understanding that you'll gain from this book, you'll be able to move on to learn newer and more advanced technologies in the world of computing.

Whom This Book Is For

In order to get the most out of this book, you'll need to have an understanding of HTML and how to create a static web page. You don't need to have any programming experience.

This book will also suit you if you have some programming experience already, and would like to turn your hand to web programming. You will know a fair amount about computing concepts, but maybe not as much about web technologies.

Alternatively, you may have a design background and know relatively little about the Web and computing concepts. For you, JavaScript will be a cheap and relatively easy introduction to the world of programming and web application development.

Whoever you are, we hope that this book lives up to your expectations.

What This Book Covers

You'll begin by looking at exactly what JavaScript is, and taking your first steps with the underlying language and syntax. You'll learn all the fundamental programming concepts, including data and data types, and structuring your code to make decisions in your programs or to loop over the same piece of code many times.

Once you're comfortable with the basics, you'll move on to one of the key ideas in JavaScript—the object. You'll learn how to take advantage of the objects that are native to the JavaScript language, such as dates and strings, and find out how these objects enable you to manage complex data and simplify

Introduction

your programs. Next, you'll see how you can use JavaScript to manipulate objects made available to you in the browser, such as forms, windows, and other controls. Using this knowledge, you can start to create truly professional-looking applications that enable you to interact with the user.

Long pieces of code are very hard to get right every time—even for the experienced programmer—and JavaScript code is no exception. You look at common syntax and logical errors, how you can spot them, and how to use the Microsoft Script Debugger to aid you with this task. Also, you need to examine how to handle the errors that slip through the net, and ensure that these do not detract from the experience of the end user of your application.

From here, you'll move on to more advanced topics, such as using cookies and jazzing up your web pages with dynamic HTML and XML. Finally, you'll be looking at a relatively new and exciting technology, remote scripting. This allows your JavaScript in a HTML page to communicate directly with a server, and useful for, say, looking up information on a database sitting on your server. If you have the Google toolbar you'll have seen something like this in action already. When you type a search word in the Google toolbar, it comes up with suggestions, which it gets via the Google search database.

All the new concepts introduced in this book will be illustrated with practical examples, which enable you to experiment with JavaScript and build on the theory that you have just learned. The appendix provides solutions to the exercises included at the end of most chapters throughout the book.

During the first half of the book, you'll also be building up a more complex sample application—an online trivia quiz—which will show you how JavaScript is used in action in a real-world situation.

What You Need to Use This Book

Because JavaScript is a text-based technology, all you really need to create documents containing JavaScript is Notepad (or your equivalent text editor).

Also, in order to try out the code in this book, you will need a web browser that supports a modern version of JavaScript. Ideally, this means Internet Explorer 6 or later and Firefox 1.5 or later. The book has been extensively tested with these two browsers. However, the code should work in most modern web browsers, although some of the code in later chapters, where you examine dynamic HTML and scripting the DOM, is specific to particular browsers; but the majority of the code presented is cross-browser. Where there are exceptions, they will be clearly noted.

Conventions

To help you get the most from the text and keep track of what's happening, we've used a number of conventions throughout the book.

Try It Out

The *Try It Out* is an exercise you should work through, following the text in the book.

1. They usually consist of a set of steps.
2. Each step has a number.
3. Follow the steps with your copy of the database.

How It Works

After each *Try It Out*, the code you've typed will be explained in detail.

Boxes like this one hold important, not-to-be forgotten information that is directly relevant to the surrounding text.

Tips, hints, tricks, and asides to the current discussion are offset and placed in italics like this.

As for styles in the text:

- ☐ We *highlight in italic type* new terms and important words when we introduce them.
- ☐ We show keyboard strokes like this: Ctrl+A.
- ☐ We show file names, URLs, and code within the text like so: `persistence.properties`.
- ☐ We present code in two different ways:

In code examples we highlight new and important code with a gray background.

The gray highlighting is not used for code that's less important in the present context, or that has been shown before.

Source Code

As you work through the examples in this book, you may choose either to type in all the code manually or to use the source-code files that accompany the book. All of the source code used in this book is available for download at www.wrox.com. Once at the site, simply locate the book's title (either by using the Search box or by using one of the title lists) and click the Download Code link on the book's detail page to obtain all the source code for the book.

Because many books have similar titles, you may find it easiest to search by ISBN; this book's ISBN is 978-0-470-05151-1.

Once you download the code, just decompress it with your favorite compression tool. Alternately, you can go to the main Wrox code download page at www.wrox.com/dynamic/books/download.aspx to see the code available for this book and all other Wrox books.

Errata

We make every effort to ensure that there are no errors in the text or in the code. However, no one is perfect, and mistakes do occur. If you find an error in one of our books, like a spelling mistake or faulty piece of code, we would be very grateful for your feedback. By sending in errata you may save another reader hours of frustration, and at the same time you will be helping us provide even higher-quality information.

To find the errata page for this book, go to www.wrox.com and locate the title using the Search box or one of the title lists. Then, on the book details page, click the Book Errata link. On this page you can view all errata that have been submitted for this book and posted by Wrox editors. A complete book list, including links to each book's errata, is also available at www.wrox.com/misc-pages/booklist.shtml.

If you don't spot "your" error on the Book Errata page, go to www.wrox.com/contact/techsupport.shtml and complete the form there to send us the error you have found. We'll check the information and, if appropriate, post a message to the book's errata page and fix the problem in subsequent editions of the book.

p2p.wrox.com

For author and peer discussion, join the P2P forums at p2p.wrox.com. The forums are a Web-based system on which you can post messages relating to Wrox books and related technologies and interact with other readers and technology users. The forums offer a subscription feature to e-mail you topics of interest of your choosing when new posts are made to the forums. Wrox authors, editors, other industry experts, and your fellow readers are present on these forums.

At <http://p2p.wrox.com> you will find a number of different forums that will help you not only as you read this book, but also as you develop your own applications. To join the forums, just follow these steps:

1. Go to p2p.wrox.com and click the Register link.
2. Read the terms of use and click Agree.
3. Complete the required information to join as well as any optional information you wish to provide, and click Submit.
4. You will receive an e-mail with information describing how to verify your account and complete the joining process.

You can read messages in the forums without joining P2P, but in order to post your own messages, you must join.

Once you join, you can post new messages and respond to messages other users post. You can read messages at any time on the Web. If you would like to have new messages from a particular forum e-mailed to you, click the Subscribe to this Forum icon by the forum name in the forum listing.

For more information about how to use the Wrox P2P, be sure to read the P2P FAQs for answers to questions about how the forum software works, as well as many common questions specific to P2P and Wrox books. To read the FAQs, click the FAQ link on any P2P page.

1

Introduction to JavaScript and the Web

In this introductory chapter, you look at what JavaScript is, what it can do for you, and what you need in order to use it. With these foundations in place, you will see throughout the rest of the book how JavaScript can help you to create powerful web applications for your web site.

The easiest way to learn something is by actually doing it, so throughout the book you'll create a number of useful example programs using JavaScript. This process starts in this chapter, by the end of which you will have created your first piece of JavaScript code.

Additionally, over the course of the book you'll develop a complete JavaScript web application: an online trivia quiz. By seeing it develop, step by step, you'll get a good understanding of how to create your own web applications. At the end of this chapter, you'll look at the finished trivia quiz and consider the ideas behind its design.

Introduction to JavaScript

In this section you take a brief look at what JavaScript is, where it came from, how it works, and what sorts of useful things you can do with it.

What Is JavaScript?

Having bought this book, you are probably already well aware that JavaScript is some sort of *computer language*, but what is a computer language? Put simply, a computer language is a series of instructions that tell the computer to do something. That something can be one of a wide variety of things, including displaying text, moving an image, or asking the user for information. Normally the instructions, or what is termed *code*, are *processed* from the top line downward. This simply means that the computer looks at the code you've written, works out what action you want taken, and then takes that action. The actual act of processing the code is called *running* or *executing* it.

Chapter 1: Introduction to JavaScript and the Web

In natural English, here are instructions, or code, you might write to make a cup of instant coffee:

1. Put coffee crystals in cup.
2. Fill kettle with water.
3. Put kettle on to boil.
4. Has the kettle boiled? If so, then pour water into cup; otherwise, continue to wait.
5. Drink coffee.

You'd start running this code from the first line (instruction 1), and then continue to the next (instruction 2), then the next, and so on until you came to the end. This is pretty much how most computer languages work, JavaScript included. However, there are occasions when you might change the flow of execution, or even skip over some code, but you'll see more of this in Chapter 3.

JavaScript is an interpreted language, rather than a compiled language. What is meant by the terms *interpreted* and *compiled*?

Well, to let you in on a secret, your computer doesn't really understand JavaScript at all. It needs something to interpret the JavaScript code and convert it into something that it understands; hence it is an *interpreted language*. Computers understand only *machine code*, which is essentially a string of binary numbers (that is, a string of zeros and ones). As the browser goes through the JavaScript, it passes it to a special program called an *interpreter*, which converts the JavaScript to the machine code your computer understands. It's a bit like having a translator to translate English into Spanish, for example. The important point to note is that the conversion of the JavaScript happens at the time the code is run; it has to be repeated every time this happens. JavaScript is not the only interpreted language; there are others, including VBScript.

The alternative *compiled language* is one in which the program code is converted to machine code before it's actually run, and this conversion only has to be done once. The programmer uses a compiler to convert the code that he wrote to machine code, and it is this machine code that is run by the program's user. Compiled languages include Visual Basic and C++. Using a real-world analogy, it's like having someone translate your English document into Spanish. Unless you change the document, you can use it without retranslation as much as you like.

Perhaps this is a good point to dispel a widespread myth: JavaScript is not the script version of the Java language. In fact, although they share the same name, that's virtually all they do share. Particularly good news is that JavaScript is much, much easier to learn and use than Java. In fact, languages like JavaScript are the easiest of all languages to learn, but they are still surprisingly powerful.

JavaScript and the Web

For most of this book you'll look at JavaScript code that runs inside a web page loaded into a browser. All you need in order to create these web pages is a text editor — for example, Windows Notepad — and a web browser, such as Firefox or Internet Explorer, with which you can view your pages. These browsers come equipped with JavaScript interpreters.

In fact, the JavaScript language first became available in the web browser Netscape Navigator 2. Initially, it was called LiveScript. However, because Java was the hot technology of the time, Netscape decided

that JavaScript sounded more exciting. When JavaScript really took off, Microsoft decided to add its own brand of JavaScript, called JScript, to Internet Explorer. Since then, Netscape, Microsoft, and others have released improved versions and included them in their latest browsers. Although these different brands and versions of JavaScript have much in common, there are enough differences to cause problems if you're not careful. Initially you'll be creating code that'll work with most browsers, whether Firefox, Internet Explorer, or Netscape. Later chapters look at features available only to Firefox 1.5 or later and Internet Explorer 6 and 7. You'll look into the problems with different browsers and versions of JavaScript later in this chapter, and see how to deal with them.

The majority of the web pages containing JavaScript that you create in this book can be stored on your hard drive and loaded directly into your browser from the hard drive itself, just as you'd load any normal file (such as a text file). However, this is not how web pages are loaded when you browse web sites on the Internet. The Internet is really just one great big network connecting computers. Access to web sites is a special service provided by particular computers on the Internet; the computers providing this service are known as *web servers*.

Basically the job of a web server is to hold lots of web pages on its hard drive. When a browser, usually on a different computer, requests a web page contained on that web server, the web server loads it from its own hard drive and then passes the page back to the requesting computer via a special communications protocol called *Hypertext Transfer Protocol (HTTP)*. The computer running the web browser that makes the request is known as the *client*. Think of the client/server relationship as a bit like a customer/shopkeeper relationship. The customer goes into a shop and says, "Give me one of those." The shopkeeper serves the customer by reaching for the item requested and passing it back to the customer. In a web situation, the client machine running the web browser is like the customer, and the web server providing the page requested is like the shopkeeper.

When you type an address into the web browser, how does it know which web server to get the page from? Well, just as shops have addresses, say, 45 Central Avenue, Sometownsville, so do web servers. Web servers don't have street names; instead they have *Internet protocol (IP) addresses*, which uniquely identify them on the Internet. These consist of four sets of numbers, separated by dots; for example, 127.0.0.1.

If you've ever surfed the net, you're probably wondering what on earth I'm talking about. Surely web servers have nice `www.somewebsite.com` names, not IP addresses? In fact, the `www.somewebsite.com` name is the "friendly" name for the actual IP address; it's a whole lot easier for us humans to remember. On the Internet, the friendly name is converted to the actual IP address by computers called *domain name servers*, which your Internet service provider will have set up for you.

Toward the end of the book, you'll go through the process of setting up your own web server in a step-by-step guide. You'll see that web servers are not just dumb machines that pass pages back to clients, but in fact can do a bit of processing themselves using JavaScript. You'll look at this later in the book as well.

One last thing: Throughout this book we'll be referring to the Internet Explorer browser as IE.

Why Choose JavaScript?

JavaScript is not the only scripting language; there are others such as VBScript and Perl. So why choose JavaScript over the others?

Chapter 1: Introduction to JavaScript and the Web

The main reason for choosing JavaScript is its widespread use and availability. Both of the most commonly used browsers, IE and Firefox, support JavaScript, as do almost all of the less commonly used browsers. So you can assume that most people browsing your web site will have a version of JavaScript installed, though it is possible to use a browser's options to disable it.

Of the other scripting languages we mentioned, VBScript, which can be used for the same purposes as JavaScript, is supported only by Internet Explorer running on the Windows operating system, and Perl is not used at all in web browsers.

JavaScript is also very versatile and not just limited to use within a web page. For example, it can be used in Windows to automate computer-administration tasks and inside Adobe Acrobat .pdf files to control the display of the page just as in web pages, although Acrobat uses a more limited version of JavaScript. However, the question of which scripting language is the most powerful and useful has no real answer. Pretty much everything that can be done in JavaScript can be done in VBScript, and vice versa.

What Can JavaScript Do for Me?

The most common uses of JavaScript are interacting with users, getting information from them, and validating their actions. For example, say you want to put a drop-down menu on the page so that users can choose where they want to go to on your web site. The drop-down menu might be plain old HTML, but it needs JavaScript behind it to actually do something with the user's input. Other examples of using JavaScript for interactions are given by forms, which are used for getting information from the user. Again, these may be plain HTML, but you might want to check the validity of the information that the user is entering. For example, if you had a form taking a user's credit card details in preparation for the online purchase of goods, you'd want to make sure he had actually filled in those details before you sent the goods. You might also want to check that the data being entered are of the correct type, such as a number for his age rather than text.

JavaScript can also be used for various tricks. One example is switching an image in a page for a different one when the user rolls her mouse over it, something often seen in web page menus. Also, if you've ever seen scrolling messages in the browser's status bar (usually at the bottom of the browser window) or inside the page itself and wondered how that works, this is another JavaScript trick that you'll learn about later in the book. You'll also see how to create expanding menus that display a list of choices when a user rolls his or her mouse over them, another commonly seen JavaScript-driven trick.

Tricks are okay up to a point, but even more useful are small applications that provide a real service. For example, a mortgage seller's web site with a JavaScript-driven mortgage calculator, or a web site about financial planning that includes a calculator that works out your tax bill for you. With a little inventiveness, you'll be amazed at what can be achieved.

Tools Needed to Create JavaScript Web Applications

All that you need to get started with creating JavaScript code for web applications is a simple text editor, such as Windows Notepad, or one of the many slightly more advanced text editors that provide line numbering, search and replace, and so on. An alternative is a proper HTML editor; you'll need one that enables you to edit the HTML source code, because that's where you need to add your JavaScript. A

number of very good tools specifically aimed at developing web-based applications, such as the excellent Dreamweaver 8 from Adobe, are also available. However, this book concentrates on JavaScript, rather than any specific development tool. When it comes to learning the basics, it's often best to write the code by hand rather than relying on a tool to do it for you. This helps you to understand the fundamentals of the language before you attempt the more advanced logic that is beyond a tool's capability. When you have a good understanding of the basics, you can use tools as timesavers so that you can spend more time on the more advanced and more interesting coding.

You'll also need a browser to view your web pages in. It's best to develop your JavaScript code on the sort of browsers you expect visitors to use to access your web site. You'll see later in the chapter that there are different versions of JavaScript, each supported by different versions of the web browsers. Each of these JavaScript versions, while having a common core, also contains various extensions to the language. All the examples that we give in this book have been tested on Firefox version 1.5, and IE versions 6 and 7. Wherever a piece of code does not work on any of these browsers, a note to this effect has been made in the text.

If you're running Windows, you'll almost certainly have IE installed. If not, a trip to www.microsoft.com/windows/ie/default.msp will get you the latest version.

Firefox can be found at www.mozilla.com/firefox/all.html. When installing Firefox it's worth going for the custom setup. This will give you the option later on of choosing which bits to install. In particular it's worth selecting the Developer Tools component. While not essential, it's an extra that's nice to have.

Even if your browser supports JavaScript, it is possible to disable this functionality in the browser. So before you start on your first JavaScript examples in the next section, you should check to make sure JavaScript is enabled in your browser.

To do this in Firefox, choose Options from the Tools menu on the browser. In the window that appears, click the Content tab. From this tab make sure the Enable JavaScript check box is selected, as shown in Figure 1-1.

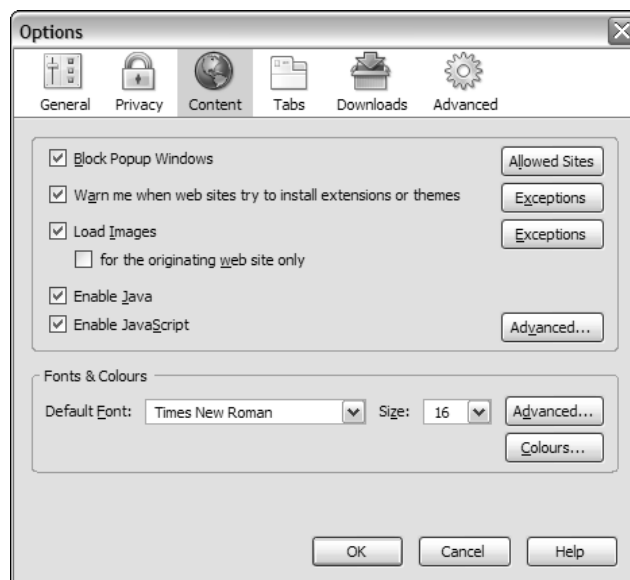


Figure 1-1

Chapter 1: Introduction to JavaScript and the Web

It is harder to turn off scripting in Internet Explorer. Choose Internet Options from the Tools menu on the browser, click the Security tab, and check whether the Internet or Local intranet options have custom security settings. If either of them does, click the Custom Level button and scroll down to the Scripting section. Check that Active Scripting is set to Enable.

A final point to note is how to open the code examples in your browser. For this book, you simply need to open the file on your hard drive in which an example is stored. You can do this in a number of ways. One way in IE6 is to choose Open from the File menu, and click the Browse button to browse to where you stored the code. Similarly, in Firefox choose Open File from the File menu, browse to the file you want, and click the Choose File button.

IE7, however, has a new menu structure and this doesn't include an Open File option. You can get around this by typing the drive letter of your hard drive followed by a colon in the address bar, for example, C: for your C drive. Alternatively you can switch back to the Classic menu of earlier versions of IE (see Figure 1-2). To do this you need to go to the Tools menu, select the Toolbars menu option, then ensure the Classic Menu option is selected.



Figure 1-2

The `<script>` Tag and Your First Simple JavaScript Program

Enough talk about the subject of JavaScript; it's time to look at how to put it into your web page. In this section, you write your first piece of JavaScript code.

Inserting JavaScript into a web page is much like inserting any other HTML content; you use tags to mark the start and end of your script code. The tag used to do this is the `<script>` tag. This tells the browser that the following chunk of text, bounded by the closing `</script>` tag, is not HTML to be displayed, but rather script code to be processed. The chunk of code surrounded by the `<script>` and `</script>` tags is called a *script block*.

Basically, when the browser spots `<script>` tags, instead of trying to display the contained text to the user, it uses the browser's built-in JavaScript interpreter to run the code's instructions. Of course, the code might give instructions about changes to the way the page is displayed or what is shown in the page, but the text of the code itself is never shown to the user.

You can put the `<script>` tags inside the header (between the `<head>` and `</head>` tags), or inside the body (between the `<body>` and `</body>` tags) of the HTML page. However, although you can put them outside these areas — for example, before the `<html>` tag or after the `</html>` tag — this is not permitted in the web standards and so is considered bad practice.

The `<script>` tag has a number of attributes, but the most important one is the `type` attribute. As you saw earlier, JavaScript is not the only scripting language available, and different scripting languages need to be processed in different ways. You need to tell the browser which scripting language to expect so that it knows how to process that language. Your opening script tag will look like this:

```
<script type="text/javascript">
```

Including the `type` attribute is good practice, but within a web page it can be left off. Browsers such as IE and Firefox use JavaScript as their default script language. This means that if the browser encounters a `<script>` tag with no `type` attribute set, it assumes that the script block is written in JavaScript. However, use of the `type` attribute is specified as mandatory by W3C (the World Wide Web Consortium), which sets the standards for HTML.

Okay, let's take a look at the first page containing JavaScript code.

Try It Out Painting the Document Red

This is a simple example of using JavaScript to change the background color of the browser. In your text editor (I'm using Windows Notepad), type the following:

```
<html>
<body BGCOLOR="WHITE">
<p>Paragraph 1</p>
<script type="text/javascript">
    document.bgColor = "RED";
</script>
</body>
</html>
```

Save the page as `ch1_examp1.htm` to a convenient place on your hard drive. Now load it into your web browser. You should see a red web page with the text `Paragraph 1` in the top-left corner. But wait — didn't you set the `<body>` tag's `BGCOLOR` attribute to white? Okay, let's look at what's going on here.

How It Works

The page is contained within `<html>` and `</html>` tags. This block contains a `<body>` element. When you define the opening `<body>` tag, you use HTML to set the page's background color to white.

```
<BODY BGCOLOR="WHITE">
```

Then you let the browser know that your next lines of code are JavaScript code by using the `<script>` start tag.

```
<script type="text/javascript">
```

Chapter 1: Introduction to JavaScript and the Web

Everything from here until the close tag, `</script>`, is JavaScript and is treated as such by the browser. Within this script block, you use JavaScript to set the document's background color to red.

```
document.bgColor = "RED";
```

What you might call the *page* is known as the *document* for the purpose of scripting in a web page. The document has lots of properties, including its background color, `bgColor`. You can reference properties of the document by writing `document`, followed by a dot, followed by the property name. Don't worry about the use of `document` at the moment; you look at it in depth later in the book.

Note that the preceding line of code is an example of a JavaScript *statement*. Every line of code between the `<script>` and `</script>` tags is called a statement, although some statements may run on to more than one line.

You'll also see that there's a semicolon (;) at the end of the line. You use a semicolon in JavaScript to indicate the end of a statement. In practice, JavaScript is very relaxed about the need for semicolons, and when you start a new line, JavaScript will usually be able to work out whether you mean to start a new line of code. However, for good coding practice, you should use a semicolon at the end of statements of code, and a single JavaScript statement should fit onto one line rather than continue on to two or more lines. Moreover, you'll find there are times when you must include a semicolon, which you'll come to later in the book.

Finally, to tell the browser to stop interpreting your text as JavaScript and start interpreting it as HTML, you use the script close tag:

```
</script>
```

You've now looked at how the code works, but you haven't looked at the order in which it works. When the browser loads in the web page, the browser goes through it, rendering it tag by tag from top to bottom of the page. This process is called *parsing*. The web browser starts at the top of the page and works its way down to the bottom of the page. The browser comes to the `<body>` tag first and sets the document's background to white. Then it continues parsing the page. When it comes to the JavaScript code, it is instructed to change the document's background to red.

Try It Out The Way Things Flow

Let's extend the previous example to demonstrate the parsing of a web page in action. Type the following into your text editor:

```
<html>
<body bgcolor="WHITE">
<p>Paragraph 1</p>
<script type="text/javascript">
  // Script block 1
  alert("First Script Block");
</script>
<p>Paragraph 2</p>
<script type="text/javascript">
  // Script block 2
```

```
document.bgColor = "RED";  
alert("Second Script Block");  
</script>  
<p>Paragraph 3</p>  
</body>  
</html>
```

Save the file to your hard drive as `ch1_examp2.htm` and then load it into your browser. When you load the page you should see the first paragraph, `Paragraph 1`, followed by a message box displayed by the first script block. The browser halts its parsing until you click the OK button. As you can see in Figure 1-3, the page background is white, as set in the `<body>` tag, and only the first paragraph is currently displayed.

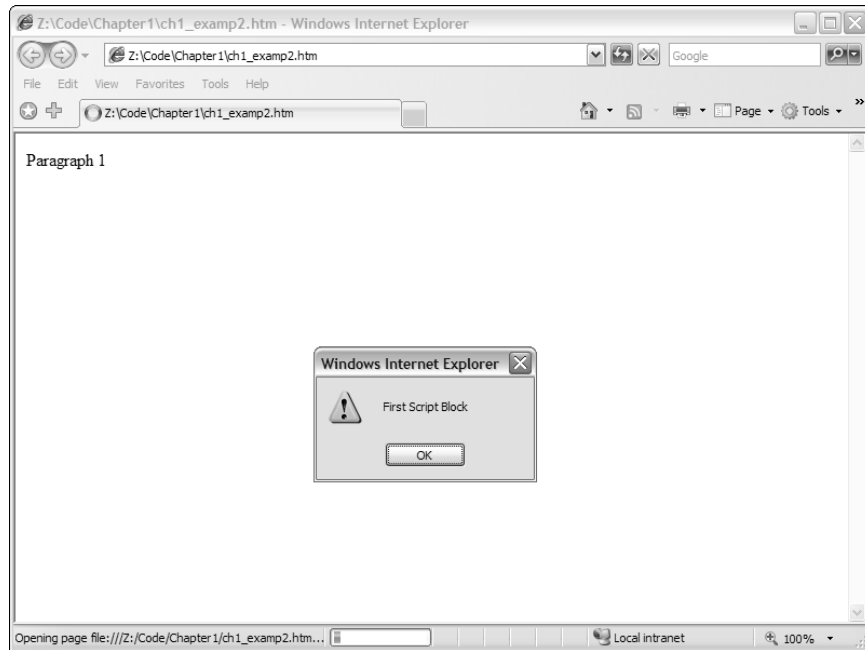


Figure 1-3

Click the OK button, and the parsing continues. The browser displays the second paragraph, and the second script block is reached, which changes the background color to red. Another message box is displayed by the second script block, as shown in Figure 1-4.

Click OK, and again the parsing continues, with the third paragraph, `Paragraph 3`, being displayed. The web page is complete, as shown in Figure 1-5.

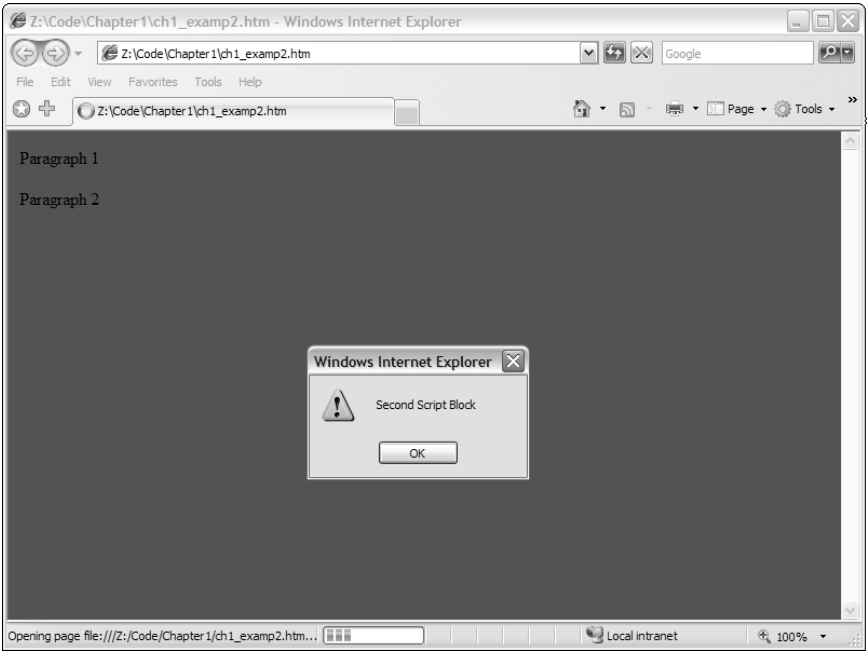


Figure 1-4

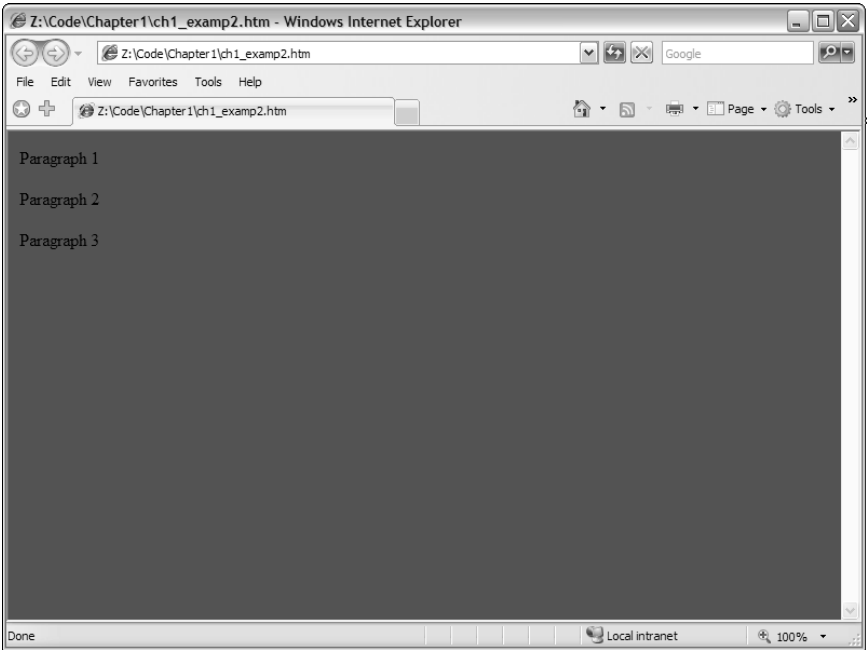


Figure 1-5

How It Works

The first part of the page is the same as in our earlier example. The background color for the page is set to white in the definition of the `<body>` tag, and then a paragraph is written to the page.

```
<html>
<body BGCOLOR="WHITE">
<p>Paragraph 1</p>
```

The first new section is contained in the first script block.

```
<script type="text/javascript">
  // Script block 1
  alert("First Script Block");
</script>
```

This script block contains two lines, both of which are new to you. The first line —

```
// Script block 1
```

is just a *comment*, solely for your benefit. The browser recognizes anything on a line after a double forward slash (`//`) to be a comment and does not do anything with it. It is useful for you as a programmer because you can add explanations to your code that make it easier to remember what you were doing when you come back to your code later.

The `alert()` function in the second line of code is also new to you. Before learning what it does, you need to know what a *function* is.

Functions are defined more fully in Chapter 3, but for now you need only to think of them as pieces of JavaScript code that you can use to do certain tasks. If you have a background in math, you may already have some idea of what a function is: A function takes some information, processes it, and gives you a result. A function makes life easier for you as a programmer because you don't have to think about how the function does the task — you can just concentrate on when you want the task done.

In particular, the `alert()` function enables you to alert or inform the user about something by displaying a message box. The message to be given in the message box is specified inside the parentheses of the `alert()` function and is known as the function's *parameter*.

The message box displayed by the `alert()` function is *modal*. This is an important concept, which you'll come across again. It simply means that the message box won't go away until the user closes it by clicking the OK button. In fact, parsing of the page stops at the line where the `alert()` function is used and doesn't restart until the user closes the message box. This is quite useful for this example, because it enables you to demonstrate the results of what has been parsed so far: The page color has been set to white, and the first paragraph has been displayed.

When you click OK, the browser carries on parsing down the page through the following lines:

```
<p>Paragraph 2</p>
<script type="text/javascript">
  // Script block 2
  document.bgColor = "RED";
  alert("Second Script Block");
</script>
```

Chapter 1: Introduction to JavaScript and the Web

The second paragraph is displayed, and the second block of JavaScript is run. The first line of the script block code is another comment, so the browser ignores this. You saw the second line of the script code in the previous example—it changes the background color of the page to red. The third line of code is the `alert()` function, which displays the second message box. Parsing is brought to a halt until you close the message box by clicking OK.

When you close the message box, the browser moves on to the next lines of code in the page, displaying the third paragraph and finally ending the web page.

```
<p>Paragraph 3</p>
</body>
</html>
```

Another important point raised by this example is the difference between setting properties of the page, such as background color, via HTML and doing the same thing using JavaScript. The method of setting properties using HTML is *static*: A value can be set only once and never changed again by means of HTML. Setting properties using JavaScript enables you to dynamically change their values. By the term *dynamic*, we are referring to something that can be changed and whose value or appearance is not set in stone.

This example is just that, an example. In practice if you wanted the page's background to be red, you would set the `<body>` tag's `BGColor` attribute to "RED", and not use JavaScript at all. Where you would want to use JavaScript is where you want to add some sort of intelligence or logic to the page. For example, if the user's screen resolution is particularly low, you might want to change what's displayed on the page; with JavaScript, you can do this. Another reason for using JavaScript to change properties might be for special effects—for example, making a page fade in from white to its final color.

A Brief Look at Browsers and Compatibility Problems

You've seen in the preceding example that by using JavaScript you can change a web page's document background color using the `bgColor` property of the document. The example worked whether you used a Netscape or Microsoft browser, because both types of browsers support a document with a `bgColor` property. You can say that the example is *cross-browser compatible*. However, it's not always the case that the property or language feature available in one browser will be available in another browser. This is even sometimes the case between versions of the same browser.

The version numbers for Internet Explorer and Firefox browsers are usually written as a decimal number; for example, Firefox has a version 1.5. In this book we use the following terminology to refer to these versions. By version 1.x we mean all versions starting with the number 1. By version 1.0+ we mean all versions with a number greater than or equal to 1.

One of the main headaches involved in creating web-based JavaScript is the differences between different web browsers, the level of HTML they support, and the functionality their JavaScript interpreters can handle. You'll find that in one browser you can move an image using just a couple of lines of code, but that in another it'll take a whole page of code, or even prove impossible. One version of JavaScript will contain a method to change text to uppercase, and another won't. Each new release of IE or Firefox browsers sees new and exciting features added to their HTML and JavaScript support. The good news is that to a much greater extent than ever before, browser creators are complying with standards set by organizations such as the W3C. Also, with a little ingenuity, you can write JavaScript that will work with both IE and Firefox browsers.

Which browsers you want to support really comes down to the browsers you think the majority of your web site's visitors, that is, your *user base*, will be using. This book is aimed at both IE6 and later and Firefox 1.5 and later.

If you want your web site to be professional, you need to somehow deal with older browsers. You could make sure your code is backward compatible—that is, it only uses features available in older browsers. However, you may decide that it's simply not worth limiting yourself to the features of older browsers. In this case you need to make sure your pages degrade gracefully. In other words, make sure that although your pages won't work in older browsers, they will fail in a way that means the user is either never aware of the failure or is alerted to the fact that certain features on the web site are not compatible with his or her browser. The alternative to degrading gracefully is for your code to raise lots of error messages, cause strange results to be displayed on the page, and generally make you look like an idiot who doesn't know what you're doing!

So how do you make your web pages degrade gracefully? You can do this by using JavaScript to determine which browser the web page is running in after it has been partially or completely loaded. You can use this information to determine what scripts to run or even to redirect the user to another page written to make best use of her particular browser. In later chapters, you see how to find out what features the browser supports and take appropriate action so that your pages work acceptably on as many browsers as possible.

Introducing the “Who Wants To Be A Billionaire” Trivia Quiz

Over the course of the first nine chapters of this book, you'll be developing a full web-based application, namely a trivia quiz. The trivia quiz works with both Firefox and IE6+ web browsers, making full use of their JavaScript capabilities.

Let's take a look at what the quiz will finally look like. The main starting screen is shown in Figure 1-6. Here the user can choose a time limit. Using a JavaScript-based timer, you keep track of how much time has elapsed.

Chapter 1: Introduction to JavaScript and the Web

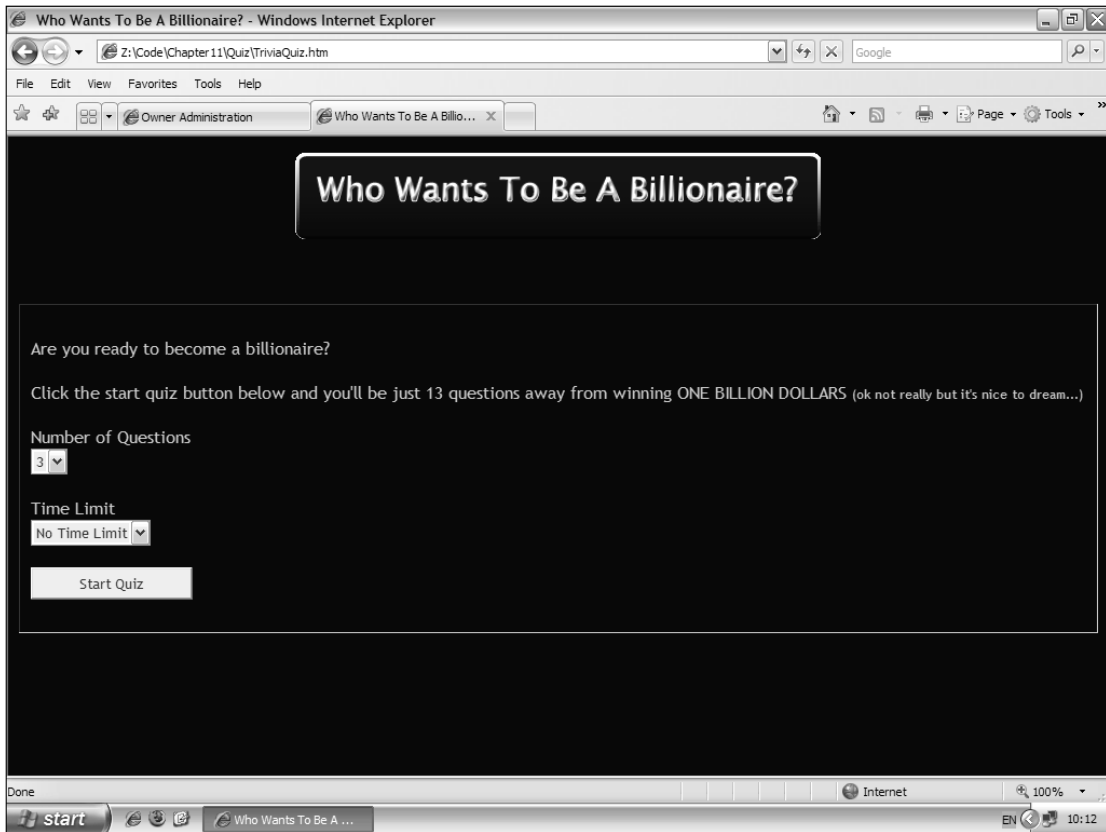


Figure 1-6

After clicking the Start Quiz button, the user is faced with a random choice of questions pulled from a database that you'll create to hold the trivia questions. There are two types of questions. The first, as shown in Figure 1-7, is the multiple-choice question. There is no limit to the number of answer options that you can specify for these types of questions: JavaScript handles them without the need for each question to be programmed differently.

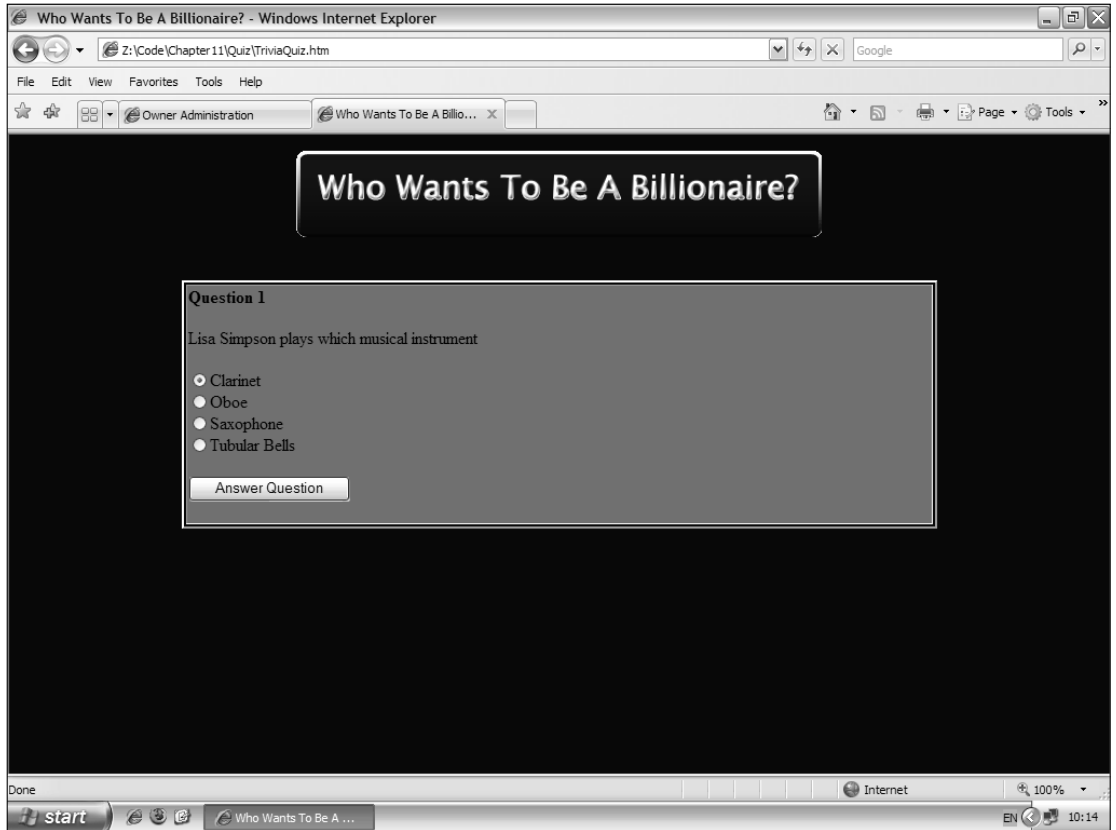


Figure 1-7

The second question style is a text-based one. The user types the answer into the text box provided, and then JavaScript does its best to intelligently interpret what the user has written. For example, for the question shown in Figure 1-8, Saxaphone is the correct answer. However, the JavaScript has been programmed to also accept the abbreviated form sax as a correct answer. You find out how to do this in Chapter 8.



Figure 1-8

Finally, after the questions have all been answered, the final page of the quiz displays how much the user has won.

Ideas Behind the Coding of the Trivia Quiz

You've taken a brief look at the final version of the trivia quiz in action and will be looking at the actual code in later chapters, but it's worthwhile to consider the guiding principles behind its design and programming.

One of the most important ideas is code reuse. You save time and effort by making use of the same code again and again. Quite often in a web application you'll find that you need to do the same thing over and over again. For example, you'll need to make repeated use of the code that checks whether a question has been answered correctly. You could make as many copies of the code as you need, and add this code to your page wherever you need it. However, this makes maintaining the code difficult, because if you need to correct an error or add a new feature, you will need to make the change to the code in lots of different places. Once the code for a web application grows from a few lines in one page to many lines over a number of pages, it's quite difficult to actually keep track of the places where you have copied the code. So, with reuse in mind, the trivia quiz keeps in one place all the important code that will need to be used a number of times.

The same goes for any data you use. For example, in the trivia quiz you keep track of the number of questions that have been answered, and update this information in as few places as possible.

Sometimes you have no choice but to put important code in more than one place — for example, when you need information that can only be obtained in a particular circumstance. However, if you can keep it in one place, you'll find doing so makes coding more efficient.

In the trivia quiz, I've also tried to split the code into specific *functions*. You will be looking at JavaScript functions in detail in Chapter 3. In the trivia quiz, the function that provides a randomly selected question for your web page to display is in one place, regardless of whether this is a multiple-choice question or a purely text-based question. By doing this, you're not only writing code just once, you're also making life easier for yourself by keeping code that provides the same service or function in one place. As you'll see later in the book, the code for creating these different question types is very different, but at least putting it in the same logical place makes it easy to find.

When creating your own web-based applications, you might find it useful to break the larger concept, here a trivia quiz, into smaller ideas. Breaking it down makes writing the code a lot easier. Rather than sitting down with a blank screen and thinking, "Right, now I must write a trivia quiz," you can think, "Right, now I must write some code to create a question." We find this technique makes coding a lot less scary and easier to get started on. This method of splitting the requirements of a piece of code into smaller and more manageable parts is often referred to as "divide and conquer."

Let's use the trivia quiz as an example. The trivia quiz application needs to do the following things:

- ☐ Ask a question.
- ☐ Retrieve and check the answer provided by the user to see if it's correct.
- ☐ Keep track of how many questions have been asked.
- ☐ Keep track of how many questions the user has answered correctly.
- ☐ If it's a timed quiz, keep track of the time remaining, and stop the quiz when the time is up.
- ☐ Show a final summary of the number of correct answers given out of the number answered.

These are the core ingredients for the trivia quiz. You may want to do other things, such as keep track of the number of user visits, but these are really external to the functionality of the trivia quiz.

After you've broken the whole concept into various logical areas, it's sometimes worth using the divide-and-conquer technique again to break the areas down into even smaller chunks, particularly if an area is quite complex or involved. As an example, let's take the first item from the preceding list.

Asking a question will involve the following:

- ☐ Retrieving the question data from where they're stored, for example from a database.
- ☐ Processing the data and converting them into a form that can be presented to the user. Here you need to create HTML to be displayed in a web page. How the data are processed depends on the question style: multiple choice or text based.
- ☐ Displaying the question for the user to answer.

Chapter 1: Introduction to JavaScript and the Web

As you build up the trivia quiz over the course of the book, you'll look at its design and some of the tricks and tactics that go into that design in more depth. You'll also break down each function as you come to it, to make it clear what needs to be done.

What Functionality to Add and Where?

How do you build up the functionality needed in the trivia quiz? The following list should give you an idea of what you add and in which chapter.

In Chapter 2, you start the quiz off by defining the multiple-choice questions that will be asked. You do this using something called an *array*, which is also introduced in that chapter.

In Chapter 3, where you learn about functions in more detail, you add a function to the code that will check to see whether the user has entered the correct answer.

After a couple of chapters of theory, in Chapter 6 you get the quiz into its first “usable” state. You display the questions to the user, and allow the user to answer those questions.

In Chapter 7, you enhance the quiz by turning it into what is called a *multi-frame application*. You add a button that the user can click to start the quiz, and specify that the quiz must finish after all the questions have been asked, repeating them indefinitely.

In Chapter 8, you add the text-based questions to the quiz. These must be treated slightly differently from multiple-choice questions, both in how they are displayed to the user and in how the user's answers are checked. As you saw earlier, the quiz will accept a number of different correct answers for these questions.

In Chapter 9, you allow the user to choose whether he or she wants to have a time limit for the quiz. If users choose to impose a time limit upon themselves, you count down the time in the status bar of the window and inform them when their time is up.

In Chapter 11, you complete the quiz by storing information about the user's previous results, using *cookies*, which are introduced in that chapter. This enables us to give the user a running average score at the end of the quiz.

Summary

At this point you should have a feel for what JavaScript is and what it can do. In particular, this brief introduction covered the following:

- ❑ You looked into the process the browser follows when interpreting your web page. It goes through the page element by element (parsing), and acts upon your HTML tags and JavaScript code as it comes to them.
- ❑ When you are developing for the web using JavaScript, you can choose to have your code executed in one of two places: server-side or client-side. Client-side is essentially the side on which the browser is running—the user's machine. Server-side refers to any processing or storage done on the web server itself.

- ❑ Unlike many programming languages, JavaScript requires just a text editor to start creating code. Something like Windows Notepad is fine for getting started, though more extensive tools will prove valuable once you get more experience.
- ❑ JavaScript code is embedded into the web page itself along with the HTML. Its existence is marked out by the use of `<script>` tags. As with HTML, script executes from the top of the page and works down to the bottom, interpreting and executing the code statement by statement.
- ❑ You were introduced to the online trivia quiz, which is the case study that you'll be building over the course of the book. You took a look at some of the design ideas behind the trivia quiz's coding, and learned how the functionality of the quiz will be built up over the course of the book.

2

Data Types and Variables

One of the main uses of computers is to process and display information. By processing, we mean that the information is modified, interpreted, or filtered in some way by the computer. For example, on an online banking web site, a customer may request details of all moneys paid out from his account in the last month. Here the computer would retrieve the information, filter out any information not related to payments made in the last month, and then display what's left in a web page. In some situations, information is processed without being displayed, and at other times, information is obtained directly without being processed. For example, in a banking environment, regular payments may be processed and transferred electronically without any human interaction or display.

In computing we refer to information as *data*. Data come in all sorts of forms, such as numbers, text, dates, and times, to mention just a few. In this chapter, you look specifically at how JavaScript handles data such as numbers and text. An understanding of how data are handled is fundamental to any programming language.

The chapter starts by looking at the various types of data JavaScript can process. Then you look at how you can store these data in the computer's memory so you can use them again and again in the code. Finally, you see how to use JavaScript to manipulate and process the data.

Types of Data in JavaScript

Data can come in many different forms, or *types*. You'll recognize some of the data types that JavaScript handles from the world outside programming — for example, numbers and text. Other data types are a little more abstract and are used to make programming easier; one example is the object data type, which you won't see in detail until Chapter 4.

Some programming languages are strongly typed languages. In these languages, whenever you use a piece of data, you need to explicitly state what sort of data you are dealing with, and use of those data must follow strict rules applicable to its type. For example, you can't add a number and a word together.

Chapter 2: Data Types and Variables

JavaScript, on the other hand, is a weakly typed language and a lot more forgiving about how you use different types of data. When you deal with data, you often don't need to specify type; JavaScript will work that out for itself. Furthermore, when you are using different types of data at the same time, JavaScript will work out behind the scenes what it is you're trying to do.

Given how easygoing JavaScript is about data, why talk about data types at all? Why not just cut to the chase and start using data without worrying about their type?

First of all, while JavaScript is very good at working out what data it's dealing with, there are occasions when it'll get things wrong, or at least not do what you want it to do. In these situations, you need to make it explicit to JavaScript what sort of data type you intended and how it should be used. To do that, you first need to know a little bit about data types.

A second reason is that data types enable you to use data effectively in your code. The things that can be done with data and the results you'll get depend on the type of data being used, even if you don't specify explicitly what type it is. For example, although trying to multiply two numbers together makes sense, doing the same thing with text doesn't. Also, the result of adding numbers is very different from the result of adding text. With numbers you get the sum, but with text you get one big piece of text consisting of the other pieces joined together.

Let's take a brief look at some of the more commonly used data types: numerical, text, and Boolean. You will see how to use them later in the chapter.

Numerical Data

Numerical data come in two forms:

- ❑ Whole numbers, such as 145, which are also known as *integers*. These numbers can be positive or negative and can span a very wide range in JavaScript: -2^{53} to 2^{53} .
- ❑ Fractional numbers, such as 1.234, which are also known as *floating-point* numbers. Like integers, they can be positive or negative, and they also have a massive range.

In simple terms, unless you're writing specialized scientific applications, you're not going to face problems with the size of numbers available in JavaScript. Also, although you can treat integers and floating-point numbers differently when it comes to storing them, JavaScript actually treats them both as floating-point numbers. It kindly hides the detail from you so you generally don't need to worry about it. One exception is when you want an integer but you have a floating-point number, in which case you'll round the number to make it an integer. You'll take a look at rounding numbers later in this chapter.

Text Data

Another term for one or more characters of text is a *string*. You tell JavaScript that text is to be treated as text and not as code simply by enclosing it inside quote marks (`"`). For example, `"Hello World"` and `"A"` are examples of strings that JavaScript will recognize. You can also use the single quote marks (`'`), so `'Hello World'` and `'A'` are also examples of strings that JavaScript will recognize. However, you must end the string with the same quote mark that you started it with. Therefore, `"A '` is not a valid JavaScript string, and neither is `'Hello World"`.

What if you want a string with a single quote mark in the middle, say a string like `Peter O'Toole`? If you enclose it in double quotes, you'll be fine, so `"Peter O'Toole"` is recognized by JavaScript. However, `'Peter O'Toole'` will produce an error. This is because JavaScript thinks that your text string is `Peter O` (that is, it treats the middle single quote as marking the end of the string) and falls over wondering what the `Toole'` is.

Another way around this is to tell JavaScript that the middle `'` is part of the text and is not indicating the end of the string. You do this by using the backslash character (`\`), which has special meaning in JavaScript and is referred to as an *escape character*. The backslash tells the browser that the next character is not the end of the string, but part of the text. So `'Peter O\'Toole'` will work as planned.

What if you want to use a double quote inside a string enclosed in double quotes? Well, everything just said about the single quote still applies. So `'Hello "Paul"'` works, but `"Hello "Paul" "` won't. However, `"Hello \"Paul\" "` will also work.

JavaScript has a lot of other special characters, which can't be typed in but can be represented using the escape character in conjunction with other characters to create *escape sequences*. These work much the same as in HTML. For example, more than one space in a row is ignored in HTML, so a space is represented by the term ` `. Similarly, in JavaScript there are instances where you can't use a character directly but must use an escape sequence. The following table details some of the more useful escape sequences.

Escape Sequences	Character Represented
<code>\b</code>	Backspace
<code>\f</code>	Form feed
<code>\n</code>	New line
<code>\r</code>	Carriage return
<code>\t</code>	Tab
<code>\'</code>	Single quote
<code>\"</code>	Double quote
<code>\\</code>	Backslash
<code>\xNN</code>	NN is a hexadecimal number that identifies a character in the Latin-1 character set.

The least obvious of these is the last, which represents individual characters by their character number in the Latin-1 character set rather than by their normal appearance. Let's pick an example: Say you wanted to include the copyright symbol (©) in our string. What would your string need to look like? The answer is `"\xA9 Paul Wilton"`.

Similarly, you can refer to characters using their Unicode escape sequence. These are written `\uNNNN`, where `NNNN` refers to the Unicode number for that particular character. For example, to refer to the copyright symbol using this method, you would use the string `\u00A9`.

Boolean Data

The use of yes or no, positive or negative, and true or false is commonplace in the physical world. The idea of true and false is also fundamental to digital computers; they don't understand maybes, only true and false. In fact, the concept of "yes or no" is so useful it has its own data type in JavaScript: the *Boolean* data type. The Boolean type has two possible values: `true` for yes and `false` for no.

The purpose of Boolean data in JavaScript is just the same as in the world outside programming: They enable us to answer questions and make decisions based on the answer. For example, if you are asked, "Is this book about JavaScript?" you would hopefully answer, "Yes it is," or you might also say, "That's true." Similarly you might say, "If it's false that the subject of the book is JavaScript, then put it down." Here you have a Boolean logic statement (named after its inventor George Boole), which asks a question and then does something based on whether the answer is true or false. In JavaScript, you can use the same sort of Boolean logic to give our programs decision-making abilities. You'll be taking a more detailed look at Boolean logic in the next chapter.

Variables — Storing Data in Memory

Data can be stored either permanently or temporarily.

You will want to keep important data, such as the details of a person's bank account, in a permanent store. For example, when Ms. Bloggs takes 10 dollars or pounds or euros out of her account, you want to deduct the money from her account and keep a permanent record of the new balance. Information like this might be stored in something called a *database*.

However, there are other cases where you don't want to permanently store data, but simply want to keep a temporary note of it. Let's look at an example. Say Ms. Bloggs has a loan from BigBadBank Inc., and she wants to find out how much is still outstanding on this loan. She goes to the online banking page for loans and clicks a link to find out how much she owes. This is data that will be stored permanently somewhere. However, suppose you also provide a facility for increasing loan repayments to pay off the loan early. If Ms. Bloggs enters an increased repayment amount into the text box on the web page, you might want to show how much sooner the loan will be paid. This will involve a few possibly complex calculations, so to make it easier, you want to write code that calculates the result in several stages, storing the result at each stage as you go along, before providing a final result. After you've done the calculation and displayed the results, there's no need to permanently store the results for each stage, so rather than use a database, you need to use something called a *variable*. Why is it called a variable? Well, perhaps because a variable can be used to store temporary data that can be altered, or varied.

Another bonus of variables is that unlike permanent storage, which might be saved to disk or magnetic tape, variables are held in the computer's memory. This means that it is much, much faster to store and retrieve the data.

So what makes variables good places for temporarily storing your data? Well, variables have a limited lifetime. When your visitors close the page or move to a new one, your variables are lost, unless you take some steps to save them somewhere.

Each variable is given a name so that you can refer to it elsewhere in your code. These names must follow certain rules.

As with much of JavaScript code, you'll find that variable names are case sensitive. For example, `myVariable` is not the same as `myvariable`. You'll find that this is a very easy way for errors to slip into your code, even when you become an expert at JavaScript.

Also, you can't use certain names and characters for your variable names. Names you can't use are called *reserved* words. Reserved words are words that JavaScript keeps for its own use, for example the word `var` or the word `with`. Certain characters are also forbidden in variable names; for example, the ampersand (&) and the percent sign (%). You are allowed to use numbers in your variable names, but the names must not begin with numbers. So `101myVariable` is not okay, but `myVariable101` is. Let's look at some more examples.

Invalid names include

- ❑ `with`
- ❑ `99variables`
- ❑ `my%Variable`
- ❑ `theGood&theBad`

Valid names include

- ❑ `myVariable99`
- ❑ `myPercent_Variable`
- ❑ `the_Good_and_the_Bad`

You may wish to use a naming convention for your variables; for example, one that describes what sort of data you plan to hold in the variable. You can notate your variables in lots of different ways—none are right or wrong, but it's best to stick with one of them. One common method is *Hungarian notation*, where the beginning of each variable name is a three-letter identifier indicating the data type. For example, you may start integer variable names with `int`, floating-point variable names with `flt`, string variable names with `str`, and so on. However, as long as the names you use make sense and are used consistently, it really doesn't matter what convention you choose.

Declaring Variables and Giving Them Values

Before you can use a variable, you should declare its existence to the computer using the `var` keyword. This warns the computer that it needs to reserve some memory for your data to be stored in later. To declare a new variable called `myFirstVariable`, you would write the following:

```
var myFirstVariable;
```

Note that the semicolon at the end of the line is not part of the variable name, but instead is used to indicate to JavaScript the end of a statement. This line is an example of a JavaScript statement.

Once declared, a variable can be used to store any type of data. As we mentioned earlier, many other programming languages, called strongly typed languages, require you to declare not only the variable but also the type of data, such as numbers or text, that will be stored. However, JavaScript is a weakly typed language; you don't need to limit yourself to what type of data a variable can hold.

Chapter 2: Data Types and Variables

You put data into your variables, a process called *assigning values* to your variables, by using the equals sign (=). For example, if you want your variable named `myFirstVariable` to hold the number 101, you would write this:

```
myFirstVariable = 101;
```

The equals sign has a special name when used to assign values to a variable; it's called the *assignment operator*.

Try It Out Declaring Variables

Let's look at an example in which a variable is declared, store some data in it, and finally access its contents. You'll also see that variables can hold any type of data, and that the type of data being held can be changed. For example, you can start by storing text and then change to storing numbers without JavaScript having any problems. Type the following code into your text editor and save it as `ch2_examp1.htm`:

```
<html>
<head>
</head>
<body>

<script language="JavaScript" type="text/javascript">

var myFirstVariable;

myFirstVariable = "Hello";
alert(myFirstVariable);

myFirstVariable = 54321;
alert(myFirstVariable);

</script>

</body>
</html>
```

As soon as you load this into your web browser, it should show an alert box with "Hello" in it, as shown in Figure 2-1. This is the content of the variable `myFirstVariable` at that point in the code.



Figure 2-1

Click OK and another alert box appears with 54321 in it, as shown in Figure 2-2. This is the new value you assigned to the variable `myFirstVariable`.



Figure 2-2

How It Works

Within the script block, you first declare your variable.

```
var myFirstVariable;
```

Currently, its value is the `undefined` value because you've declared only its existence to the computer, not any actual data. It may sound odd, but `undefined` is an actual primitive value in JavaScript, and it enables you to do comparisons. (For example, you can check to see if a variable contains an actual value or if it has not yet been given a value, that is, if it is `undefined`.) However, in the next line you assign `myFirstVariable` a string value, namely the value `Hello`.

```
myFirstVariable = "Hello";
```

Here you have assigned the variable a *literal* value, that is, a piece of actual data rather than data obtained by a calculation or from another variable. Almost anywhere that you can use a literal string or number, you can replace it with a variable containing number or string data. You see an example of this in the next line of code, where you use your variable `myFirstVariable` in the `alert()` function that you saw in the last chapter.

```
alert(myFirstVariable);
```

This causes the first alert box to appear. Next you store a new value in your variable, this time a number.

```
myFirstVariable = 54321;
```

The previous value of `myFirstVariable` is lost forever. The memory space used to store the value is freed up automatically by JavaScript in a process called *garbage collection*. Whenever JavaScript detects that the contents of a variable are no longer usable, such as when you allocate a new value, it performs the garbage collection process and makes the memory available. Without this automatic garbage collection process, more and more of the computer's memory would be consumed, until eventually the computer would run out and the system would grind to a halt. However, garbage collection is not always as efficient as it should be and may not occur until another page is loaded.

Just to prove that the new value has been stored, use the `alert()` function again to display the variable's new contents.

```
alert(myFirstVariable);
```

Assigning Variables with the Value of Other Variables

You've seen that you can assign a variable with a number or string, but can you assign a variable with the data stored inside another variable? The answer is yes, very easily, and in exactly the same way as giving a variable a literal value. For example, if you have declared the two variables `myVariable` and `myOtherVariable`, and have given the variable `myOtherVariable` the value 22, like this:

```
var myVariable;  
var myOtherVariable;  
myOtherVariable = 22;
```

then you can use the following line to assign `myVariable` the same value as `myOtherVariable` (that is, 22).

```
myVariable = myOtherVariable;
```

Try It Out Assigning Variables the Values of Other Variables

Let's look at another example, this time assigning variables the values of other variables. Type the following code into your text editor and save it as `ch2_examp2.htm`:

```
<html>  
<body>  
  
  <script language="JavaScript" type="text/javascript">  
  
    var string1 = "Hello";  
    var string2 = "Goodbye";  
  
    alert(string1);  
    alert(string2);  
  
    string2 = string1;  
  
    alert(string1);  
    alert(string2);  
  
    string1 = "Now for something different";  
  
    alert(string1);  
    alert(string2);  
  
  </script>  
  
</body>  
</html>
```

Load the page into your browser, and you'll see a series of six `alert` boxes appear. Click OK for each one to see the next. The first two show the values of `string1` and `string2` — Hello and Goodbye, respectively.

Then you assign `string2` the value that's in `string1`. The next two alert boxes show the contents of `string1` and `string2`; this time both are `Hello`.

Finally, you change the value of `string1`. Note that the value of `string2` remains unaffected. The final two alert boxes show the new value of `string1` (Now for something different) and the unchanged value of `string2` (`Hello`).

How It Works

The first thing you do in the script block is declare your two variables, `string1` and `string2`. However, notice that you have assigned them values at the same time that you have declared them. This is a shortcut, called *initializing*, that saves you typing too much code.

```
var string1 ="Hello";  
var string2 = "Goodbye";
```

Note that you can use this shortcut with all data types, not just strings. The next two lines show the current value of each variable to the user using the `alert()` function.

```
alert(string1);  
alert(string2);
```

Then you assign `string2` the value that's contained in `string1`. To prove that the assignment has really worked, you again show the user the contents of each variable using the `alert()` function.

```
string2 = string1;  
  
alert(string1);  
alert(string2);
```

Next, you set `string1` to a new value.

```
string1 = "Now for something different";
```

This leaves `string2` with its current value, demonstrating that `string2` has its own copy of the data assigned to it from `string1` in the previous step. You'll see in later chapters that this is not always the case. However, as a general rule, basic data types, such as text and numbers, are always copied when assigned, whereas more complex data types, like the objects you come across in Chapter 4, are actually shared and not copied. For example, if you have a variable with the string `Hello` and assign five other variables the value of this variable, you now have the original data and five independent copies of the data. However, if it was an object rather than a string and you did the same thing, you'd find you still have only one copy of the data, but that six variables share it. Changing the data using any of the six variable names would change them for all the variables.

Finally, the `alert()` function is used to show the current values of each variable.

```
alert(string1);  
alert(string2);
```

Setting Up Your Browser for Errors

Although your code has been fairly simple so far, it is still possible to make errors when typing it in. As you start to look at more complex and detailed code, this will become more and more of a problem. So, before discuss how you can use the data stored in variables, this seems like a good point to discuss how to ensure that any errors that arise in your code are shown to you by the browser, so that you can go and correct them.

When you are surfing other people's web sites, you probably won't be interested in seeing when there are errors in their code. In this situation, it's tempting to find a way of switching off the display of error dialog boxes in the browser. However, as JavaScript programmers, we want to know all the gory details about errors in our own web pages; that way we can fix them before someone else spots them. It's important, therefore, to make sure the browsers we use to test our web sites are configured correctly to show errors and their details. In this section, this is exactly what we're going to do.

Displaying Errors in Firefox

Firefox keeps quiet about your errors, so if things go wrong you won't see any pop-up boxes warning you or alarm bells going off. However, one of its developer tools is the JavaScript Console, which contains details of any JavaScript problems on your page. It also reports other problems, such as invalid CSS.

To view this console from Firefox, go to the Tools menu and select JavaScript Console, as shown in Figure 2-3.

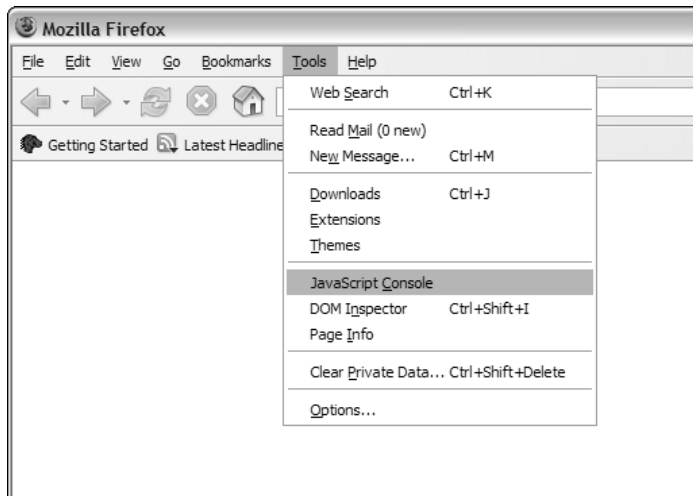


Figure 2-3

The JavaScript Console then pops open in its own separate window, as shown in Figure 2-4.

While you're developing your JavaScript code, it's as well to leave the JavaScript Console open. At the moment it is probably blank. Shortly you'll create a deliberate error and see what happens in the JavaScript Console.



Figure 2-4

Displaying Errors in Internet Explorer

Normally, IE will by default display JavaScript errors using dialog boxes. However, it is possible to turn off the displaying of such errors, in which case you need to follow a few simple steps to re-enable error displaying. First open up Internet Explorer and select the Internet Options menu from the Tools menu, as shown in Figure 2-5.



Figure 2-5

Chapter 2: Data Types and Variables

In the dialog box that appears, select the Advanced tab. Under Browsing, make sure the Disable script debugging (Other) check box is cleared and that the Display a notification about every script error check box is selected, as shown in Figure 2-6. Note that IE 4 doesn't have a Display notification about every script error check box, so you just need to clear the Disable script debugging check box. After you've done this, you can click OK to close the dialog box.

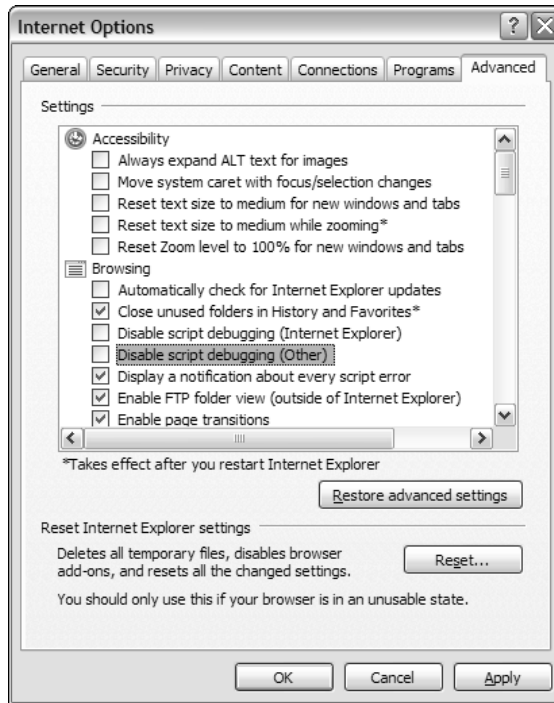


Figure 2-6

Okay, now that you have the display of error messages sorted out, you'll look at what happens when you have an error in your code. Note that as with Firefox there is a program available from Microsoft to help root out errors in your code; you'll see how to use it in Chapter 10.

What Happens When You Get an Error

As mentioned in the previous section, the use of a reserved word in a variable name will result in a JavaScript error in the browser. However, the error message displayed may not be instantly helpful since it may not indicate that you've used a reserved word in declaring your variables. Let's look at the sort of error messages you might see in this situation. Note that these error messages can also be produced by other mistakes not related to variable naming, which can get confusing at times. You'll look at these other mistakes later in the book—indeed, the whole of Chapter 10 is devoted to spotting and fixing errors.

Let's assume that you try to define a variable called `with` like this:

```
var with;
```

The word `with` is reserved in JavaScript. What errors will you see?

In Firefox you see nothing unless you open the JavaScript console, in which case you see something like what is shown in Figure 2-7.

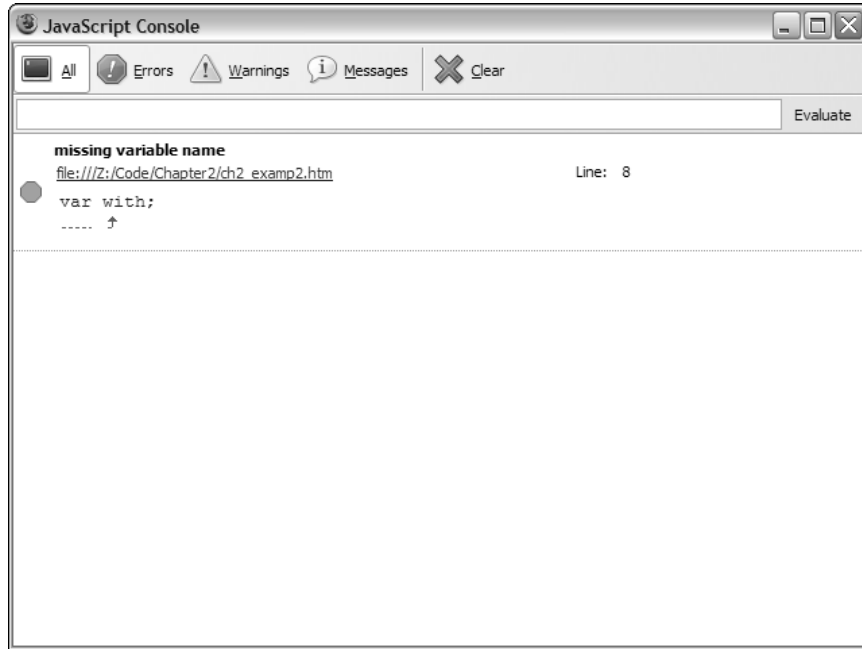


Figure 2-7

When you're developing code, it's probably easiest to leave the console open. You'll see a more sophisticated way of solving code problems in Chapter 10 when you look at Venkman, a program released by Mozilla, the creators of Firefox, to help with removing errors.

When you double-click the link to the file in the JavaScript Console, here `file:///Z:/Code/Chapter2/ch2_examp2.htm`, it'll open up the source code of the file and take you to where the error occurred, as shown in Figure 2-8.

In Internet Explorer, as long as you have the display errors enabled, as discussed in the section above, the sort of message you can expect to see is shown in Figure 2-9.

If you have IE 6+ and didn't see either of the error messages in Figure 2-8 or Figure 2-9, don't panic. In the browser's status bar (usually at the bottom of the browser window), you'll notice a little yellow triangle with an exclamation mark inside it. Double-click the yellow triangle, and the error message dialog box appears. Make sure you check the Always display this message when a page error occurs check box.

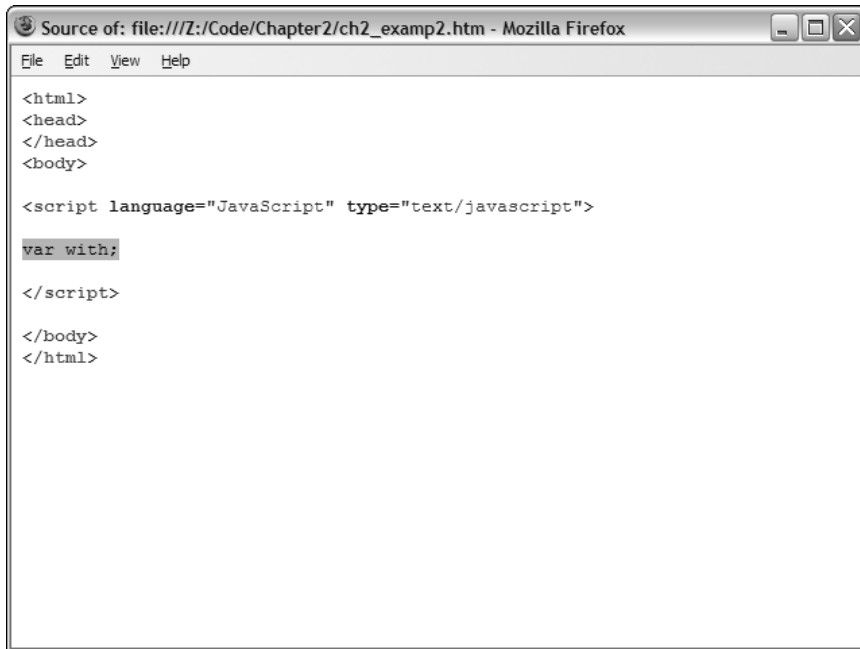


Figure 2-8

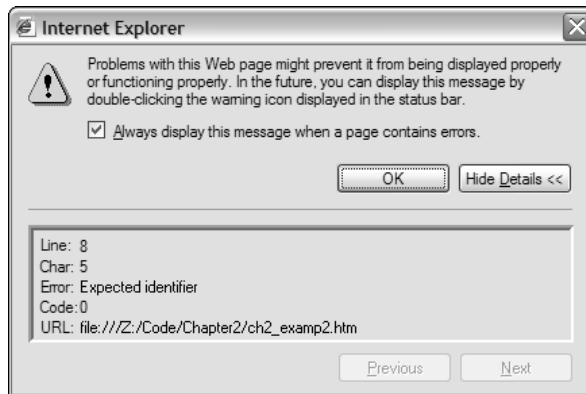


Figure 2-9

The rest of the book will show error messages for Firefox. Bear in mind, though, that it doesn't matter what your dialog box looks like, so long as you're getting an indication that an error has occurred and of what caused it.

Using Data — Calculations and Basic String Manipulation

Now that you've seen how to cope with errors, you can get back to the main subject of this chapter: data and how to use them. You've seen how to declare variables and how they can store information, but so far you haven't done anything really useful with this knowledge — so just why would you want to use variables at all?

What variables enable you to do is temporarily hold information that you can use for processing in mathematical calculations, in building up text messages, or in processing words that the user has entered. Variables are a little bit like the memory store button on the average pocket calculator. Say you were adding up your finances. You might first add up all the money you needed to spend, and then store it in temporary memory. After you had added up all your money coming in, you could deduct the amount stored in the memory to figure out how much would be left over. Variables can be used in a similar way: You can first gain the necessary user input and store it in variables, and then you can do your calculations using the values obtained.

In this section you'll see how you can put the values stored in variables to good use in both number-crunching and text-based operations.

Numerical Calculations

JavaScript has a range of basic mathematical capabilities, such as addition, subtraction, multiplication, and division. Each of the basic math functions is represented by a symbol: plus (+), minus (-), star (*), and forward slash (/), respectively. These symbols are called *operators* because they operate on the values you give them. In other words, they perform some calculation or operation and return a result to us. You can use the results of these calculations almost anywhere you'd use a number or a variable.

Imagine you were calculating the total value of items on a shopping list. You could write this calculation as follows:

Total cost of shopping = 10 + 5 + 5

or, if you actually calculate the sum, it's

Total cost of shopping = 20

Now let's see how to do this in JavaScript. In actual fact, it is very similar except that you need to use a variable to store the final total.

```
var TotalCostOfShopping;  
TotalCostOfShopping = 10 + 5 + 5;  
alert(TotalCostOfShopping);
```

Chapter 2: Data Types and Variables

First, you declare a variable, `TotalCostOfShopping`, to hold the total cost.

In the second line you have the code `10 + 5 + 5`. This piece of code is known as an *expression*. When you assign the variable `TotalCostOfShopping` the value of this expression, JavaScript automatically calculates the value of the expression (20) and stores it in the variable. Notice that we've used the equals sign to tell JavaScript to store the results of the calculation in the `TotalCostOfShopping` variable. This is called *assigning* the value of the calculation to the variable, which is why the single equals sign (=) is called the *assignment operator*.

Finally, you display the value of the variable in an alert box.

The operators for subtraction and multiplication work in exactly the same way. Division is a little different.

Try It Out Calculations

Let's take a look at an example using the division operator to see how it works. Enter the following code and save it as `ch2_examp3.htm`:

```
<html>
<body>

<script language="JavaScript" type="text/javascript">
var firstNumber = 15;
var secondNumber = 10;
var answer;
answer = 15 / 10;
alert(answer);

alert(15 / 10);

answer = firstNumber / secondNumber;
alert(answer);

</script>

</body>
</html>
```

Load this into your web browser. You should see a succession of three alert boxes, each containing the value 1.5. These values are the results of three calculations.

How It Works

The first thing you do in the script block is declare your three variables and assign the first two of these variables values that you'll be using later.

```
var firstNumber = 15;
var secondNumber = 10;
var answer;
```


Next you set the `answer` variable to the results of the calculation of the expression `15/10`. You show the value of this variable in an `alert` box.

```
answer = 15 / 10;  
alert(answer);
```

This example demonstrates one way of doing the calculation, but in reality you'd almost never do it this way.

To demonstrate that you can use expressions in places you'd use numbers or variables, you show the results of the calculation of `15/10` directly by including it in the `alert()` function.

```
alert(15 / 10);
```

Finally you do the same calculation, but this time using the two variables `firstNumber`, which was set to 15, and `secondNumber`, which was set to 10. You have the expression `firstNumber / secondNumber`, the result of which you store in our `answer` variable. Then, to prove it has all worked, you show the value contained in `answer` by using your friend the `alert()` function.

```
answer = firstNumber / secondNumber;  
alert(answer);
```

Most calculations will be done in the third way; that is, using variables, or numbers and variables, and storing the result in another variable. The reason for this is that if the calculation used literal values (actual values, such as `15 / 10`), then you might as well program in the result of the calculation, rather than force JavaScript to calculate it for you. For example, rather than writing `15 / 10`, you might as well just write `1.5`. After all, the more calculations you force JavaScript to do, the slower it will be, though admittedly just one calculation won't tax it too much.

Another reason for using the result rather than the calculation is that it makes code more readable. Which would you prefer to read in code, `1.5 * 45 - 56 / 67 + 2.567` or `69.231`? Still better, a variable named, for example, `PricePerKG`, makes code even easier to understand for someone not familiar with it.

Increment and Decrement Operators

A number of operations using the math operators are so commonly used that they have been given their own operators. The two you'll be looking at here are the *increment* and *decrement* operators, which are represented by two plus signs (`++`) and two minus signs (`--`), respectively. Basically, all they do is increase or decrease a variable's value by one. You could use the normal `+` and `-` operators to do this, for example:

```
myVariable = myVariable + 1;  
myVariable = myVariable - 1;
```

(Note that you can assign a variable a new value that is the result of an expression involving its previous value.) However, using the increment and decrement operators shortens this to

```
myVariable++;  
myVariable--;
```

Chapter 2: Data Types and Variables

The result is the same — the value of `myVariable` is increased or decreased by one — but the code is shorter. When you are familiar with the syntax, this becomes very clear and easy to read.

Right now, you may well be thinking that these operators sound as useful as a poke in the eye. However, in the next chapter when you look at how you can run the same code a number of times, you'll see that these operators are very useful and widely used. In fact, the `++` operator is so widely used it has a computer language named after it, C++. The joke here is that C++ is one up from C. (Well, that's programmer humor for you!)

As well as placing the `++` or `--` after the variable, you can also place it before, like so:

```
++myVariable;  
--myVariable;
```

When the `++` and `--` are used on their own, as they usually are, it makes no difference where they are placed, but it is possible to use the `++` and `--` operators in an expression along with other operators. For example:

```
myVar = myNumber++ - 20;
```

This code takes 20 away from `myNumber` and then increments the variable `myNumber` by one, before assigning the result to the variable `myVar`. If instead you place the `++` before, and prefix it like this:

```
myVar = ++myNumber - 20;
```

first `myNumber` is incremented by one, and then `myNumber` has 20 subtracted from it. It's a subtle difference but in some situations a very important one. Take the following code:

```
myNumber = 1;  
myVar = (myNumber++ * 10 + 1);
```

What value will `myVar` contain? Well, because the `++` is postfix (it's after the `myNumber` variable), it will be incremented afterwards. So the equation reads: Multiply `myNumber` by 10 plus 1 and then increment `myNumber` by one.

```
myVar = 1 * 10 + 1 = 11
```

Then add 1 to `myNumber` to get 12, but this is done after the value 11 has been assigned to `myVar`. Now take a look at the following code:

```
myNumber = 1;  
myVar = (++myNumber * 10 + 1);
```

This time `myNumber` is incremented by one first, then times 10 and plus 1.

```
myVar = 2 * 10 + 1 = 21
```

As you can imagine, such subtlety can easily be overlooked and lead to bugs in code; therefore, it's usually best to avoid this syntax.

Before going on, this seems to be a good point to introduce another operator, `+=`. This operator can be used as a shortcut for increasing the value held by a variable by a set amount. For example,

```
myVar += 6;
```

does exactly the same thing as

```
myVar = myVar + 6;
```

You can also do the same thing for subtraction and multiplication, as shown here:

```
myVar -= 6;  
myVar *= 6;
```

which is equivalent to

```
myVar = myVar - 6;  
myVar = myVar * 6;
```

Operator Precedence

You saw that symbols that perform some function — like `+`, which adds two numbers together, and `-`, which subtracts one number from another — are called operators. Unlike people, not all operators are created equal; some have a higher *precedence* — that is, they get dealt with sooner. A quick look at a simple example will help demonstrate my point.

```
<html>  
<body>  
  
<script language="JavaScript" type="text/javascript">  
  
var myVariable;  
  
myVariable = 1 + 1 * 2;  
  
alert(myVariable);  
  
</script>  
  
</body>  
</html>
```

If you were to type this in, what result would you expect the `alert` box to show as the value of `myVariable`? You might expect that since $1 + 1 = 2$ and $2 * 2 = 4$, the answer is 4. Actually, you'll find that the `alert` box shows 3 as the value stored in `myVariable` as a result of the calculation. So what gives? Doesn't JavaScript add up right?

Well, you probably already know the reason from your understanding of mathematics. The way JavaScript does our calculation is to first calculate $1 * 2 = 2$, and then use this result in the addition, so that JavaScript finishes off with $1 + 2 = 3$.

Chapter 2: Data Types and Variables

Why? Because `*` has a higher precedence than `+`. The `=` symbol, also an operator (called the assignment operator), has the lowest precedence—it always gets left until last.

The `+` and `-` operators have an equal precedence, so which one gets done first? Well, JavaScript works from left to right, so if operators with equal precedence exist in a calculation, they get calculated in the order in which they appear when going from left to right. The same applies to `*` and `/`, which are also of equal precedence.

Try It Out Fahrenheit to Centigrade

Take a look at a slightly more complex example—a Fahrenheit to centigrade converter. (Centigrade is another name for the Celsius temperature scale.) Type in this code and save it as `ch2_examp4.htm`:

```
<html>
<body>

<script language="JavaScript" type="text/javascript">
// Equation is °C = 5/9 (°F - 32).
var degFahren = prompt("Enter the degrees in Fahrenheit",50);
var degCent;

degCent = 5/9 * (degFahren - 32);

alert(degCent);

</script>

</body>
</html>
```

If you load the page into your browser, you should see a prompt box, like that shown in Figure 2-10, that asks you to enter the degrees in Fahrenheit to be converted. The value 50 is already filled in by default.

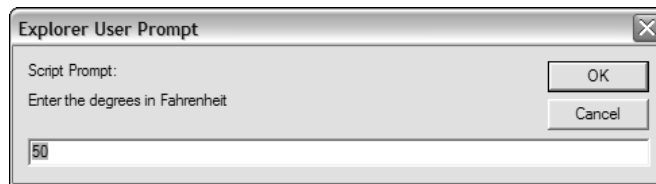


Figure 2-10

If you leave it at 50 and click OK, an `alert` box with the number 10 in it appears. This represents 50 degrees Fahrenheit converted to centigrade.

Reload the page and try changing the value in the prompt box to see what results you get. For example, change the value to 32 and reload the page. This time you should see 0 appear in the box.

As it's still a fairly simple example, there's no checking of data input so it'll let you enter `abc` as the degrees Fahrenheit. Later, in the "Data Type Conversion" section of this chapter, you'll see how to spot invalid characters posing as numeric data.

How It Works

The first line of the script block is a comment since it starts with two forward slashes (`//`). It contains the equation for converting Fahrenheit temperatures to centigrade and is in the example code solely for reference.

```
// Equation is °C = 5/9 (°F - 32).
```

Your task is to represent this equation in JavaScript code. You start by declaring your variables, `degFahren` and `degCent`.

```
var degFahren = prompt("Enter the degrees in Fahrenheit",50);  
var degCent;
```

Instead of initializing the `degFahren` variable to a literal value, you get a value from the user using the `prompt()` function. The `prompt()` function works in a similar way to an `alert()` function, except that as well as displaying a message, it also contains a text box in which the user can enter a value. It is this value that will be stored inside the `degFahren` variable. The value returned is a text string but this will be implicitly converted by JavaScript to a number when you use it as a number, as discussed in the section on data type conversion later in this chapter.

You pass two pieces of information to the `prompt()` function:

- ❑ The text to be displayed — usually a question that prompts the user for input.
- ❑ The default value that is contained in the input box when the prompt dialog box first appears.

These two pieces of information must be specified in the given order and separated by a comma. If you don't want a default value to be contained in the input box when the prompt box opens, you should use an empty string (`" "`) for the second piece of information.

As you can see in the preceding code, the text is "Enter the degrees in Fahrenheit," and the default value in the input box is 50.

Next in the script block comes the equation represented in JavaScript. You store the result of the equation in the `degCent` variable. You can see that the JavaScript looks very much like the equation you have in the comment, except that you use `degFahren` instead of `°F`, and `degCent` rather than `°C`.

```
degCent = 5/9 * (degFahren - 32);
```

The calculation of the expression on the right-hand side of the equals sign raises a number of important points. First, just as in math, the JavaScript equation is read from left to right, at least for the basic math functions like `+`, `-`, and so on. Secondly, as you saw earlier, just as there is precedence in math, there is in JavaScript.

Chapter 2: Data Types and Variables

Starting from the left, first JavaScript works out $5/9 = .5556$ (approximately). Then it comes to the multiplication, but wait . . . the last bit of our equation, `degFahren - 32`, is in parentheses. This raises the order of precedence and causes JavaScript to calculate the result of `degFahren - 32` before doing the multiplication. For example, when `degFahren` is set to 50, $(\text{degFahren} - 32) = (50 - 32) = 18$. Now JavaScript does the multiplication, $.5556 * 18$, which is approximately 10.

What if you didn't use the parentheses? Then your code would be

```
degCent = 5/9 * degFahren - 32;
```

The calculation of $5/9$ remains the same, but then JavaScript would have calculated the multiplication, $5/9 * \text{degFahren}$. This is because the multiplication takes precedence over the subtraction. When `degFahren` is 50, this equates to $5/9 * 50 = 27.7778$. Finally JavaScript would have subtracted the 32, leaving the result as -4.2221 ; not the answer you want!

Finally, in your script block, you display the answer using the `alert()` function.

```
alert(degCent);
```

That concludes a brief look at basic calculations with JavaScript. However, in Chapter 4 you'll be looking at the `Math` object, which enables you to do more complex calculations.

Basic String Operations

In an earlier section, you looked at the text or string data type, as well as numerical data. Just as numerical data have associated operators, strings have operators too. This section introduces some basic string manipulation techniques using such operators. Strings are covered in more depth in Chapter 4, and advanced string handling is covered in Chapter 8.

One thing you'll find yourself doing again and again in JavaScript is joining two strings together to make one string—a process that's termed *concatenation*. For example, you may want to concatenate the two strings "Hello " and "Paul" to make the string "Hello Paul". So how do you concatenate? Easy! Use the `+` operator. Recall that when applied to numbers, the `+` operator adds them up, but when used in the context of two strings, it joins them together.

```
var concatString = "Hello " + "Paul";
```

The string now stored in the variable `concatString` is "Hello Paul". Notice that the last character of the string "Hello " is a space—if you left this out, your concatenated string would be "HelloPaul".

Try It Out Concatenating Strings

Let's look at an example using the `+` operator for string concatenation. Type the following code and save it as `ch2_exam5.htm`:

```
<html>
<body>

<script language="JavaScript" type="text/javascript">

var greetingString = "Hello";
```

```
var myName = prompt("Please enter your name", "");
var concatString;

document.write(greetingString + " " + myName + "<br>");

concatString = greetingString + " " + myName;

document.write(concatString);

</script>

</body>
</html>
```

If you load it into your web browser, you should see a prompt box asking for your name.

Enter your name and click OK. You should see a greeting and your name displayed twice on the web page.

How It Works

You start the script block by declaring three variables. You set the first variable, `greetingString`, to a string value. The second variable, `myName`, is assigned to whatever is entered by the user in the prompt box. You do not initialize the third variable, `concatString`, here. It will be used to store the result of the concatenation that you'll do later in the code.

```
var greetingString = "Hello";
var myName = prompt("Please enter your name", "");
var concatString;
```

In the last chapter, you saw how the web page was represented by the concept of a document and that it had a number of different properties, such as `bgColor`. You can also use `document` to write text and HTML directly into the page itself. You do this by using the word `document`, followed by a dot, and then `write()`. You then use `document.write()` much as you do the `alert()` function, in that you put the text that you want displayed in the web page inside the parentheses following the word `write`. Don't worry too much about this here, though, because it will all be explained in detail in Chapter 4. However, you now make use of `document.write()` in your code to write the result of an expression to the page.

```
document.write(greetingString + " " + myName + "<br>");
```

The expression written to the page is the concatenation of the value of the `greetingString` variable, a space (" "), the value of the `myName` variable, and the HTML `
` tag, which causes a line break. For example, if you enter `Paul` into the prompt box, the value of this expression will be as follows:

```
Hello Paul<br>
```

In the next line of code is a similar expression. This time it is just the concatenation of the value in the variable `greetingString`, a space, and the value in the variable `myName`. You store the result of this expression in the variable `concatString`. Finally, you write the contents of the variable `concatString` to the page using `document.write()`.

```
concatString = greetingString + " " + myName;
document.write(concatString);
```

Mixing Numbers and Strings

What if you want to mix text and numbers in an expression? A prime example of this would be in the temperature converter you saw earlier. In the example, you just display the number without telling the user what it actually means. What you really want to do is display the number with descriptive text wrapped around it, such as “The value converted to degrees centigrade is 10.”

Mixing numbers and text is actually very easy. You can simply join them together using the `+` operator. JavaScript is intelligent enough to know that when both a string and a number are involved, you’re not trying to do numerical calculations, but rather that you want to treat the number as a string and join it to the text. For example, to join the text `My age is` and the number `101` together, you could simply do the following:

```
alert("My age is " + 101);
```

This would produce an `alert` box with “My age is 101” inside it.

Try It Out Making the Temperature Converter User-Friendly

You can try out this technique of concatenating strings and numbers in our temperature-converter example. You’ll output some explanatory text, along with the result of the conversion calculation. The changes that you need to make are very small, so load `ch2_examp4.htm` into your text editor and change the following line. Then save it as `ch2_examp6.htm`.

```
<html>
<body>

<script language="JavaScript" type="text/javascript">

// Equation is °C = 5/9 (°F - 32).

var degFahren = prompt("Enter the degrees in Fahrenheit",50);
var degCent;

degCent = 5/9 * (degFahren - 32);

alert(degFahren + "\xB0 Fahrenheit is " + degCent + "\xB0 centigrade");

</script>

</body>
</html>
```

Load the page into your web browser. Click OK in the prompt box to submit the value 50, and this time you should see the box shown in Figure 2-11.



Figure 2-11

How It Works

This example is identical to `ch2_examp4.htm`, except for one line:

```
alert(degFahren + "\xB0 Fahrenheit is " + degCent + "\xB0 centigrade");
```

so we will just look at this line here. You can see that the `alert()` function contains an expression. Let's look at that expression more closely.

First is the variable `degFahren`, which contains numerical data. You concatenate that to the string `"\xB0 Fahrenheit is "`. JavaScript realizes that because you are adding a number and a string, you want to join them together into one string rather than trying to take their sum, and so automatically converts the number contained in `degFahren` to a string. You next concatenate this string to the variable `degCent`, containing numerical data. Again JavaScript converts the value of this variable to a string. Finally you concatenate to the string `"\xB0 centigrade"`.

Note also the escape sequence used to insert the degree character into the strings. You'll remember from earlier in the chapter that `\xNN` can be used to insert special characters not available to type in directly. (*NN* is a hexadecimal number representing a character from the Latin-1 character table). So when JavaScript spots `\xB0` in a string, instead of showing those characters it does a lookup to see what character is represented by `B0` and shows that instead.

Something to be aware of when using special characters is they are not necessarily cross-platform-compatible. Although you can use `\xNN` for a certain character on a Windows computer, you may find you need to use a different character on a Mac or a Unix machine.

You'll look at more string manipulation techniques in Chapter 4—you'll see how to search strings and insert characters in the middle of them, and in Chapter 8 you'll see some very sophisticated string techniques.

Data Type Conversion

As you saw, if you add a string and a number together, JavaScript makes the sensible choice and converts the number to a string, then concatenates the two. Usually JavaScript has enough sense to make data type conversions like this whenever it needs to, but there are some situations in which you need to convert the type of a piece of data yourself. For example, you may be given a piece of string data that you want to think of as a number. This is especially likely if you are using forms to collect data from the user. Any values input by the user are treated as strings, even though they may contain numerical data, such as the user's age.

Chapter 2: Data Types and Variables

Why is changing the type of the data so important? Consider a situation in which you collect two numbers from the user using a form and want to calculate their sum. The two numbers are available to you as strings, for example "22" and "15". When you try to calculate the sum of these values using "22" + "15" you get the result "2215", because JavaScript thinks you are trying to concatenate two strings rather than trying to find the sum of two numbers.

In this section you'll look at two conversion functions that convert strings to numbers: `parseInt()` and `parseFloat()`.

Let's take `parseInt()` first. This function takes a string and converts it to an integer. The name is a little confusing at first—why `parseInt()` rather than `convertToInt()`? The main reason for the name comes from the way that the function works. It actually goes through (that is, parses) each character of the string you ask it to convert and sees if it's a valid number. If it is valid, `parseInt()` uses it to build up the number; if it is not valid, the command simply stops converting and returns the number it has converted so far.

For example, if your code is `parseInt("123")`, JavaScript will convert the string "123" to the number 123. For the code `parseInt("123abc")`, JavaScript will also return the number 123. When the JavaScript interpreter gets to the letter a, it assumes the number has ended and gives 123 as the integer version of the string "123abc".

The `parseFloat()` function works in the same way as `parseInt()`, except that it returns floating-point numbers—fractional numbers—and that a decimal point in the string, which it is converting, is considered to be part of the allowable number.

Try It Out Converting Strings to Numbers

Let's look at an example using `parseInt()` and `parseFloat()`. Enter the following code and save it as `ch2_examp7.htm`:

```
<html>
<body>

<script language="JavaScript" type="text/javascript">

var myString = "56.02 degrees centigrade";
var myInt;
var myFloat;

document.write("\n" + myString + "\n is " + parseInt(myString) +
    " as an integer" + "<BR>");

myInt = parseInt(myString);
document.write("\n" + myString + "\n when converted to an integer equals " +
    myInt + "<BR>");

myFloat = parseFloat(myString);
document.write("\n" + myString +
    "\n when converted to a floating point number equals " + myFloat);

</script>

</body>
</html>
```

Load it into your browser, and you'll see three lines written in the web page, as shown in Figure 2-12.

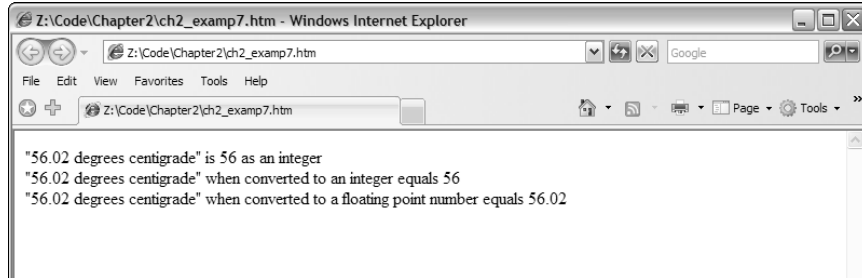


Figure 2-12

How It Works

Your first task in the script block is to declare some variables. The variable `myString` is declared and initialized to the string you want to convert. You could just as easily have used the string directly in this example rather than storing it in a variable, but in practice you'll find that you use variables more often than literal values. You also declare the variables `myInt` and `myFloat`, which will hold the converted numbers.

```
var myString = "56.02 degrees centigrade";  
var myInt;  
var myFloat;
```

Next, you write to the page the converted integer value of `myString` displayed inside a user-friendly sentence you build up using string concatenation. Notice that you use the escape sequence `\` to display quotes (") around the string you are converting.

```
document.write("\"" + myString + "\" is " + parseInt(myString) +  
" as an integer" + "<BR>");
```

As you can see, you can use `parseInt()` and `parseFloat()` in the same places you would use a number itself or a variable containing a number. In fact, in this line the JavaScript interpreter is doing two conversions. First it converts `myString` to an integer, because that's what you asked for by using `parseInt()`. Then it automatically converts that integer number back to a string, so it can be concatenated with the other strings to make up your sentence. Also note that only the 56 part of the `myString` variable's value is considered a valid number when you're dealing with integers. Anything after the 6 is considered invalid and is ignored.

Next you do the same conversion of `myString` using `parseInt()`, but this time you store the result in the `myInt` variable. On the following line you use the result in some text you display to the user:

```
myInt = parseInt(myString);  
document.write("\"" + myString + "\" when converted to an integer equals " +  
myInt + "<BR>");
```

Chapter 2: Data Types and Variables

Again, though `myInt` holds a number, the JavaScript interpreter knows that `+`, when a string and a number are involved, means you want the `myInt` value converted to a string and concatenated to the rest of the string so it can be displayed.

Finally, you use `parseFloat()` to convert the string in `myString` to a floating-point number, which you store in the variable `myFloat`. This time the decimal point is considered to be a valid part of the number, so it's anything after the 2 that is ignored. Again you use `document.write()` to write the result to the web page inside a user-friendly string.

```
myFloat = parseFloat(myString);
document.write("\"" + myString +
    "\" when converted to a floating point number equals " + myFloat);
```

Dealing with Strings That Won't Convert

Some strings simply are not convertible to numbers, such as strings that don't contain any numerical data. What happens if you try to convert these strings? As a little experiment, try changing the preceding example so that `myString` holds something that is not convertible. For example, change the line

```
var myString = "56.02 degrees centigrade";
```

to

```
var myString = "I'm a name not a number";
```

Now reload the page in your browser and you should see what's shown in Figure 2-13.

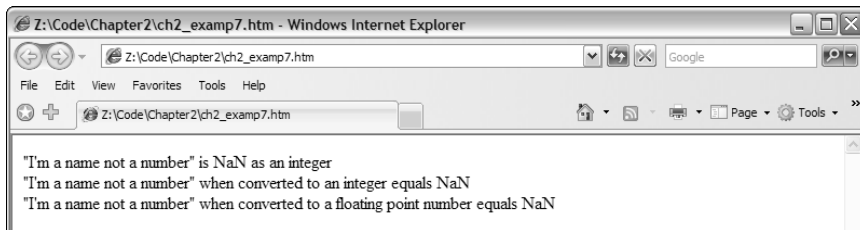


Figure 2-13

You can see that in the place of the numbers you got before, you get `NaN`. What sort of number is that? Well, it's *Not a Number* at all!

If you use `parseInt()` or `parseFloat()` with any string that is empty or does not start with at least one valid digit, you get `NaN`, meaning *Not a Number*.

`NaN` is actually a special value in JavaScript. It has its own function, `isNaN()`, which checks whether something is `NaN` or not. For example,

```
myVar1 = isNaN("Hello");
```

will store the value `true` in the variable `myVar1`, since `"Hello"` is not a number, whereas

```
myVar2 = isNaN("34");
```

will store the value `false` in the variable `myVar2`, since `34` can be converted successfully from a string to a number by the `isNaN()` function.

In the next chapter you'll see how you can use the `isNaN()` function to check the validity of strings as numbers, something that proves invaluable when dealing with user input, as you'll see in Chapter 6.

Arrays

Now we're going to look at a new concept — something called an *array*. An array is similar to a normal variable, in that you can use it to hold any type of data. However, it has one important difference, which you'll see below.

As you have already seen, a normal variable can only hold one piece of data at a time. For example, you can set `myVariable` to be equal to `25` like so:

```
myVariable = 25;
```

and then go and set it to something else, say `35`:

```
myVariable = 35;
```

However, when you set the variable to `35`, the first value of `25` is lost. The variable `myVariable` now holds just the number `35`.

The following table illustrates the variable:

Variable name	Value
<code>myVariable</code>	<code>35</code>

The difference between such a normal variable and an array is that an array can hold *more than one* item of data at the same time. For example, you could use an array with the name `myArray` to store both the numbers `25` and `35`. Each place where a piece of data can be stored in an array is called an *element*.

How do you distinguish between these two pieces of data in an array? You give each piece of data an *index* value. To refer to that piece of data you enclose its index value in square brackets after the name of the array. For example, an array called `myArray` containing the data `25` and `35` could be illustrated using the following table:

Element name	Value
<code>myArray[0]</code>	<code>25</code>
<code>myArray[1]</code>	<code>35</code>

Chapter 2: Data Types and Variables

Notice that the index values start at 0 and not 1. Why is this? Surely 1 makes more sense — after all, we humans tend to say the first item of data, followed by the second item, and so on. Unfortunately, computers start from 0, and think of the first item as the zero item, the second as the first item, and so on. Confusing, but you'll soon get used to this.

Arrays can be very useful since you can store as many (within the limits of the language, which specifies a maximum of two to the power of 32 elements) or as few items of data in an array as you want. Also, you don't have to say up front how many pieces of data you want to store in an array, though you can if you wish.

So how do you create an array? This is slightly different from declaring a normal variable. To create a new array, you need to declare a variable name and tell JavaScript that you want it to be a new array using the `new` keyword and the `Array()` function. For example, the array `myArray` could be defined like this:

```
var myArray = new Array();
```

Note that, as with everything in JavaScript, the code is case-sensitive, so if you type `array()` rather than `Array()`, the code won't work. This way of defining an array will be explained further in Chapter 4.

As with normal variables, you can also declare your variable first, and then tell JavaScript you want it to be an array. For example:

```
var myArray;  
myArray = new Array();
```

Earlier you learned that you can say up front how many elements the array will hold if you want to, although this is not necessary. You do this by putting the number of elements you want to specify between the parentheses after `Array`. For example, to create an array that will hold six elements, you write the following:

```
var myArray = new Array(6);
```

You have seen how to declare a new array, but how do you store your pieces of data inside it? You can do this when you define your array by including your data inside the parentheses, with each piece of data separated by a comma. For example:

```
var myArray = new Array("Paul", 345, "John", 112, "Bob", 99);
```

Here the first item of data, "Paul", will be put in the array with an index of 0. The next piece of data, 345, will be put in the array with an index of 1, and so on. This means that the element with the name `myArray[0]` contains the value "Paul", the element with the name `myArray[1]` contains the value 345, and so on.

Note that you can't use this method to declare an array containing just one piece of numerical data, such as 345, because JavaScript assumes that you are declaring an array that will hold 345 elements.

This leads to another way of declaring data in an array. You could write the preceding line like this:

```
var myArray = new Array();  
myArray[0] = "Paul";  
myArray[1] = 345;  
myArray[2] = "John";  
myArray[3] = 112;  
myArray[4] = "Bob";  
myArray[5] = 99;
```

You use each element name as you would a variable, assigning them with values. You'll learn this method of declaring the values of array elements in the following "Try It Out" section.

Obviously, in this example the first way of defining the data items is much easier. However, there will be situations in which you want to change the data stored in a particular element in an array after they have been declared. In that case you will have to use the latter method of defining the values of the array elements.

You'll also spot from the preceding example that you can store different data types in the same array. JavaScript is very flexible as to what you can put in an array and where you can put it.

Before going on to an example, note here that if, for example, you had defined your array called `myArray` as holding three elements like this:

```
var myArray = new Array(3);
```

and then defined a value in the element with index 130

```
myArray[130] = "Paul";
```

JavaScript would not complain and would happily assume that you had changed your mind and wanted an array that had (at least) 131 elements in it.

Try It Out An Array

In the following example, you'll create an array to hold some names. You'll use the second method described in the preceding section to store these pieces of data in the array. You'll then display the data to the user. Type the code in and save it as `ch2_examp8.htm`.

```
<html>  
<body>  
  
  <script language="JavaScript" type="text/javascript">  
  
    var myArray = new Array();  
    myArray[0] = "Bob";  
    myArray[1] = "Pete";  
    myArray[2] = "Paul";  
  
    document.write("myArray[0] = " + myArray[0] + "<BR>");
```

Chapter 2: Data Types and Variables

```
document.write("myArray[2] = " + myArray[2] + "<BR>");
document.write("myArray[1] = " + myArray[1] + "<BR>");

myArray[1] = "Mike";
document.write("myArray[1] changed to " + myArray[1]);

</script>

</body>
</html>
```

If you load this into your web browser, you should see a web page that looks something like the one shown in Figure 2-14.

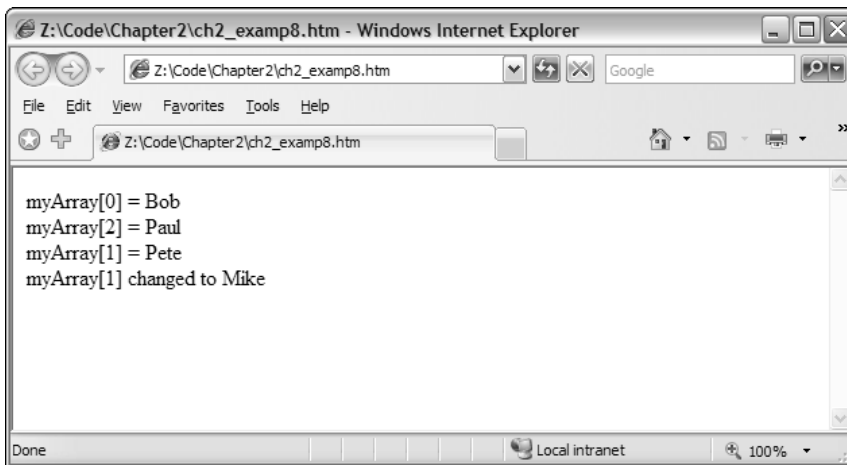


Figure 2-14

How It Works

The first task in the script block is to declare a variable and tell the JavaScript interpreter you want it to be a new array.

```
var myArray = new Array();
```

Now that you have your array defined, you can store some data in it. Each time you store an item of data with a new index, JavaScript automatically creates a new storage space for it. Remember that the first element will be at `myArray[0]`.

Let's take each addition to the array in turn and see what's happening. Before you add anything, your array is empty. Then you add an array element with the following line:

```
myArray[0] = "Bob";
```


Your array now looks like this:

Index	Data Stored
0	Bob

Then you add another element to the array, this time with an index of 1.

```
myArray[1] = "Pete";
```

Index	Data Stored
0	Bob
1	Pete

Finally, you add another element to the array with an index of 2.

```
myArray[2] = "Paul";
```

Your array now looks like this:

Index	Data Stored
0	Bob
1	Pete
2	Paul

Next, you use a series of `document.write()` functions to insert the values that each element of the array contains into the web page. Here the array is out of order just to demonstrate that you can access it that way.

```
document.write("myArray[0] = " + myArray[0] + "<BR>");  
document.write("myArray[2] = " + myArray[2] + "<BR>");  
document.write("myArray[1] = " + myArray[1] + "<BR>");
```

You can treat each particular position in an array as if it's a standard variable. So you can use it to do calculations, transfer its value to another variable or array, and so on. However, if you try to access the data inside an array position before you have defined it, you'll get `undefined` as a value.

Finally, you change the value of the second array position to "Mike". You could have changed it to a number because, just as with normal variables, you can store any data type at any time in each individual data position in an array.

```
myArray[1] = "Mike";
```

Chapter 2: Data Types and Variables

Now your array’s contents look like this:

Index	Data Stored
0	Bob
1	Mike
2	Paul

Just to show that the change you made has worked, you use `document.write()` to display the second element’s value.

```
document.write("myArray[1] changed to " + myArray[1]);
```

A Multi-Dimensional Array

Suppose you want to store a company’s personnel information in an array. You might have data such as names, ages, addresses, and so on. One way to create such an array would be to store the information sequentially — the first name in the first element of the array, then the corresponding age in the next element, the address in the third, the next name in the fourth element, and so on. Your array could look something like this:

Index	Data Stored
0	Name1
1	Age1
2	Address1
3	Name2
4	Age2
5	Address2
6	Name3
7	Age3
8	Address3

This would work, but there is a neater solution: using a *multi-dimensional array*. Up to now you have been using single-dimension arrays. In these arrays each element is specified by just one index — that is, one dimension. So, taking the preceding example, you can see Name1 is at index 0, Age1 is at index 1, and so on.

A multi-dimensional array is one with two or more indexes for each element. For example, this is how your personnel array could look as a two-dimensional array:

Index	0	1	2
0	Name1	Name2	Name3
1	Age1	Age2	Age3
2	Address1	Address2	Address3

You'll see how to create such multi-dimensional arrays in the following "Try It Out" section.

Try It Out A Two-Dimensional Array

The following example illustrates how you can create such a multi-dimensional array in JavaScript code, and how you can access the elements of this array. Type in the code and save it as `ch2_examp9.htm`.

```
<html>
<body>

<script language="JavaScript" type="text/javascript">

var personnel = new Array();

personnel[0] = new Array();
personnel[0][0] = "Name0";
personnel[0][1] = "Age0";
personnel[0][2] = "Address0";

personnel[1] = new Array();
personnel[1][0] = "Name1";
personnel[1][1] = "Age1";
personnel[1][2] = "Address1";

personnel[2] = new Array();
personnel[2][0] = "Name2";
personnel[2][1] = "Age2";
personnel[2][2] = "Address2";

document.write("Name : " + personnel[1][0] + "<BR>");
document.write("Age : " + personnel[1][1] + "<BR>");
document.write("Address : " + personnel[1][2]);

</script>

</body>
</html>
```

If you load it into your web browser, you'll see three lines written into the page, which represent the name, age, and address of the person whose details are stored in the `personnel[1]` element of the array, as shown in Figure 2-15.

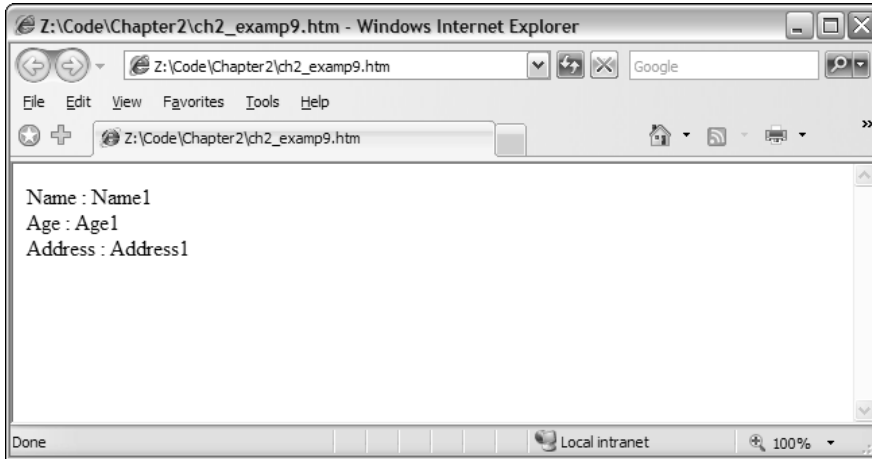


Figure 2-15

How It Works

The first thing to do in this script block is declare a variable, `personnel`, and tell JavaScript that you want it to be a new array.

```
var personnel = new Array();
```

Then you do something new; you tell JavaScript you want index 0 of the `personnel` array, that is, the element `personnel[0]`, to be another new array.

```
personnel[0] = new Array();
```

So what's going on? Well, the truth is that JavaScript doesn't actually support multi-dimensional arrays, only single ones. However, JavaScript enables us to fake multi-dimensional arrays by creating an array inside another array. So what the preceding line is doing is creating a new array inside the element with index 0 of our `personnel` array.

In the next three lines you put values into the newly created `personnel[0]` array. JavaScript makes it easy to do this: You just state the name of the array, `personnel[0]`, followed by another index in square brackets. The first index (0) belongs to the `personnel` array; the second index belongs to the `personnel[0]` array.

```
personnel[0][0] = "Name0";  
personnel[0][1] = "Age0";  
personnel[0][2] = "Address0";
```

After these lines of code, our array looks like this:

Index	0
0	Name0
1	Age0
2	Address0

The numbers at the top, at the moment just 0, refer to the `personnel` array. The numbers going down the side, 0, 1, and 2, are actually indices for the new `personnel[0]` array inside the `personnel` array.

For the second person's details, you repeat the process, but this time you are using the `personnel` array element with index 1.

```
personnel[1] = new Array();  
personnel[1][0] = "Name1";  
personnel[1][1] = "Age1";  
personnel[1][2] = "Address1";
```

Now your array looks like this:

Index	0	1
0	Name0	Name1
1	Age0	Age1
2	Address0	Address1

You create a third person's details in the next few lines. You are now using the element with index 2 inside the `personnel` array to create a new array.

```
personnel[2] = new Array();  
personnel[2][0] = "Name2";  
personnel[2][1] = "Age2";  
personnel[2][2] = "Address2";
```

The array now looks like this:

Index	0	1	2
0	Name0	Name1	Name2
1	Age0	Age1	Age2
2	Address0	Address1	Address2

Chapter 2: Data Types and Variables

You have now finished creating your multi-dimensional array. You end the script block by accessing the data for the second person (`Name1`, `Age1`, `Address1`) and displaying it in the page by using `document.write()`. As you can see, accessing the data is very much the same as storing them. You can use the multi-dimensional array anywhere you would use a normal variable or single-dimension array.

```
document.write("Name : " + personnel[1][0] + "<BR>");
document.write("Age : " + personnel[1][1] + "<BR>");
document.write("Address : " + personnel[1][2]);
```

Try changing the `document.write()` commands so that they display the first person's details. The code would look like this:

```
document.write("Name : " + personnel[0][0] + "<BR>");
document.write("Age : " + personnel[0][1] + "<BR>");
document.write("Address : " + personnel[0][2]);
```

It's possible to create multi-dimensional arrays of three, four, or even a hundred dimensions, but things can start to get very confusing, and you'll find that you rarely, if ever, need more than two dimensions. To give you an idea, here's how to declare and access a five-dimensional array:

```
var myArray = new Array();
myArray[0] = new Array();
myArray[0][0] = new Array();
myArray[0][0][0] = new Array();
myArray[0][0][0][0] = new Array();

myArray[0][0][0][0][0] = "This is getting out of hand"

document.write(myArray[0][0][0][0][0]);
```

That's it for arrays for now, but you'll return to them in Chapter 4 where you find out something shocking about them. You'll also learn about some of their more advanced features.

The “Who Wants To Be A Billionaire” Trivia Quiz — Storing the Questions Using Arrays

Okay, it's time to make your first steps in building the online trivia quiz. You're going to lay the foundations by defining the data that make up the questions and answers used in the quiz.

In this chapter you're just going to define multiple-choice questions, which have a single-letter answer. You'll be using arrays to store the questions and answers: a two-dimensional array for the questions and a single-dimensional one for the matching answers.

The format of each multiple-choice question will be the question followed by all the possible choices for answers. The correct answer to the question is specified using the letter corresponding to that answer.

For example, the question, “Who were the Beatles?” has options:

- A. A sixties rock group from Liverpool
- B. Four musically gifted insects
- C. German cars
- D. I don’t know. Can I have the questions on baseball please?

And the answer in this case is A.

So how do you store this information in our arrays? Let’s look at the array holding the questions first. You define the array something like this:

Index	0	1	2
0	Text for Question 0	Text for Question 1	Text for Question 2
1	Possible Answer A for Question 0	Possible Answer A for Question 1	Possible Answer A for Question 2
2	Possible Answer B for Question 0	Possible Answer B for Question 1	Possible Answer B for Question 2
3	Possible Answer C for Question 0	Possible Answer C for Question 1	Possible Answer C for Question 2
4	Possible Answer D for Question 0	Possible Answer D for Question 1	Possible Answer D for Question 2

Of course you can extend this array if you create further questions.

The answers array will then be defined something like this:

Index	Value
0	Correct answer to Question 0
1	Correct answer to Question 1
2	Correct answer to Question 2

Again, you can extend this array as you add more questions.

Now that you have an idea of how you are going to store the question data, let’s have a look at the code. The name of the page to add the code to is `trivia_quiz.htm`. You start by creating the HTML tags at the top of the page.

```
<html>
<head>
<title>Wrox Online Trivia Quiz</title>
</head>
<body>
```

Chapter 2: Data Types and Variables

Then, in the body of the page, you start a JavaScript block in which you declare two variables, `questions` and `answers`, and define them as new arrays. The purpose of these variables should be pretty self-explanatory! However, as in the rest of the code, you add comments so that it is easy to work out what you are doing.

```
<script language="JavaScript" type="text/javascript">

// questions and answers arrays will holds questions and answers
var questions = new Array();
var answers = new Array();
```

Next you move straight on to define our first question. Since the questions will be in a two-dimensional array, your first task is to set `questions[0]` to a new array. You assign the first element in this array, `questions[0][0]`, to the text of the question, and the following elements to the possible answers.

```
// define question 1
questions[0] = new Array();

// the question
questions[0][0] = "The Beatles were";

// first choice
questions[0][1] = "A sixties rock group from Liverpool";

// second choice
questions[0][2] = "Four musically gifted insects";

// third choice
questions[0][3] = "German cars";

// fourth choice
questions[0][4] = "I don't know. Can I have the questions on baseball please?";
```

Having defined the first question, let's set the first answer. For multiple-choice questions you need only to set the element with the corresponding index in the `answers` array to the character representing the correct choice. In the previous question the correct answer is "A sixties rock group from Liverpool". As this is the first choice, its letter is A.

```
// assign answer for question 1
answers[0] = "A";
```

Let's define two more questions for the quiz. They both take the same format as the first question.

```
// define question 2
questions[1] = new Array();
questions[1][0] = "Homer Simpson's favorite food is";
questions[1][1] = "Fresh salad";
questions[1][2] = "Doughnuts";
questions[1][3] = "Bread and water";
questions[1][4] = "Apples";

// assign answer for question 2
```



```
answers[1] = "B";

// define question 3
questions[2] = new Array();
questions[2][0] = "Lisa Simpson plays which musical instrument?";
questions[2][1] = "Clarinet";
questions[2][2] = "Oboe";
questions[2][3] = "Saxophone";
questions[2][4] = "Tubular bells";

// assign answer for question 3
answers[2] = "C";
```

You end the script block by creating an alert box that tells you that the array has been initialized.

```
alert("Array Initialized");

</script>
</body>
</html>
```

Save the page as `trivia_quiz.htm`. That completes the definition of your quiz's questions and answers. In the next chapter you can move on to writing code that checks the correct answers to the questions against the answers supplied by the user.

Summary

In this chapter you have built up knowledge of the fundamentals of JavaScript's data types and variables, and how to use them in operations. In particular, you saw that

- ☐ JavaScript supports a number of types of data, such as numbers, text, and Booleans.
- ☐ Text is represented by strings of characters and is surrounded by quotes. You must match the quotes surrounding strings. Escape characters enable you to include characters in your string that cannot be typed.
- ☐ Variables are JavaScript's means of storing data, such as numbers and text, in memory so that they can be used again and again in your code.
- ☐ Variable names must not include certain illegal characters, like the percent sign (%) and the ampersand (&), or be a reserved word, like `with`.
- ☐ Before you can give a value to a variable, you must declare its existence to the JavaScript interpreter.
- ☐ JavaScript has the four basic math operators, represented by the symbols plus (+), minus (-), star (*), and forward slash (/). To assign values of a calculation to a variable, you use the equals sign (=), termed the assignment operator.
- ☐ Operators have different levels of precedence, so multiplication and division will be calculated before addition and subtraction.

Chapter 2: Data Types and Variables

- ❑ Strings can be joined together, or concatenated, to produce one big string by means of the + operator. When numbers and strings are concatenated with the + operator, JavaScript automatically converts the number into a string.
- ❑ Although JavaScript's automatic data conversion suits us most of the time, there are occasions when you need to force the conversion of data. You saw how `parseInt()` and `parseFloat()` can be used to convert strings to numbers. Attempting to convert strings that won't convert will result in `NaN` (Not a Number) being returned.
- ❑ Arrays are a special type of variable that can hold more than one piece of data. The data are inserted and accessed by means of a unique index number.

Exercise Questions

Suggested solutions to these questions can be found in Appendix A.

Question 1

Write a JavaScript program to convert degrees centigrade into degrees Fahrenheit, and to write the result to the page in a descriptive sentence. The JavaScript equation for Fahrenheit to centigrade is as follows:

$$\text{degFahren} = 9 / 5 * \text{degCent} + 32$$

Question 2

The following code uses the `prompt()` function to get two numbers from the user. It then adds those two numbers together and writes the result to the page:

```
<html>
<body>
<script language="JavaScript" type="text/javascript">

var firstNumber = prompt("Enter the first number","");
var secondNumber = prompt("Enter the second number","");
var theTotal = firstNumber + secondNumber;
document.write(firstNumber + " added to " + secondNumber + " equals " +
    theTotal);

</script>
</body>
</html>
```

However, if you try the code out, you'll discover that it doesn't work. Why not?

Change the code so that it does work.

3

Decisions, Loops, and Functions

So far, you've seen how to use JavaScript to get user input, perform calculations and tasks with that input, and write the results to a web page. However, a pocket calculator can do all this, so what is it that makes computers different? That is to say, what gives computers the appearance of having intelligence? The answer is the capability to make decisions based on information gathered.

How will decision-making help you in creating web sites? In the last chapter you wrote some code that converted temperature in degrees Fahrenheit to centigrade. You obtained the degrees Fahrenheit from the user using the `prompt()` function. This worked fine if the user entered a valid number, such as 50. If, however, the user entered something invalid for the Fahrenheit temperature, such as the string `aaa`, you would find that your code no longer works as expected. Now, if you had some decision-making capabilities in your program, you could check to see if what the user has entered is valid. If it is, you can do the calculation, and if it isn't, you can tell the user why and ask him to enter a valid number.

Validation of user input is probably one of the most common uses of decision making in JavaScript, but it's far from being the only use. The trivia quiz also needs some decision-making capabilities so that you can check if the answer given by the user is right or wrong. If it's right, you need to take certain steps, such as telling the user that she is right and increasing her score. If the answer is wrong, a different set of code needs to be executed to tell her that she's wrong.

In this chapter you'll look at how decision making is implemented in JavaScript and how you can use it to make your code smarter.

Decision Making — The if and switch Statements

All programming languages enable you to make decisions — that is, they enable the program to follow a certain course of action depending on whether a particular *condition* is met. This is what gives programming languages their intelligence.

For example, in a situation in which you use JavaScript code that is compatible only with version 4 or later browsers, the condition could be that the user is using a version 4 or later browser. If you discover that this condition is not met, you could direct him to a set of pages that are compatible with earlier browsers.

Conditions are comparisons between variables and data, such as the following:

- ☐ Is *A* bigger than *B*?
- ☐ Is *X* equal to *Y*?
- ☐ Is *M* not equal to *N*?

For example, if the variable `browserVersion` held the version of the browser that the user was using, the condition would be this:

- ☐ Is `browserVersion` greater than or equal to 4?

You'll notice that all of these questions have a yes or no answer — that is, they are Boolean based and can only evaluate to `true` or `false`. How do you use this to create decision-making capabilities in your code? You get the browser to test for whether the condition is `true`. If (and only if) it is `true`, you execute a particular section of code.

Look at another example. Recall from Chapter 1 the natural English instructions used to demonstrate how code flows. One of these instructions for making a cup of coffee is:

- ☐ Has the kettle boiled? If so, then pour water into cup; otherwise, continue to wait.

This is an example of making a decision. The condition in this instruction is “Has the kettle boiled?” It has a `true` or `false` answer. If the answer is `true`, you pour the water into the cup. If it isn't `true`, you continue to wait.

In JavaScript, you can change the flow of the code's execution depending on whether a condition is `true` or `false`, using an `if` statement or a `switch` statement. You will look at these shortly, but first we need to introduce some new operators that are essential for the definition of conditions — *comparison operators*.

Comparison Operators

In the last chapter you saw how mathematical functions, such as addition and division, were represented by symbols, such as plus (+) and forward slash (/), called operators. You also saw that if you want to give a variable a value, you can assign to it a value or the result of a calculation using the equals sign (=), termed the assignment operator.

Decision making also has its own operators, which enable you to test conditions. Comparison operators, just like the mathematical operators you saw in the last chapter, have a left-hand side (LHS) and a right-hand side (RHS), and the comparison is made between the two. The technical terms for these are the *left operand* and the *right operand*. For example, the less-than operator, with the symbol `<`, is a comparison operator. You could write `23 < 45`, which translates as “Is 23 less than 45?” Here, the answer would be `true` (see Figure 3-1).

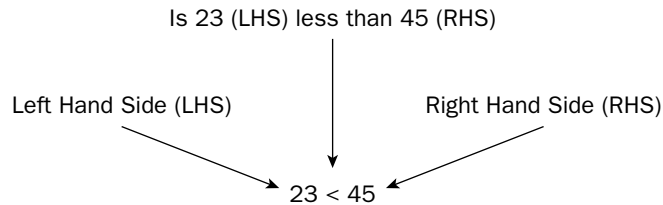


Figure 3-1

There are other comparison operators, the more useful of which are summarized in the following table:

Operator Symbol	Purpose
<code>==</code>	Tests if LHS is equal to RHS
<code><</code>	Tests if LHS is less than RHS
<code>></code>	Tests if LHS is greater than RHS
<code><=</code>	Tests if LHS is less than or equal to RHS
<code>>=</code>	Tests if LHS is greater than or equal to RHS
<code>!=</code>	Tests if LHS is not equal to RHS

You’ll see these comparison operators in use in the next section when you look at the `if` statement.

Precedence

Recall from Chapter 2 that operators have an order of precedence. This applies also to the comparison operators. The `==` and `!=` comparison operators have the lowest order of precedence, and the rest of the comparison operators, `<`, `>`, `<=`, and `>=`, have an equal precedence.

All of these comparison operators have a precedence that is below operators, such as `+`, `-`, `*`, and `/`. This means that if you make a comparison such as `3 * 5 > 2 * 5`, the multiplication calculations are worked out first, before their results are compared. However, in these circumstances, it’s both safer and clearer if you wrap the calculations on either side inside parentheses, for example, `(3 * 5) > (2 * 5)`. As a general rule, it’s a good idea to use parentheses to ensure that the precedence is clear, or you may find yourself surprised by the outcome.

Assignment versus Comparison

One very important point to mention is the ease with which the assignment operator (=) and the comparison operator (==) can be mixed up. Remember that the = operator assigns a value to a variable and that the == operator compares the value of two variables. Even when you have this idea clear, it's amazingly easy to put one equals sign where you meant to put two.

Assigning the Results of Comparisons

You can store the results of a comparison in a variable, as shown in the following example:

```
var age = prompt("Enter age:", "");
var isOverSixty = parseInt(age) > 60;
document.write("Older than 60: " + isOverSixty);
```

Here you obtain the user's age using the `prompt()` function. This returns, as a string, whatever value the user enters. You then convert that to a number using the `parseInt()` function you saw in the previous chapter and use the greater-than operator to see if it's greater than 60. The result (either true or false) of the comparison will be stored in the variable `isOverSixty`.

If the user enters 35, the `document.write()` on the final line will write this to the page:

```
Older than 60: false
```

If the user entered 61, this will be displayed:

```
Older than 60: true
```

The if Statement

The `if` statement is one you'll find yourself using in almost every program that is more than a couple of lines long. It works very much as it does in the English language. For example, you might say in English, "If the room temperature is more than 80 degrees Fahrenheit, then I'll turn the air conditioning on." In JavaScript, this would translate into something like this:

```
if (roomTemperature > 80)
{
    roomTemperature = roomTemperature - 10;
}
```

How does this work? See Figure 3-2.

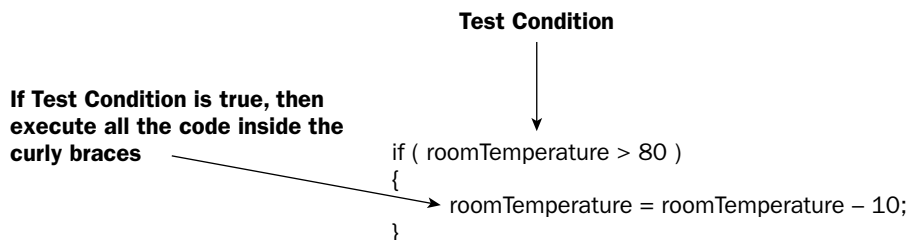


Figure 3-2

Notice that the test condition is placed in parentheses and follows the `if` keyword. Also, note that there is no semicolon at the end of this line. The code to be executed if the condition is `true` is placed in curly braces on the line after the condition, and each of these lines of code does end with a semicolon.

The curly braces, `{ }`, have a special purpose in JavaScript: They mark out a *block* of code. Marking out lines of code as belonging to a single block means that JavaScript will treat them all as one piece of code. If the condition of an `if` statement is `true`, JavaScript executes the next line or block of code following the `if` statement. In the preceding example, the block of code has only one statement, so we could equally as well have written this:

```
if (roomTemperature > 80)
    roomTemperature = roomTemperature - 10;
```

However, if you have a number of lines of code that you want to execute, you need the braces to mark them out as a single block of code. For example, a modified version of the example with three lines of code would have to include the braces.

```
if (roomTemperature > 80)
{
    roomTemperature = roomTemperature - 10;
    alert("It's getting hot in here");
    alert("Air conditioning switched on");
}
```

A particularly easy mistake to make is to forget the braces when marking out a block of code to be executed. Instead of the code in the block being executed when the condition is true, you'll find that *only the first line* after the `if` statement is executed. However, the other lines will always be executed regardless of the outcome of the test condition. To avoid mistakes like these, it's a good idea to always use braces, even where there is only one statement. If you get into this habit, you'll be less likely to leave them out when they are actually needed.

Try It Out The if Statement

Let's return to our temperature converter example from Chapter 2 and add some decision-making functionality. Enter the following code and save it as `ch3_examp1.htm`:

```
<html>
<body>

<script language="JavaScript" type="text/javascript">

var degFahren = Number(prompt("Enter the degrees Fahrenheit",32));
var degCent;

degCent = 5/9 * (degFahren - 32);

document.write(degFahren + "\xB0 Fahrenheit is " + degCent +
    "\xB0 centigrade<BR>");

if (degCent < 0)
{
```

Chapter 3: Decisions, Loops, and Functions

```
document.write("That's below the freezing point of water");
}

if (degCent == 100)
    document.write("That's the boiling point of water");

</script>

</body>
</html>
```

Load the page into your browser and enter 32 into the prompt box for the Fahrenheit value to be converted. With a value of 32, neither of the `if` statement's conditions will be true, so the only line written in the page will be that shown in Figure 3-3.

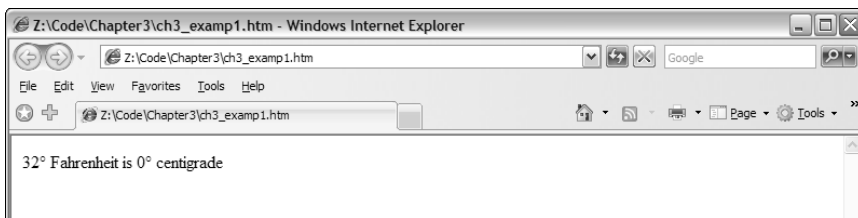


Figure 3-3

Now reload the page and enter 31 for the Fahrenheit value. This time you'll see two lines in the page, as shown in Figure 3-4.

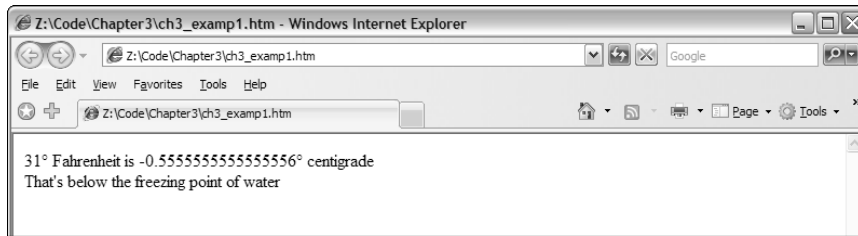


Figure 3-4

Finally, reload the page again, but this time, enter 212 in the prompt box. The two lines shown in Figure 3-5 will appear in the page.

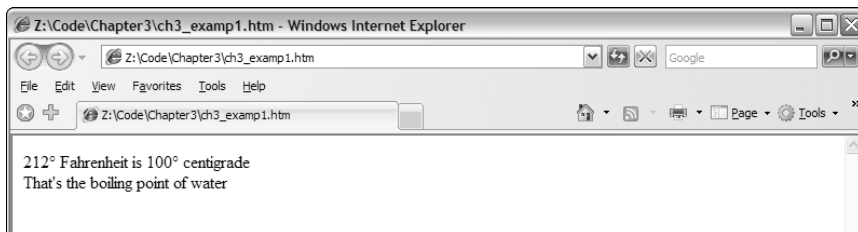


Figure 3-5

How It Works

The first part of the script block in this page is taken from the example `ch2_examp4.htm` in Chapter 2. You declare two variables, `degFahren` and `degCent`. The variable `degFahren` is given an initial value obtained from the user with the `prompt()` function. Note the `prompt()` function returns a string value, which you then explicitly convert to a numeric value using the `Number()` function. The variable `degCent` is then set to the result of the calculation $5/9 * (degFahren - 32)$, which is the Fahrenheit-to-centigrade conversion calculation.

```
var degFahren = Number(prompt("Enter the degrees Fahrenheit",32));
var degCent;

degCent = 5/9 * (degFahren - 32);
```

Then you write the result of your calculation to the page.

```
document.write(degFahren + "\xB0 Fahrenheit is " + degCent +
"\xB0 centigrade<BR>");
```

Now comes the new code; the first of two `if` statements.

```
if (degCent < 0)
{
    document.write("That's below the freezing point of water");
}
```

This `if` statement has the condition that asks, “Is the value of the variable `degCent` less than zero?” If the answer is yes (`true`), the code inside the curly braces executes. In this case, you write a sentence to the page using `document.write()`. If the answer is no (`false`), the processing moves on to the next line after the closing brace. Also worth noting is the fact that the code inside the `if` statement’s opening brace is indented. This is not necessary, but it is a good practice to get into because it makes your code much easier to read.

When trying out the example, you started by entering 32, so that `degFahren` will be initialized to 32. In this case the calculation `degCent = 5/9 * (degFahren - 32)` will set `degCent` to 0. So the answer to the question “Is `degCent` less than zero?” is `false`, because `degCent` is equal to zero, not less than zero. The code inside the curly braces will be skipped and never executed. In this case, the next line to be executed will be the second `if` statement’s condition, which we’ll discuss shortly.

When you entered 31 in the prompt box, `degFahren` was set to 31, so the variable `degCent` will be `-0.555555555556`. So how does your `if` statement look now? It evaluates to “Is `-0.555555555556` less than zero?” The answer this time is `true`, and the code inside the braces, here just a `document.write()` statement, executes.

Finally, when you entered 212, how did this alter the `if` statement? The variable `degCent` is set to 100 by the calculation, so the `if` statement now asks the question, “Is 100 less than zero?” The answer is `false`, and the code inside the braces will be skipped over.

In the second `if` statement, you evaluate the condition “Is the value of variable `degCent` equal to 100?”

```
if (degCent == 100)
    document.write("That's the boiling point of water");
```

Chapter 3: Decisions, Loops, and Functions

There are no braces here, so if the condition is `true`, the only code to execute is the first line below the `if` statement. When you want to execute multiple lines in the case of the condition being `true`, then braces are required.

You saw that when `degFahren` is 32, `degCent` will be 0. So your `if` statement will be “Is 0 equal to 100?” The answer is clearly `false`, and the code won’t execute. Again, when you set `degFahren` to 31, `degCent` will be calculated to be `-0.555555555556`; “Is `-0.555555555556` equal to 100?” is also `false`, and the code won’t execute.

Finally, when `degFahren` is set to 212, `degCent` will be 100. This time the `if` statement is “Is 100 equal to 100?” and the answer is `true`, so the `document.write()` statement executes.

As you have seen already, one of the most common errors in JavaScript, even for experts, is using one equals sign for evaluating, rather than the necessary two. Take a look at the following code extract:

```
if (degCent = 100)
    document.write("That's the boiling point of water");
```

This condition will always evaluate to `true`, and the code below the `if` statement will always execute. Worse still, your variable `degCent` will be set to 100. Why? Because a single equals sign assigns values to a variable; only a double equals sign compares values. The reason an assignment always evaluates to `true` is that the result of the assignment expression is the value of the right-hand side expression and this is the number 100, which is then implicitly converted to a Boolean and any number besides 0 and NaN converts to `true`.

Logical Operators

You should have a general idea of how to use conditions in `if` statements now, but how do you use a condition such as “Is `degFahren` greater than zero, but less than 100?” There are two conditions to test here. You need to test whether `degFahren` is greater than zero *and* whether `degFahren` is less than 100.

JavaScript enables you to use such multiple conditions. To do this you need to learn about three more operators, the logical operators AND, OR, and NOT. The symbols for these are listed in the following table.

Operator	Symbol
AND	&&
OR	
NOT	!

Notice that the AND and OR operators are *two* symbols repeated: `&&` and `||`. If you type just one symbol, `&` or `|`, strange things will happen because these are special operators called *bitwise operators* used in binary operations — for logical operations you must always use two.

After you’ve learned about the three logical operators, you’ll take a look at how to use them in `if` statements, with plenty of practical examples. So if it seems a bit confusing on first read, don’t panic. All will become clear. Let’s look at how each of these works, starting with the AND operator.

AND

Recall that we talked about the left-hand side (LHS) and the right-hand side (RHS) of the operator. The same is true with the AND operator. However, now the LHS and RHS of the condition are Boolean values (usually the result of a condition).

The AND operator works very much as it does in English. For example, you might say, “If I feel cold *and* I have a coat, then I’ll put my coat on.” Here, the left-hand side of the “and” word is “Do I feel cold?” and this can be evaluated as `true` or `false`. The right-hand side is “Do I have a coat?” which again is evaluated to either `true` or `false`. If the left-hand side is `true` (I am cold) *and* the right-hand side is `true` (I do have a coat), then you put your coat on.

This is very similar to how the AND operator works in JavaScript. The AND operator actually produces a result, just as adding two numbers together produces a result. However, the AND operator takes two Boolean values (on its LHS and RHS) and results in another Boolean value. If the LHS and RHS conditions evaluate to `true`, the result will be `true`. In any other circumstance, the result will be `false`.

Following is a *truth table* of possible evaluations of left-hand sides and right-hand sides and the result when AND is used.

Left-Hand Side	Right-Hand Side	Result
<code>true</code>	<code>true</code>	<code>true</code>
<code>false</code>	<code>true</code>	<code>false</code>
<code>true</code>	<code>false</code>	<code>false</code>
<code>false</code>	<code>false</code>	<code>false</code>

Although the table is, strictly speaking, true, it’s worth noting that JavaScript doesn’t like doing unnecessary work. Well, who does! If the left-hand side is `false`, then even if the right-hand side does evaluate to `true` it won’t make any difference to the final result — it’ll still be `false`. So to avoid wasting time, if the left-hand side is `false`, JavaScript doesn’t even bother checking the right-hand side and just returns a result of `false`.

OR

Just like AND, OR also works much as it does in English. For example, you might say that if it is raining *or* if it is snowing, then you’ll take an umbrella. If either of the conditions “it is raining” or “it is snowing” is `true`, then you will take an umbrella.

Again, just like AND, the OR operator acts on two Boolean values (one from its left-hand side and one from its right-hand side) and returns another Boolean value. If the left-hand side evaluates to `true` or the right-hand side evaluates to `true`, the result returned is `true`. Otherwise, the result is `false`. The following table shows the possible results.

Left-Hand Side	Right-Hand Side	Result
true	true	true
false	true	true
true	false	true
false	false	false

As with the AND operator, JavaScript likes to avoid doing things that make no difference to the final result. If the left-hand side is `true`, then whether the right-hand side is `true` or `false` makes no difference to the final result—it'll still be `true`. So, to avoid work, if the left-hand side is `true`, the right-hand side is not evaluated, and JavaScript simply returns `true`. The end result is the same—the only difference is in how JavaScript arrives at the conclusion. However, it does mean you should not rely on the right-hand side of the OR operator to be executed.

NOT

In English, we might say, “If I’m *not* hot, then I’ll eat soup.” The condition being evaluated is whether we’re hot. The result is `true` or `false`, but in this example we act (eat soup) if the result is `false`.

However, JavaScript is used to executing code only if a condition is `true`. So if you want a `false` condition to cause code to execute, you need to switch that `false` value to `true` (and any `true` value to `false`). That way you can trick JavaScript into executing code after a `false` condition.

You do this using the NOT operator. This operator reverses the logic of a result; it takes one Boolean value and changes it to the other Boolean value. So it changes `true` to `false` and `false` to `true`. This is sometimes called *negation*.

To use the NOT operator, you put the condition you want reversed in parentheses and put the `!` symbol in front of the parentheses. For example:

```
if (!(degCent < 100))
{
    // Some code
}
```

Any code within the braces will be executed only if the condition `degCent < 100` is `false`.

The following table details the possible results when using NOT.

Right-Hand Side	Result
true	false
false	true

Multiple Conditions Inside an if Statement

The previous section started by asking how you could use the condition “Is `degFahren` greater than zero, but less than 100?” One way of doing this would be to use two `if` statements, one nested inside another. *Nested* simply means that there is an outer `if` statement, and inside this an inner `if` statement. If the condition for the outer `if` statement is `true`, then (and only then) the nested inner `if` statement’s condition will be tested.

Using nested `if` statements, your code would be:

```
if (degCent < 100)
{
    if (degCent > 0)
    {
        document.write("degCent is between 0 and 100");
    }
}
```

This would work, but it’s a little verbose and can be quite confusing. JavaScript offers a better alternative — using multiple conditions inside the condition part of the `if` statement. The multiple conditions are strung together with the logical operators you just looked at. So the preceding code could be rewritten like this:

```
if (degCent > 0 && degCent < 100)
{
    document.write("degCent is between 0 and 100");
}
```

The `if` statement’s condition first evaluates whether `degCent` is greater than zero. If that is `true`, the code goes on to evaluate whether `degCent` is less than 100. Only if both of these conditions are `true` will the `document.write()` code line execute.

Try It Out Multiple Conditions

This example demonstrates multi-condition `if` statements using the AND, OR, and NOT operators. Type in the following code, and save it as `ch3_examp2.htm`:

```
<html>
<body>

<script language="JavaScript" type="text/javascript">

var myAge = Number(prompt("Enter your age",30));

if (myAge >= 0 && myAge <= 10)
{
    document.write("myAge is between 0 and 10<br>");
}

if ( !(myAge >= 0 && myAge <= 10) )
{
```

Chapter 3: Decisions, Loops, and Functions

```
document.write("myAge is NOT between 0 and 10<br>");  
}  
  
if ( myAge >= 80 || myAge <= 10 )  
{  
    document.write("myAge is 80 or above OR 10 or below<br>");  
}  
  
if ( (myAge >= 30 && myAge <= 39) || (myAge >= 80 && myAge <= 89) )  
{  
    document.write("myAge is between 30 and 39 or myAge is between 80 and 89");  
}  
  
</script>  
  
</body>  
</html>
```

When you load it into your browser, you should see a prompt box appear. Enter the value 30, then press Return, and the lines shown in Figure 3-6 are written to the web page.

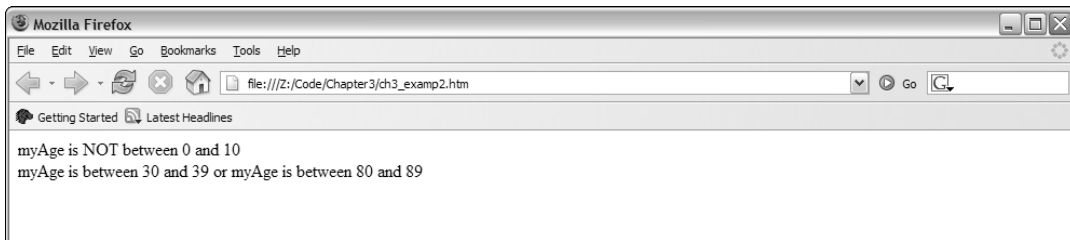


Figure 3-6

How It Works

The script block starts by defining the variable `myAge` and initializing it to the value entered by the user in the prompt box and converted to a number.

```
var myAge = Number(prompt("Enter your age",30));
```

After this are four `if` statements, each using multiple conditions. You'll look at each in detail in turn.

The easiest way to work out what multiple conditions are doing is to split them up into smaller pieces and then evaluate the combined result. In this example you have entered the value 30, which has been stored in the variable `myAge`. You'll substitute this value into the conditions to see how they work.

Here's the first `if` statement:

```
if (myAge >= 0 && myAge <= 10)  
{  
    document.write("myAge is between 0 and 10<BR>");  
}
```

The first `if` statement is asking the question “Is `myAge` between 0 and 10?” You’ll take the LHS of the condition first, substituting your particular value for `myAge`. The LHS asks “Is 30 greater than or equal to 0?” The answer is `true`. The question posed by the RHS condition is “Is 30 less than or equal to 10?” The answer is `false`. These two halves of the condition are joined using `&&`, which indicates the AND operator. Using the AND results table shown earlier, you can see that if LHS is `true` and RHS is `false`, you have an overall result of `false`. So the end result of the condition for the `if` statement is `false`, and the code inside the braces won’t execute.

Let’s move on to the second `if` statement.

```
if ( !(myAge >= 0 && myAge <= 10) )
{
    document.write("myAge is NOT between 0 and 10<BR>");
}
```

The second `if` statement is posing the question “Is `myAge` not between 0 and 10?” Its condition is similar to that of the first `if` statement, but with one small difference: You have enclosed the condition inside parentheses and put the NOT operator (`!`) in front.

The part of the condition inside the parentheses is evaluated and, as before, produces the same result — `false`. However, the NOT operator reverses the result and makes it `true`. Because the `if` statement’s condition is `true`, the code inside the braces *will* execute this time, causing a `document.write()` to write a response to the page.

What about the third `if` statement?

```
if ( myAge >= 80 || myAge <= 10 )
{
    document.write("myAge is either 80 and above OR 10 or below<BR>");
}
```

The third `if` statement asks, “Is `myAge` greater than or equal to 80, or less than or equal to 10?” Taking the LHS condition first — “Is 30 greater than or equal to 80?” — the answer is `false`. The answer to the RHS condition — “Is 30 less than or equal to 10?” — is again `false`. These two halves of the condition are combined using `||`, which indicates the OR operator. Looking at the OR result table earlier in this section, you see that `false` OR `false` produces a result of `false`. So again the `if` statement’s condition evaluates to `false`, and the code within the curly braces does not execute.

The final `if` statement is a little more complex.

```
if ( (myAge >= 30 && myAge <= 39) || (myAge >= 80 && myAge <= 89) )
{
    document.write("myAge is between 30 and 39 " +
        "or myAge is between 80 and 89<BR>");
}
```

It asks the question, “Is `myAge` between 30 and 39 or between 80 and 89?” Let’s break the condition down into its component parts. There is a left-hand-side and a right-hand-side condition, combined by means of an OR operator. However, the LHS and RHS themselves have an LHS and RHS each, which

Chapter 3: Decisions, Loops, and Functions

are combined using AND operators. Notice how parentheses are used to tell JavaScript which parts of the condition to evaluate first, just as you would do with numbers in a mathematical calculation.

Let's look at the LHS of the condition first, namely `(myAge >= 30 && myAge <= 39)`. By putting the condition into parentheses, you ensure that it's treated as a single condition; no matter how many conditions are inside the parentheses, it only produces a single result, either `true` or `false`. Breaking down the conditions in the parentheses, you have "Is 30 greater than or equal to 30?" with a result of `true`, and "Is 30 less than or equal to 39?" again with a result of `true`. From the AND table, you know `true AND true` produces a result of `true`.

Now let's look at the RHS of the condition, namely `(myAge >= 80 && myAge <= 89)`. Again breaking the condition down, you see that the LHS asks, "Is 30 greater than or equal to 80?" which gives a `false` result, and the RHS asks, "Is 30 less than or equal to 89?" which gives a `true` result. You know that `false AND true` gives a `false` result.

Now you can think of your `if` statement's condition as looking like `(true || false)`. Looking at the OR results table, you can see that `true OR false` gives a result of `true`, so the code within the braces following the `if` statement will execute, and a line will be written to the page.

However, remember that JavaScript does not evaluate conditions where they won't affect the final result, and the preceding condition is one of those situations. The LHS of the condition evaluated to `true`. After that, it does not matter if the RHS of the condition is `true` or `false` because only one of the conditions in an OR operation needs to be `true` for a result of `true`. Thus JavaScript does not actually evaluate the RHS of the condition. We did so simply for demonstration purposes.

As you have seen, the easiest way to approach understanding or creating multiple conditions is to break them down into the smallest logical chunks. You'll find that with experience, you will do this almost without thinking, unless you have a particularly tricky condition to evaluate.

Although using multiple conditions is often better than using multiple `if` statements, there are times when it makes your code harder to read and therefore harder to understand and debug. It's possible to have 10, 20, or more than 100 conditions inside your `if` statement, but can you imagine trying to read an `if` statement with even 10 conditions? If you feel that your multiple conditions are getting too complex, break them down into smaller logical chunks.

For example, imagine you want to execute some code if `myAge` is in the ranges 30–39, 80–89, or 100–115, using different code in each case. You could write the statement like so:

```
if ( (myAge >= 30 && myAge <= 39) || (myAge >= 80 && myAge <= 89) ||
    (myAge >= 100 && myAge <= 115) )
{
    document.write("myAge is between 30 and 39 " +
        "or myAge is between 80 " +
        "and 89 or myAge is between 100 and 115");
}
```

There's nothing wrong with this, but it is starting to get a little long and difficult to read. Instead you could create another `if` statement for the code executed for the 100–115 range.

else and else if

Imagine a situation where you want some code to execute if a certain condition is true, and some other code to execute if it is false. You can achieve this by having two `if` statements, as shown in the following example:

```
if (myAge >= 0 && myAge <= 10)
{
    document.write("myAge is between 0 and 10");
}

if ( !(myAge >= 0 && myAge <= 10) )
{
    document.write("myAge is NOT between 0 and 10");
}
```

The first `if` statement tests whether `myAge` is between 0 and 10, and the second for the situation where `myAge` is not between 0 and 10. However, JavaScript provides an easier way of achieving this: with an `else` statement. Again, the use of the word `else` is similar to its use in the English language. You might say, “If it is raining, I will take an umbrella; otherwise I will take a sun hat.” In JavaScript you can say `if` the condition is true, then execute one block of code; `else` execute an alternative block. Rewriting the preceding code using this technique, you would have the following:

```
if (myAge >= 0 && myAge <= 10)
{
    document.write("myAge is between 0 and 10");
}
else
{
    document.write("myAge is NOT between 0 and 10");
}
```

Writing the code like this makes it simpler and therefore easier to read. Plus it also saves JavaScript from testing a condition to which you already know the answer.

You could also include another `if` statement with the `else` statement. For example

```
if (myAge >= 0 && myAge <= 10)
{
    document.write("myAge is between 0 and 10");
}
else if ( (myAge >= 30 && myAge <= 39) || (myAge >= 80 && myAge <= 89) )
{
    document.write("myAge is between 30 and 39 " +
        "or myAge is between 80 and 89");
}
else
{
    document.write("myAge is NOT between 0 and 10, " +
        "nor is it between 30 and 39, nor is it between 80 and 89");
}
```

Chapter 3: Decisions, Loops, and Functions

The first `if` statement checks whether `myAge` is between 0 and 10 and executes some code if that's true. If it's false, an `else if` statement checks if `myAge` is between 30 and 39 or 80 and 89, and executes some other code if either of those conditions is true. Failing that, you have a final `else` statement, which catches the situation in which the value of `myAge` did not trigger `true` in any of the earlier `if` conditions.

When using `if` and `else if`, you need to be extra careful with your curly braces to ensure that the `if` and `else if` statements start and stop where you expect, and you don't end up with an `else` that doesn't belong to the right `if`. This is quite tricky to describe with words—it's easier to see what we mean with an example.

```
if (myAge >= 0 && myAge <= 10)
{
  document.write("myAge is between 0 and 10");
  if (myAge == 5)
  {
    document.write("You're 5 years old");
  }
  else
  {
    document.write("myAge is NOT between 0 and 10");
  }
}
```

Notice that we haven't indented the code. Although this does not matter to JavaScript, it does make the code more difficult for humans to read and hides the missing curly brace that should be before the final `else` statement.

Correctly formatted and with the missing bracket inserted, the code looks like this:

```
if (myAge >= 0 && myAge <= 10)
{
  document.write("myAge is between 0 and 10<br>");
  if (myAge == 5)
  {
    document.write("You're 5 years old");
  }
}
else
{
  document.write("myAge is NOT between 0 and 10");
}
```

As you can see, the code is working now; it is also a lot easier to see which code is part of which `if` block.

Comparing Strings

Up to this point you have been looking exclusively at using comparison operators with numbers. However, they work just as well with strings. All that's been said and done with numbers applies to strings, but with one important difference. You are now comparing data alphabetically rather than numerically, so there are a few traps to watch out for.

In the following code, you compare the variable `myName`, which contains the string `"Paul"`, with the string literal `"Paul"`.

```
var myName = "Paul";
if (myName == "Paul")
{
    alert("myName is Paul");
}
```

How does JavaScript deal with this? Well, it goes through each letter in turn on the LHS and checks it with the letter in the same position on the RHS to see if it's actually the same. If at any point it finds a difference, it stops, and the result is `false`. If, after having checked each letter in turn all the way to the end, it confirms that they are all the same, it returns `true`. The condition in the preceding `if` statement will return `true`, and so you'll see an alert box.

However, string comparison in JavaScript is case sensitive. So `"P"` is not the same as `"p"`. Taking the preceding example, but changing the variable `myName` to `"paul"`, you find that the condition is `false` and the code inside the `if` statement does not execute.

```
var myName = "paul";
if (myName == "Paul")
{
    alert("myName is Paul");
}
```

The `>=`, `>`, `<=`, and `<` operators work with strings as well as with numbers, but again it is an alphabetical comparison. So `"A" < "B"` is `true`, because A comes before B in the alphabet. However, JavaScript's case sensitivity comes into play again. `"A" < "B"` is `true`, but `"a" < "B"` is `false`. Why? Because uppercase letters are treated as always coming *before* lowercase letters. Why is this? Each letter has a code number in the ASCII and Unicode character sets, and the code numbers for uppercase letters are lower than the code numbers for lowercase letters. This is something to watch out for when writing your own code.

The simplest way to avoid confusion with different cases is to convert both strings to either uppercase or lowercase before you compare them. You can do this easily using the `toUpperCase()` or `toLowerCase()` functions, which you'll learn about in the next chapter.

The switch Statement

You saw earlier how the `if` and `else if` statements could be used for checking various conditions; if the first condition is not valid, then another is checked, and another, and so on. However, when you want to check the value of a particular variable for a large number of possible values, there is a more efficient alternative, namely the `switch` statement. The structure of the `switch` statement is given in Figure 3-7.

The best way to think of the `switch` statement is "Switch to the code where the case matches." The `switch` statement has four important elements:

- ☐ The test expression
- ☐ The case statements
- ☐ The break statements
- ☐ The default statement

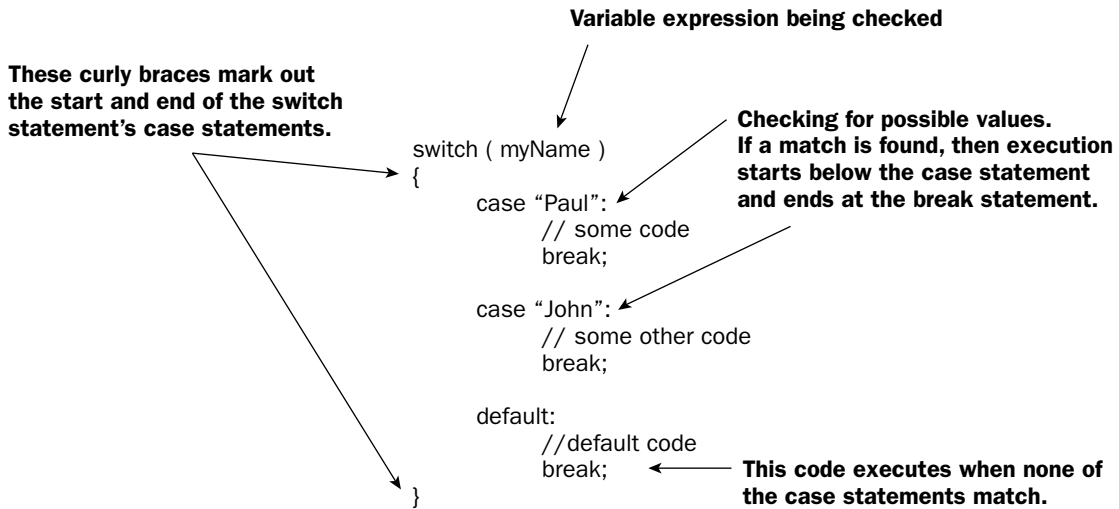


Figure 3-7

The test expression is given in the parentheses following the `switch` keyword. In the previous example you are testing using the variable `myName`. Inside the parentheses, however, you could have any valid expression.

Next come the `case` statements. It's the `case` statements that do the condition checking. To indicate which `case` statements belong to your `switch` statement, you must put them inside the curly braces following the test expression. Each `case` statement specifies a value, for example `"Paul"`. The `case` statement then acts like `if (myName == "Paul")`. If the variable `myName` did contain the value `"Paul"`, execution would commence from the code starting below the `case: "Paul"` statement and would continue to the end of the `switch` statement. This example has only two `case` statements, but you can have as many as you like.

In most cases, you want only the block of code directly underneath the relevant `case` statement to execute, and not *all* the code below the relevant `case` statement, including any other `case` statements. To achieve this, you put a `break` statement at the end of the code that you want executed. This tells JavaScript to stop executing at that point and leave the `switch` statement.

Finally you have the `default` case, which (as the name suggests) is the code that will execute when none of the other `case` statements match. The `default` statement is optional; if you have no default code that you want to execute, you can leave it out, but remember that in this case no code will execute if no `case` statements match. It is a good idea to include a `default` case, unless you are absolutely sure that you have all your options covered.

Try It Out Using the switch Statement

Let's take a look at the `switch` statement in action. The following example illustrates a simple guessing game. Type in the code and save it as `ch3_examp3.html`.

```
<html>
<body>

<script language="JavaScript" type="text/javascript">
```

```
var secretNumber = prompt("Pick a number between 1 and 5:", "");
secretNumber = parseInt(secretNumber);

switch (secretNumber)
{
case 1:
    document.write("Too low!");
    break;

case 2:
    document.write("Too low!");
    break;

case 3:
    document.write("You guessed the secret number!");
    break;

case 4:
    document.write("Too high!");
    break;

case 5:
    document.write("Too high!");
    break;

default:
    document.write("You did not enter a number between 1 and 5.");
    break;
}
document.write("<BR>Execution continues here");

</script>

</body>
</html>
```

Load this into your browser and enter, for example, the value 1 in the prompt box. You should then see something like what is shown in Figure 3-8.

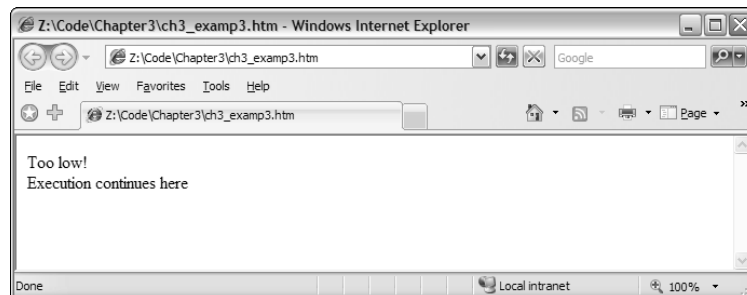


Figure 3-8

Chapter 3: Decisions, Loops, and Functions

If, on the other hand, you enter the value 3, you should see a friendly message letting you know that you guessed the secret number correctly, as shown in Figure 3-9.

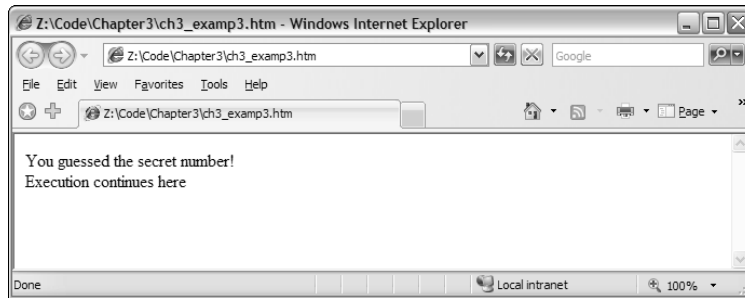


Figure 3-9

How It Works

First you declare the variable `secretNumber` and set it to the value entered by the user via the prompt box. Note that you use the `parseInt()` function to convert the string that is returned from `prompt()` to an integer value.

```
var secretNumber = prompt("Pick a number between 1 and 5:", "");
secretNumber = parseInt(secretNumber);
```

Next you create the start of the `switch` statement.

```
switch (secretNumber)
{
```

The expression in parentheses is simply the variable `secretNumber`, and it's this number that the `case` statements will be compared against.

You specify the block of code encompassing the `case` statements using curly braces. Each `case` statement checks one of the numbers between 1 and 5, because this is what you have specified to the user that she should enter. The first simply outputs a message that the number she has entered is too low.

```
case 1:
    document.write("Too low!");
    break;
```

The second `case` statement, for the value 2, has the same message, so the code is not repeated here. The third `case` statement lets the user know that she has guessed correctly.

```
case 3:
    document.write("You guessed the secret number!");
    break;
```

Finally, the fourth and fifth `case` statements output a message that the number the user has entered is too high.

```
case 4:
    document.write("Too high!");
    break;
```

You do need to add a `default` case in this example, since the user might very well (despite the instructions) enter a number that is not between 1 and 5, or even perhaps a letter. In this case you add a message to let the user know that there is a problem.

```
default:
    document.write("You did not enter a number between 1 and 5.");
    break;
```

A `default` statement is also very useful for picking up bugs—if you have coded some of the `case` statements incorrectly, you will pick that up very quickly if you see the `default` code being run when it shouldn't be.

You finally have added the closing brace indicating the end of the `switch` statement. After this you output a line to indicate where the execution continues.

```
}
document.write("<BR>Execution continues here");
```

Note that each `case` statement ends with a `break` statement. This is important to ensure that execution of the code moves to the line after the end of the `switch` statement. If you forget to include this, you could end up executing the code for each `case` following the `case` that matches.

Executing the Same Code for Different Cases

You may have spotted a problem with the `switch` statement in this example—you want to execute the same code if the user enters a 1 or a 2, and the same code for a 4 or a 5. However, in order to achieve this, you have had to repeat the code in each case. What you want is an easier way of getting JavaScript to execute the same code for different cases. Well, that's easy! Simply change the code so that it looks like this:

```
switch (secretNumber)
{
    case 1:
    case 2:
        document.write("Too low!");
        break;

    case 3:
        document.write("You guessed the secret number!");
        break;

    case 4:
    case 5:
        document.write("Too high!");
        break;

    default:
        document.write("You did not enter a number between 1 and 5.");
        break;
}
```

Chapter 3: Decisions, Loops, and Functions

If you load this into your browser and experiment with entering some different numbers, you should see that it behaves exactly like the previous code.

Here, you are making use of the fact that if there is no `break` statement underneath the code for a certain `case` statement, execution will continue through each following `case` statement until a `break` statement or the end of the `switch` is reached. Think of it as a sort of free fall through the `switch` statement until you hit the `break` statement.

If the `case` statement for the value 1 is matched, execution simply continues until the `break` statement under `case 2`, so effectively you can execute the same code for both cases. The same technique is used for the `case` statements with values 4 and 5.

Looping — The `for` and `while` Statements

Looping means repeating a block of code when a condition is `true`. This is achieved in JavaScript with the use of two statements, the `while` statement and the `for` statement. You'll be looking at these shortly, but why would you want to repeat blocks of code anyway?

Well, take the situation where you have a series of results, say the average temperature for each month in a year, and you want to plot these on a graph. The code needed for plotting each point will most likely be the same. So, rather than write out the code 12 times (once for each point), it's much easier to execute the same code 12 times by using the next item of data in the series. This is where the `for` statement would come in handy, because you know how many times you want the code to execute.

In another situation, you might want to repeat the same piece of code when a certain condition is `true`, for example, while the user keeps clicking a Start Again button. In this situation, the `while` statement would be very useful.

The `for` Loop

The `for` statement enables you to repeat a block of code a certain number of times. The syntax is illustrated in Figure 3-10.

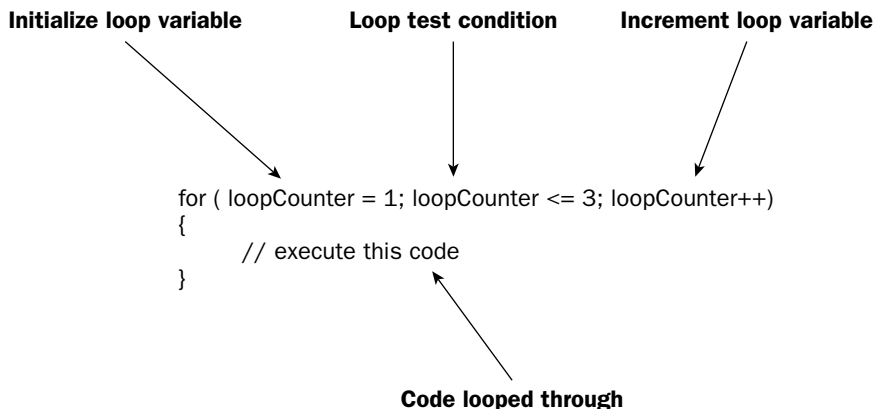


Figure 3-10

Let's look at the makeup of a `for` statement. You can see from Figure 3-10 that, just like the `if` and `switch` statements, the `for` statement also has its logic inside parentheses. However, this time that logic split into three parts, each part separated by a semicolon. For example, in Figure 3-10 you have the following:

```
(var loopCounter = 1; loopCounter <= 3; loopCounter++)
```

The first part of the `for` statement's logic is the *initialization* part of the `for` statement. To keep track of how many times you have looped through the code, you need a variable to keep count. It's in the initialization part that you initialize variables. In the example you have declared `loopCounter` and set it to the value of 1. This part is only executed once during the execution of the loops, unlike the other parts. You don't need to declare the variable if it was declared earlier in the code.

```
var loopCounter;  
for (loopCounter = 1; loopCounter <= 3; loopCounter++)
```

Following the semicolon, you have the *test condition* part of the `for` statement. The code inside the `for` statement will keep executing for as long as this test condition evaluates to `true`. After the code is looped through each time, this condition is tested. In Figure 3-10, you execute for as long as `loopCounter` is less than or equal to 3. The number of times a loop is performed is often called the number of *iterations*.

Finally, you have the *increment* part of the `for` loop, where variables in our loop's test condition have their values incremented. Here you can see that `loopCounter` is incremented by one by means of the `++` operator you saw in Chapter 2. Again, this part of the `for` statement is repeated with every loop of the code. Although we call it the increment part, it can actually be used to decrease or *decrement* the value—for example, if you wanted to count down from the top element in an array to the first.

After the `for` statement comes the block of code that will be executed repeatedly, as long as the test condition is `true`. This block of code is contained within curly braces. If the condition is never `true`, even at the first test of the loop condition, then the code inside the `for` loop will be skipped over and never executed.

Putting all this together, how does the `for` loop work?

1. Execute initialization part of the `for` statement.
2. Check the test condition. If `true`, continue; if not, exit the `for` statement.
3. Execute code in the block after the `for` statement.
4. Execute the increment part of the `for` statement.
5. Repeat steps 2 through 4 until the test condition is `false`.

Try It Out Converting a Series of Fahrenheit Values

Let's change the temperature converter so that it converts a series of values, stored in an array, from Fahrenheit to centigrade. You will be using the `for` statement to go through each element of the array. Type in the code and save it as `ch3_exam4.htm`.

```
<html>  
<body>  
  
<script language="JavaScript" type="text/javascript">
```

Chapter 3: Decisions, Loops, and Functions

```
var degFahren = new Array(212, 32, -459.15);
var degCent = new Array();
var loopCounter;

for (loopCounter = 0; loopCounter <= 2; loopCounter++)
{
    degCent[loopCounter] = 5/9 * (degFahren[loopCounter] - 32);
}

for (loopCounter = 2; loopCounter >= 0; loopCounter--)
{
    document.write("Value " + loopCounter + " was " + degFahren[loopCounter] +
        " degrees Fahrenheit");
    document.write(" which is " + degCent[loopCounter] +
        " degrees centigrade<BR>");
}

</script>

</body>
</html>
```

On loading this into your browser, you'll see a series of three lines in the page, containing the results of converting our array of Fahrenheit values into centigrade (as shown in Figure 3-11).

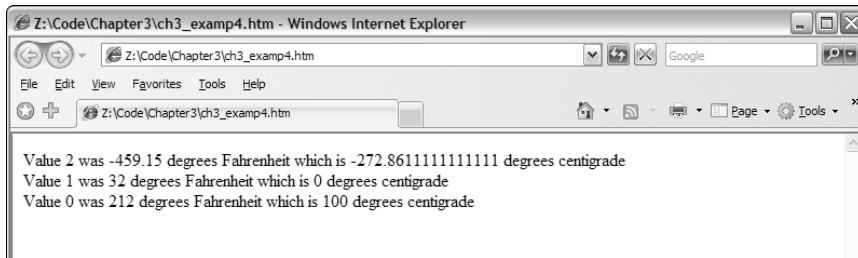


Figure 3-11

How It Works

The first task is to declare the variables you are going to use. First, you declare and initialize `degFahren` to contain an array of three values: 212, 32, and -459.15. Next, `degCent` is declared as an empty array. Finally, `loopCounter` is declared and will be used to keep track of which array index you are accessing during your looping.

```
var degFahren = new Array(212, 32, -459.15);
var degCent = new Array();
var loopCounter;
```

Following this comes our first `for` loop.

```
for (loopCounter = 0; loopCounter <= 2; loopCounter++)
{
    degCent[loopCounter] = 5/9 * (degFahren[loopCounter] - 32);
}
```

In the first line, you start by initializing the `loopCounter` to 0. Then the `for` loop's test condition, `loopCounter <= 2`, is checked. If this condition is `true`, the loop executes for the first time. After the code inside the curly braces has executed, the incrementing part of the `for` loop, `loopCounter++`, will be executed, and then the test condition will be re-evaluated. If it's still `true`, another execution of the loop code is performed. This continues until the `for` loop's test condition evaluates to `false`, at which point looping will end, and the first statement after the closing curly brace will be executed.

The code inside the curly braces is the equation you saw in earlier examples, only this time you are placing its result into the `degCent` array, with the index being the value of `loopCounter`.

In the second `for` loop, you write the results contained in the `degCent` array to the screen.

```
for (loopCounter = 2; loopCounter >= 0; loopCounter--)
{
    document.write("Value " + loopCounter + " was " + degFahren[loopCounter] +
        " degrees Fahrenheit");
    document.write(" which is " + degCent[loopCounter] +
        " degrees centigrade<BR>");
}
```

This time you're counting *down* from 2 to 0. The variable `loopCounter` is initialized to 2, and the loop condition remains `true` until `loopCounter` is less than 0. This time `loopCounter` is actually decremented each time rather than incremented, by means of `loopCounter--`. Again, `loopCounter` is serving a dual purpose: It keeps count of how many loops you have done and also provides the index position in the array.

Note that in these examples, you've used whole numbers in your loops. However, there is no reason why you can't use fractional numbers, although it's much less common to do so.

The `for...in` Loop

This loop works primarily with arrays, and as you'll see in the next chapter, it also works with something called objects. It enables you to loop through each element in the array without having to know how many elements the array actually contains. In plain English, what this loop says is "For each element in the array, execute some code." Rather than your having to work out the index number of each element, the `for...in` loop does it for you and automatically moves to the next index with each iteration (loop through).

Its syntax for use with arrays is:

```
for (index in arrayName)
{
    //some code
}
```

Chapter 3: Decisions, Loops, and Functions

In this code extract, `index` is a variable you declare prior to the loop, which will automatically be populated with the next index value in the array. `arrayName` is the name of the variable holding the array you want to loop through.

Let's look at an example to make things clearer. You'll define an array and initialize it with three values.

```
var myArray = new Array("Paul", "Paula", "Pauline");
```

To access each element using a conventional `for` loop, you'd write this:

```
var loopCounter;  
for (loopCounter = 0; loopCounter < 3; loopCounter++)  
{  
    document.write(myArray[loopCounter]);  
}
```

To do exactly the same thing with the `for...in` loop, you write this:

```
var elementIndex;  
for (elementIndex in myArray)  
{  
    document.write(myArray[elementIndex]);  
}
```

As you can see, the code in the second example is a little clearer, as well as being shorter. Both methods work equally well and will iterate three times. However, if you increase the size of the array, for example by adding the element `myArray[3] = "Philip"`, the first method will still loop only through the first three elements in the array, whereas the second method will loop through all four elements.

The while Loop

Whereas the `for` loop is used for looping a certain number of times, the `while` loop enables you to test a condition and keep on looping while it's true. The `for` loop is useful when you know how many times you need to loop, for example when you are looping through an array that you know has a certain number of elements. The `while` loop is more useful when you don't know how many times you'll need to loop. For example, if you are looping through an array of temperature values and want to continue looping when the temperature value contained in the array element is less than 100, you will need to use the `while` statement.

Let's take a look at the structure of the `while` statement, as illustrated in Figure 3-12.

You can see that the `while` loop has fewer parts to it than the `for` loop. The `while` loop consists of a condition which, if it evaluates to `true`, causes the block of code inside the curly braces to execute once; then the condition is re-evaluated. If it's still `true`, the code is executed again, the condition is re-evaluated again, and so on until the condition evaluates to `false`.

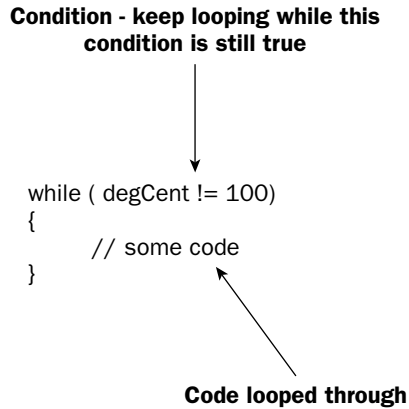


Figure 3-12

One thing to watch out for is that if the condition is `false` to start with, the `while` loop never executes. For example:

```
degCent = 100;

while (degCent != 100)
{
    // some code
}
```

Here, the loop will run if `degCent` does not equal 100. However, since `degCent` is 100, the condition is `false`, and the code never executes.

In practice you would normally expect the loop to execute once; whether it executes again will depend on what the code inside the loop has done to variables involved in the loop condition. For example:

```
degCent = new Array();
degFahren = new Array(34, 123, 212);
var loopCounter = 0;
while (loopCounter < 3)
{
    degCent[loopCounter] = 5/9 * (degFahren[loopCounter] - 32);
    loopCounter++;
}
```

The loop will execute so long as `loopCounter` is less than 3. It's the code inside the loop (`loopCounter++`;) that increments `loopCounter` and will eventually cause `loopCounter < 3` to be `false` so that the loop stops. Execution will then continue on the first line after the closing brace of the `while` statement.

Chapter 3: Decisions, Loops, and Functions

Something to watch out for is the *infinite loop* — a loop that will never end. Suppose you forgot to include the `loopCounter++;` line in the code. Leaving this line out would mean that `loopCounter` will remain at 0, so the condition `(loopCounter < 3)` will always be `true`, and the loop will continue until the user gets bored and cross, and shuts down her browser. However, it is an easy mistake to make and one JavaScript won't warn you about.

It's not just missing lines that can cause infinite loops, but also mistakes inside the loop's code. For example:

```
var testVariable = 0;
while (testVariable <= 10)
{
    alert("Test Variable is " + testVariable);
    testVariable++;
    if (testVariable = 10)
    {
        alert("The last loop");
    }
}
```

See if you can spot the deliberate mistake that leads to an infinite loop — yes, it's the `if` statement that will cause this code to go on forever. Instead of using `==` as the comparison operator in the condition of the `if` statement, you put `=`, so `testVariable` is set to 10 again in each loop, despite the line `testVariable++`. This means that at the start of each loop, the test condition always evaluates to `true`, since 10 is less than or equal to 10. Put the extra `=` in to make `if (testVariable == 10)`, and everything is fine.

The *do...while* loop

With the `while` loop, you saw that the code inside the loop only executes if the condition is `true`; if it's `false`, the code never executes, and execution instead moves to the first line after the `while` loop. However, there may be times when you want the code in the `while` loop to execute at least once, regardless of whether the condition in the `while` statement evaluates to `true`. It might even be that some code inside the `while` loop needs to be executed before you can test the `while` statement's condition. It's situations like this for which the `do...while` loop is ideal.

Look at an example in which you want to get the user's age via a prompt box. You want to show the prompt box but also make sure that what the user has entered is a number.

```
var userAge;
do
{
    userAge = prompt("Please enter your age", "");
}
while (isNaN(userAge) == true);
```

The code line within the loop —

```
userAge = prompt("Please enter your age", "");
```

— will be executed regardless of the `while` statement's condition. This is because the condition is not checked *until* one loop has been executed. If the condition is `true`, the code is looped through again. If it's `false`, then looping stops.

Note that within the `while` statement's condition, you are using the `isNaN()` function that you saw in Chapter 2. This checks whether the `userAge` variable's value is `NaN` (not a number). If it is not a number, the condition returns a value of `true`; otherwise it returns `false`. As you can see from the example, it enables you to test the user input to ensure the right data has been entered. The user might lie about his age, but at least you know he entered a number!

The `do...while` loop is fairly rare; there's not much you can't do without it, so it's best avoided unless really necessary.

The *break* and *continue* Statements

You met the `break` statement earlier when you looked at the `switch` statement. Its function inside a `switch` statement is to stop code execution and move execution to the next line of code after the closing curly brace of the `switch` statement. However, the `break` statement can also be used as part of the `for` and `while` loops when you want to exit the loop prematurely. For example, suppose you're looping through an array, as you did in the temperature conversion example, and you hit an invalid value. In this situation, you might want to stop the code in its tracks, notify the user that the data is invalid, and leave the loop. This is one situation where the `break` statement comes in handy.

Let's see how you could change the example where you converted a series of Fahrenheit values (`ch3_examp4.htm`) so that if you hit a value that's not a number you stop the loop and let the user know about the invalid data.

```
<script language="JavaScript" type="text/javascript">
var degFahren = new Array(212, "string data", -459.67);
var degCent = new Array();
var loopCounter;

for (loopCounter = 0; loopCounter <= 2; loopCounter++)
{
    if (isNaN(degFahren[loopCounter]))
    {
        alert("Data '" + degFahren[loopCounter] + "' at array index " +
            loopCounter + " is invalid");
        break;
    }

    degCent[loopCounter] = 5/9 * (degFahren[loopCounter] - 32);
}
```

You have changed the initialization of the `degFahren` array so that it now contains some invalid data. Then, inside the `for` loop, an `if` statement is added to check whether the data in the `degFahren` array is not a number. This is done by means of the `isNaN()` function; it returns `true` if the value passed to it in the parentheses, here `degFahren[loopCounter]`, is not a number. If the value is not a number, you tell the user where in the array you have the invalid data. Then you break out of the `for` loop altogether, using the `break` statement, and code execution continues on the first line after the end of the `for` statement.

That's the `break` statement, but what about `continue`? The `continue` statement is similar to `break` in that it stops the execution of a loop at the point where it is found, but instead of leaving the loop, it starts execution at the next iteration, starting with the `for` or `while` statement's condition being re-evaluated, just as if the last line of the loop's code had been reached.

Chapter 3: Decisions, Loops, and Functions

In the `break` example, it was all or nothing—if even one piece of data was invalid, you broke out of the loop. It might be better if you tried to convert all the values in `degFahren`, but if you hit an invalid item of data in the array, you notify the user and continue with the next item, rather than giving up as our `break` statement example does.

```
if (isNaN(degFahren[loopCounter]))
{
    alert("Data '" + degFahren[loopCounter] + "' at array index " +
        loopCounter + " is invalid");
    continue;
}
```

Just change the `break` statement to a `continue`. You will still get a message about the invalid data, but the third value will also be converted.

Functions

A function is something that performs a particular task. Take a pocket calculator as an example. It performs lots of basic calculations, such as addition and subtraction. However, many also have function keys that perform more complex operations. For example, some calculators have a button for calculating the square root of a number, and others even provide statistical functions, such as the calculation of an average. Most of these functions could be done with the basic mathematical operations of add, subtract, multiply, and divide, but that might take a lot of steps—it's much simpler for the user if she only needs to press one button. All she needs to do is provide the data—numbers in this case—and the function key does the rest.

Functions in JavaScript work a little like the function buttons on a pocket calculator: They encapsulate a block of code that performs a certain task. Over the course of the book so far, you have come across a number of handy built-in functions that perform a certain task, such as the `parseInt()` and `parseFloat()` functions, which convert strings to numbers, and the `isNaN()` function, which tells you whether a particular value can be converted to a number. Some of these functions return data, such as `parseInt()`, which returns an integer number; others simply perform an action, but return no data. You'll also notice that some functions can be passed data, whereas others cannot. For example, the `isNaN()` function needs to be passed some data, which it checks to see if it is `NaN`. The data that a function requires to be passed are known as its *parameter(s)*.

As you work your way through the book, you'll be coming across many more useful built-in functions, but wouldn't it be great to be able to write your own functions? After you've worked out, written, and debugged a block of code to perform a certain task, it would be nice to be able to call it again and again when you need it. JavaScript gives us the ability to do just that, and this is what you'll be concentrating on in this section.

Creating Your Own Functions

Creating and using your own functions is very simple. Figure 3-13 shows an example of a function.

You've probably already realized what this function does and how the code works. Yes, it's the infamous Fahrenheit-to-centigrade conversion code again.

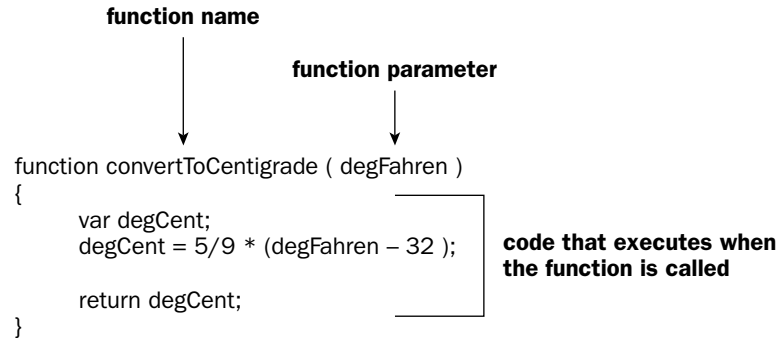


Figure 3-13

Each function you define in JavaScript must be given a unique name for that particular page. The name comes immediately after the `function` keyword. To make life easier for yourself, try using meaningful names so that when you see it being used later in your code, you'll know exactly what it does. For example, a function that takes as its parameters someone's birthday and today's date and returns the person's age could be called `getAge()`. However, the names you can use are limited, much as variable names are. For example, you can't use words reserved by JavaScript, so you can't call your function `with()` or `while()`.

The parameters for the function are given in parentheses after the function's name. A parameter is just an item of data that the function needs to be given in order to do its job. Usually, not passing the required parameters will result in an error. A function can have zero or more parameters, though even if it has no parameters you must still put the open and close parentheses after its name. For example, the top of your function definition must look like the following:

```
function myNoParamFunction()
```

You then write the code, which the function will execute when called on to do so. All the function code must be put in a block with a pair of curly braces.

Functions also give you the ability to return a value from a function to the code that called it. You use the `return` statement to return a value. In the example function given earlier, you return the value of the variable `degCent`, which you have just calculated. You don't have to return a value if you don't want to, but you should always include a `return` statement at the end of your function, although JavaScript is a very forgiving language and won't have a problem if you don't use a `return` statement at all.

When JavaScript comes across a `return` statement in a function, it treats it a bit like a `break` statement in a `for` loop—it exits the function, returning any value specified after the `return` keyword.

You'll probably find it useful to build up a "library" of functions that you use frequently in JavaScript code, which you can cut and paste into your page whenever you need them.

Having created your functions, how do you use them? Unlike the code you've seen so far, which executes when JavaScript reaches that line, functions only execute if you ask them to, which is termed *calling* or *invoking* the function. You call a function by writing its name at the point where you want it to be called and making sure that you pass any parameters it needs, separated by commas. For example:

```
myTemp = convertToCentigrade(212);
```

Chapter 3: Decisions, Loops, and Functions

This line calls the `convertToCentigrade()` function you saw earlier, passing 212 as the parameter and storing the return value from the function (that is, 100) in the `myTemp` variable.

Have a go at creating your own functions now, taking a closer look at how parameters are passed. Parameter passing can be a bit confusing, so you'll first create a simple function that takes just one parameter (the user's name) and writes it to the page in a friendly welcome string. First, you need to think of a name for your function. A short but descriptive name is `writeUserWelcome()`. Now you need to define what parameters the function expects to be passed. There's only one parameter—the user name. Defining parameters is a little like defining variables—you need to stick to the same rules for naming, so that means no spaces, special characters, or reserved words. Let's call your parameter `userName`. You need to add it inside parentheses to the end of the function name (note that you don't put a semicolon at the end of the line).

```
function writeUserWelcome(userName)
```

Okay, now you have defined your function name and its parameters; all that's left is to create the function body—that is, the code that will be executed when the function is called. You mark out this part of the function by wrapping it in curly braces.

```
function writeUserWelcome(userName)
{
    document.write("Welcome to my website " + userName + "<br>");
    document.write("Hope you enjoy it!");
}
```

The code is simple enough; you write out a message to the web page using `document.write()`. You can see that `userName` is used just as you'd use any normal variable; in fact, it's best to think of parameters as normal variables. The value that the parameter has will be that specified by the JavaScript code where the function was called.

Let's see how you would call this function.

```
writeUserWelcome("Paul");
```

Simple, really—just write the name of the function you want to call, and then in parentheses add the data to be passed to each of the parameters, here just one piece. When the code in the function is executed, the variable `userName`, used in the body of the function code, will contain the text "Paul".

Suppose you wanted to pass two parameters to your function—what would you need to change? Well, first you'd have to alter the function definition. Imagine that the second parameter will hold the user's age—you could call it `userAge` since that makes it pretty clear what the parameter's data represents. Here is the new code:

```
function writeUserWelcome(userName, userAge)
{
    document.write("Welcome to my website" + userName + "<br>");
    document.write("Hope you enjoy it<br>");
    document.write("Your age is " + userAge);
}
```

You've added a line to the body of the function that uses the parameter you have added. To call the function, you'd write the following:

```
writeUserWelcome("Paul",31);
```

The second parameter is a number, so there is no need for quotes around it. Here the `userName` parameter will be `Paul`, and the second parameter, `userAge`, will be `31`.

Try It Out Fahrenheit to Centigrade Function

Let's rewrite the temperature converter page using functions. You can cut and paste most of this code from `ch3_examp4.htm`—the parts that have changed have been highlighted. When you've finished, save it as `ch3_examp5.htm`.

```
<html>
<body>

<script language="JavaScript" type="text/javascript">

function convertToCentigrade(degFahren)
{
    var degCent;
    degCent = 5/9 * (degFahren - 32);

    return degCent;
}

var degFahren = new Array(212, 32, -459.15);
var degCent = new Array();
var loopCounter;

for (loopCounter = 0; loopCounter <= 2; loopCounter++)
{
    degCent[loopCounter] = convertToCentigrade(degFahren[loopCounter]);
}

for (loopCounter = 2; loopCounter >= 0; loopCounter--)
{
    document.write("Value " + loopCounter + " was " + degFahren[loopCounter] +
        " degrees Fahrenheit");
    document.write(" which is " + degCent[loopCounter] +
        " degrees centigrade<br>");
}

</script>

</body>
</html>
```

When you load this page into your browser, you should see exactly the same results that you had with `ch3_examp4.htm`.

How It Works

At the top of the script block you declare your `convertToCentigrade()` function. You saw this function earlier:

```
function convertToCentigrade(degFahren)
{
    var degCent;
    degCent = 5/9 * (degFahren - 32);

    return degCent;
}
```

If you're using a number of separate script blocks in a page, it's very important that the function be defined before any script calls it. If you have a number of functions, you may want to put them all in their own script block at the top of the page—between the `<head>` and `</head>` tags is good. That way you know where to find all your functions, and you can be sure that they have been declared before they have been used.

You should be pretty familiar with how the code in the function works. You declare a variable `degCent`, do your calculation, store its result in `degCent`, and then return `degCent` back to the calling code. The function's parameter is `degFahren`, which provides the information the calculation needs.

Following the function declaration is the code that executes when the page loads. First you define the variables you need, and then you have the two loops that calculate and then output the results. This is mostly the same as before, apart from the first `for` loop.

```
for (loopCounter = 0; loopCounter <= 2; loopCounter++)
{
    degCent[loopCounter] = convertToCentigrade(degFahren[loopCounter]);
}
```

The code inside the first `for` loop puts the value returned by the function `convertToCentigrade()` into the `degCent` array.

There is a subtle point to the code in this example. Notice that you declare the variable `degCent` within your function `convertToCentigrade()`, and you also declare it as an array after the function definition.

Surely this isn't allowed?

Well, this leads neatly to the next topic of this chapter—variable scope.

Variable Scope and Lifetime

What is meant by *scope*? Well, put simply, it's the scope or extent of a variable's availability—which parts of your code can access a variable and the data it contains. Any variables declared in a web page outside of a function will be available to all script on the page, whether that script is inside a function or otherwise—we term this a *global* or *page-level scope*. However, variables declared inside a function are visible *only* inside that function—no code outside the function can access them. So, for example, you

could declare a variable `degCent` in every function you have on a page *and* once on the page outside any function. However, you can't declare the variable *more* than once inside any one function or *more* than once on the page outside the functions. Note that reusing a variable name throughout a page in this way, although not illegal, is not standard good practice — it can make the code very confusing to read.

Function parameters are similar to variables: They can't be seen outside the function, and although you can declare a variable in a function with the same name as one of its parameters, it would cause a lot of confusion and might easily lead to subtle bugs being overlooked. It's therefore bad coding practice and best avoided, if only for the sake of your sanity when it comes to debugging!

So what happens when the code inside a function ends and execution returns to the point at which the code was called? Do the variables defined within the function retain their value when you call the function the next time?

The answer is no: Variables not only have the scope property — where they are visible — but they also have a *lifetime*. When the function finishes executing, the variables in that function die and their values are lost, unless you return one of them to the calling code. Every so often JavaScript performs garbage collection (which we talked about in Chapter 2), whereby it scans through the code and sees if any variables are no longer in use; if so, the data they hold are freed from memory to make way for the data of other variables.

Given that global variables can be used anywhere, why not make all of them global? Global variables are great when you need to keep track of data on a global basis. However, because they are available for modification anywhere in your code, it does mean that if they are changed incorrectly due to a bug, that bug could be anywhere within the code, making debugging difficult. It's best, therefore, to keep global variable use to a minimum, though sometimes they are a necessary evil — for example, when you need to share data among different functions.

The Trivia Quiz — Building One of the Basic Functions

In the previous chapter you declared the arrays that hold the questions and answers for your trivia quiz. You also populated these arrays with the first three questions. At that point you didn't have enough knowledge to actually make use of the data, but with what you've learned in this chapter, you can create a function that uses the information in the arrays to check whether an answer is correct.

Load `trivia_quiz.htm` and alter it as shown here:

```
<html>
<head>
<title>Wrox Online Trivia Quiz</title>

<script language="JavaScript" type="text/javascript">

function answerCorrect(questionNumber, answer)
{
```

Chapter 3: Decisions, Loops, and Functions

```
// declare a variable to hold return value
var correct = false;

// if answer provided is same as correct answer then correct variable is true
if (answer == answers[questionNumber])
    correct = true;

// return whether the answer was correct (true or false)
return correct;
}
```

```
// Questions variable will holds questions
var questions = new Array();
var answers = new Array();

// define question 1
questions[0] = new Array();

// the question
questions[0][0] = "The Beatles were";

// first choice
questions[0][1] = "A sixties rock group from Liverpool";

// second choice
questions[0][2] = "Four musically gifted insects";

// third choice
questions[0][3] = "German cars";

// fourth choice
questions[0][4] = "I don't know. Can I have the questions on baseball please?";

// assign answer for question 1
answers[0] = "A";

// define question 2
questions[1] = new Array();
questions[1][0] = "Homer Simpson's favorite food is";
questions[1][1] = "Fresh salad";
questions[1][2] = "Doughnuts";
questions[1][3] = "Bread and water";
questions[1][4] = "Apples";

// assign answer for question 2
answers[1] = "B";

// define question 3
questions[2] = new Array();
questions[2][0] = "Lisa Simpson plays which musical instrument?";
questions[2][1] = "Clarinet";
```

```
questions[2][2] = "Oboe";
questions[2][3] = "Saxophone";
questions[2][4] = "Tubular bells";

// assign answer for question 3
answers[2] = "C";
</script>
</head>

<body>

</body>
</html>
```

The only changes here are that you've removed the `alert()` function, which told the user that the array was initialized, and added a function, `answerCorrect()`, which checks whether a trivia question has been answered correctly. This function has been added inside the script block in the head of the page. The `answerCorrect()` function takes two parameters: the question index from the arrays in the parameter `questionNumber` and the answer the user has given in the parameter `answer`. It then checks whether the user's answer is in fact correct—if it is, the function returns `true`, otherwise it returns `false`.

Currently the code checks the answer given by the user by checking to see whether the element of the `answers` array with an index of `questionNumber` is equal to the `answer` parameter. Given how simple this function is, couldn't you have just included the code wherever it's needed? Why go to the bother of creating a function? The answer to this query is that we have plans for that function. Currently, its role is simply to check whether the multiple-choice response, a single letter given by the user, is the same as the letter stored in the `answers` array. However, later we'll expand the trivia quiz to handle text-based questions such as, "Which President was involved in the Watergate scandal?" and we want the answer to be considered correct whether the user enters Richard Nixon, Nixon, R Nixon, and so on. This involves more than a simple comparison, so at some point we'll expand the `answerCorrect()` function to incorporate this extra intelligence. By including it in just one function, you need to change your code in only one place and can do so without breaking other parts of your program. Code using your function expects only a `true` or `false` result—how this function comes by this result is irrelevant.

To test your new `answerCorrect()` function, let's write some code that goes through each of the questions in the `questions` array and uses the `answerCorrect()` function to work out which answer is correct. Insert the following lines into the body of the page, inside the script block after the questions and answers array have been defined. After you've finished testing, you can delete this code because it does not form part of the final trivia quiz.

```
// define question 3
questions[2] = new Array();
questions[2][0] = "Lisa Simpson plays which musical instrument";
questions[2][1] = "Clarinet";
questions[2][2] = "Oboe";
questions[2][3] = "Saxophone";
questions[2][4] = "Tubular Bells";

// assign answer for question 3
answers[2] = "C";

</script>
```

Chapter 3: Decisions, Loops, and Functions

```
</head>
<body>

<script language="javascript" type="text/javascript">
document.write("The Answer to Question 1 is " + answers[0] + "<br>");
document.write("The Answer to Question 2 is " + answers[1] + "<br>");
document.write("The Answer to Question 3 is " + answers[2]);
</script>
</body>
</html>
```

The code writes all the answers to the page by accessing the answer array and getting the letter in there that corresponds to the correct answer for each question.

If you load `trivia_quiz.htm` into your browser, you should see what is shown in Figure 3-14.

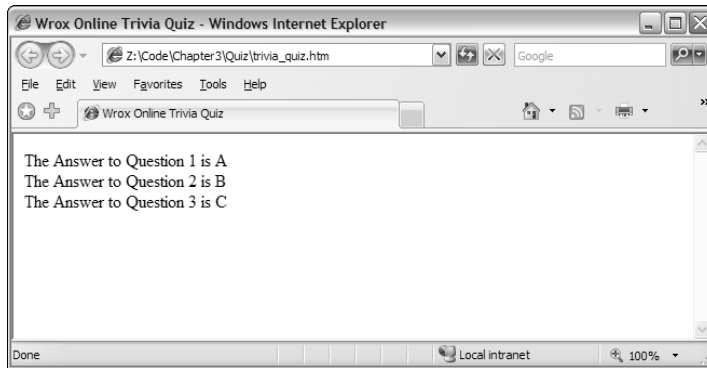


Figure 3-14

Note that the `questions` and `answers` arrays holding the question and answer data have global scope within the page, something that we warned you about in the previous section. However, here it is necessary because the arrays need to be accessed by both the function and the JavaScript code within the page.

Summary

In this chapter you have concluded your look at the core of the JavaScript language and its syntax. Everything from now on builds on these foundations, and with the less interesting syntax under your belt, you can move on to more interesting things in the remainder of the book.

The chapter looked at the following:

- ❑ **Decision making with the `if` and `switch` statements.** The ability to make decisions is essentially what gives the code its “intelligence.” Based on whether a condition is true or false, you can decide on a course of action to follow.

- ❑ **Comparison operators.** The comparison operators compare the value on the left of the operator (left-hand side, LHS) with the value on the right of the operator (right-hand side, RHS) and return a Boolean value. Here is a list of the main comparison operators:
 - ❑ `==` is the LHS equal to the RHS?
 - ❑ `!=` is the LHS not equal to the RHS?
 - ❑ `<=` is the LHS less than or equal to the RHS?
 - ❑ `>=` is the LHS greater than or equal to the RHS?
 - ❑ `<` is the LHS less than the RHS?
 - ❑ `>` is the LHS greater than the RHS?
- ❑ **The `if` statement.** Using the `if` statement, you can choose to execute a block of code (defined by being in curly braces) when a condition is `true`. The `if` statement has a test condition, specified in parentheses. If this condition evaluates to `true`, the code after the `if` statement will execute.
- ❑ **The `else` statement.** If you want code to execute when the `if` statement is `false`, you can use the `else` statement that appears after the `if` statement.
- ❑ **Logical operators.** To combine conditions, you can use the three logical operators: AND, OR, and NOT, represented by `&&`, `||`, and `!`, respectively.
 - ❑ The AND operator returns `true` only if both sides of the expression are `true`.
 - ❑ The OR operator returns `true` when either one or both sides of an expression are `true`.
 - ❑ The NOT operator reverses the logic of an expression.
- ❑ **The `switch` statement.** This compares the result of an expression with a series of possible cases, and is similar in effect to a multiple `if` statement.
- ❑ **Looping with `for`, `for...in`, `while`, and `do...while`.** It's often necessary to repeat a block of code a number of times, something JavaScript enables by looping.
 - ❑ **The `for` loop.** Useful for looping through code a certain number of times, the `for` loop consists of three parts: the initialization, test condition, and increment parts. Looping continues while the test condition is `true`. Each loop executes the block of code and then executes the increment part of the `for` loop before re-evaluating the test condition to see if the results of incrementing have changed it.
 - ❑ **The `for...in` loop.** This is useful when you want to loop through an array without knowing the number of elements in the array. JavaScript works this out for you so that no elements are missed.
 - ❑ **The `while` loop.** This is useful for looping through some code for as long as a test condition remains `true`. It consists of a test condition and the block of code that's executed only if the condition is `true`. If the condition is never `true`, then the code never executes.
 - ❑ **The `do...while` loop.** This is similar to a `while` loop, except that it executes the code once and then keeps executing the code as long as the test condition remains `true`.

- ❑ **break and continue statements.** Sometimes you have a good reason to break out of a loop prematurely, in which case you need to use the `break` statement. On hitting a `break` statement, code execution stops for the block of code marked out by the curly braces and starts immediately after the closing brace. The `continue` statement is similar to `break`, except that when code execution stops at that point in the loop, the loop is not broken out of, but instead continues as if the end of that reiteration had been reached.
- ❑ **Functions are reusable bits of code.** JavaScript has a lot of built-in functions that provide programmers services, such as converting a string to a number. However, JavaScript also enables you to define and use your own functions using the `function` keyword. Functions can have zero or more parameters passed to them and can return a value if you so wish.
- ❑ **Variable scope and lifetime.** Variables declared outside a function are available globally—that is, anywhere in the page. Any variables defined inside a function are private to that function and can't be accessed outside of it. Variables have a lifetime, the length of which depends on where the variable was declared. If it's a global variable, its lifetime is that of the page—while the page is loaded in the browser, the variable remains alive. For variables defined in a function, the lifetime is limited to the execution of that function. When the function has finished being executed, the variables die, and their values are lost. If the function is called again later in the code, the variables will be empty.

Exercise Questions

Suggested solutions to these questions can be found in Appendix A.

Question 1

A junior programmer comes to you with some code that appears not to work. Can you spot where he went wrong? Give him a hand and correct the mistakes.

```
var userAge = prompt("Please enter your age");

if (userAge = 0);
{
    alert("So you're a baby!");
}
else if ( userAge < 0 | userAge > 200)
    alert("I think you may be lying about your age");
else
{
    alert("That's a good age");
}
```

Question 2

Using `document.write()`, write code that displays the results of the 12 times table. Its output should be the results of the calculations.

```
12 * 1 = 12
12 * 2 = 24
12 * 3 = 36
.....
12 * 11 = 132
12 * 12 = 144
```

Question 3

Change the code of Question 2 so that it's a function that takes as parameters the times table required and the values at which it should start and end. For example, you might try the four times table displayed starting with $4 * 4$ and ending at $4 * 9$.

Question 4

Modify the code of Question 3 to request the times table to be displayed from the user; the code should continue to request and display times tables until the user enters -1 . Additionally, do a check to make sure that the user is entering a valid number; if the number is not valid, ask her to re-enter it.

4

JavaScript — An Object-Based Language

In this chapter, you look at a concept that is central to JavaScript, namely *objects*. But what are objects, and why are they useful?

First, we have to break it to you: You have been using objects throughout this book (for example, an array is an object). JavaScript is an object-based language, and therefore much of what you do involves manipulating objects. You'll see that when you make full use of these objects, the range of things you can do with JavaScript expands immensely.

We'll start this chapter by taking a look at the idea of what objects are and why they are important. We'll move on to what kinds of objects are used in JavaScript, how to create them and use them, and how they simplify many programming tasks for you. Finally, you'll see in more detail some of the most useful objects that JavaScript provides and how to use these in practical situations.

Not only does JavaScript itself consist of a number of these things called objects (which are also called *native JavaScript objects*), but also the browser itself is modeled as a collection of objects available for your use. You'll learn about these objects in particular in the next chapter.

Object-Based Programming

Object-based programming is a slightly scarier way of saying “programming using objects.” But what are these objects that you will be programming with? Where are they, and how and why would you want to program with them? In this section, you'll look at the answers to these questions, both in general programming terms and more specifically within JavaScript.

A Brief Introduction to Objects

To start the introduction to objects, let's think about what is meant by an object in the “real world” outside computing. The world is composed of things, or objects, such as tables, chairs, and cars (to name just a few!). Let's take a car as an example, to explore what an object really is.

Chapter 4: JavaScript — An Object-Based Language

How would you define our car? You might say it's a blue car with four-wheel drive. You might specify the speed at which it's traveling. When you do this, you are specifying *properties* of the object. For example, the car has a color property, which in this instance has the value blue.

How do you use our car? You turn the ignition key, press the gas pedal, beep the horn, change the gear (that is, choose between 1, 2, 3, 4, and reverse on a manual car, or drive and reverse on an automatic), and so on. When you do this, you are using *methods* of the object.

You can think of methods as being a bit like functions. Sometimes, you may need to use some information with the method, or pass it a parameter, to get it to work. For example, when you use the changing-gears method, you need to say which gear you want to change to. Other methods may pass information back to the owner. For example, the dipstick method will tell the owner how much oil is left in the car.

Sometimes using one or more of the methods may change one or more of the object's properties. For example, using the accelerator method will probably change the car's speed property. Other properties can't be changed: for example, the body-shape property of the car (unless you hit a brick wall with the speed property at 100 miles per hour!).

You could say that the car is defined by its collection of methods and properties. In object-based programming, the idea is to model real-world situations by objects, which are defined by their methods and properties.

Objects in JavaScript

You should now have a basic idea of what an object is — a “thing” with methods and properties. But how do you use this concept in JavaScript?

In the previous chapters you have (for the most part) been dealing with *primitive* data. These are *actual* data, such as strings and numbers. This type of data is not too complex and is fairly easy to deal with. However, not all information is as simple as primitive data. Let's look at an example to clarify things a little.

Suppose you had written a web application that displayed timetable information for buses or trains. Once the user has selected a journey, you might want to let him know how long that journey will take. To do that, you need to subtract the arrival time from the departure time.

However, that's not quite as simple as it may appear at first glance. For example, consider a departure time of 14:53 (for 2:53 p.m.) and an arrival time of 15:10 (for 3:10 p.m.). If you tell JavaScript to evaluate the expression `15.10-14.53`, you get the result 0.57, which is 57 minutes. However, you know that the real difference in time is 17 minutes. Using the normal mathematical operators on times doesn't work!

What would you need to do to calculate the difference between these two times? You would first need to separate the hours from the minutes in each time. Then, to get the difference in minutes between the two times, you would need to check whether the minutes of the arrival time were greater than the minutes of the departure. If so, you can simply subtract the departure time minutes from the arrival time minutes. If not, you need to add 60 to the arrival time minutes and subtract one from the arrival time hours to compensate, before taking the departure time minutes from the arrival time minutes. You then need to subtract the departure time hours from the arrival time hours, before putting the minutes and hours that you have arrived at back together.

This would work okay so long as the two times were in the same day. It wouldn't work, for example, with the times 23:45 and 04:32.

This way of working out the time difference obviously has its problems, but it also seems very complex. Is there an easier way to deal with more complex data such as times and dates?

This is where objects come in. You can define your departure and arrival times as `Date` objects. Because they are `Date` objects, they come with a variety of properties and methods that you can use when you need to manipulate or calculate times. For example, you can use the `getTime()` method to get the number of milliseconds between the time in the `Date` object and January 1, 1970, 00:00:00. Once you have these millisecond values for the arrival and departure times, you can simply subtract one from the other and store the result in another `Date` object. To retrieve the hours and minutes of this time, you simply use the `getHours()` and `getMinutes()` methods of the `Date` object. You'll see more examples of this later in the chapter.

The `Date` object is not the only object that JavaScript has to offer. Another object was introduced in Chapter 2, but to keep things simple, we didn't tell you it was an object at the time. That object was the `Array` object. Recall that an array is a way of holding a number of pieces of data at the same time.

`Array` objects have a property called `length` that tells you how many pieces of data, or rather how many elements, the array holds. You actually used this property in the trivia quiz in Chapter 3 to work out how many times you needed to loop through the array.

`Array` objects also have a number of methods. One example is the `sort()` method, which can be used to sort the elements within the array into alphabetical order.

You should now have an idea why objects are useful in JavaScript. You have seen the `Date` and `Array` objects, but there are many other objects that JavaScript makes available so that you can achieve more with your code. These include the `Math` and `String` objects, which we will talk more about later in the chapter.

Using JavaScript Objects

Now that you have seen the *why* of JavaScript objects, you need to look at the *what* and the *how*.

Each of the JavaScript objects has a collection of related properties and methods that can be used to manipulate a certain kind of data. For example, the `Array` object consists of methods to manipulate arrays and properties to find out information from them. In most cases, to make use of these methods and properties, you need to define your data as one of these objects. In other words, you need to create an object.

In this section, you'll look at how to go about creating an object and, having done that, how you use its properties and methods.

Creating an Object

You have already seen an example of an `Array` object being created. To create an `Array` object, you used the following JavaScript statement:

```
var myArray = new Array();
```

Chapter 4: JavaScript — An Object-Based Language

So how is this statement made up?

The first half of the statement is familiar to you. You use the `var` keyword to define a variable called `myArray`. This variable is initialized, using the equals sign assignment operator (`=`), to the right-hand side of the statement.

The right-hand side of the statement consists of two parts. First you have the operator `new`. This tells JavaScript that you want to create a new object. Next you have `Array()`. This is the *constructor* for an `Array` object. It tells JavaScript what type of object you want to create. Most objects have constructors like this. For example, the `Date` object has the `Date()` constructor. The only exception you see in this book is the `Math` object, and this will be explained in a later part of the chapter.

You also saw in Chapter 2 that you can pass parameters to the constructor `Array()` to add data to your object. For example, to create an `Array` object that has three elements containing the data "Paul", "Paula", and "Pauline", you use

```
var myArray = new Array("Paul", "Paula", "Pauline");
```

Let's see some more examples, this time using the `Date` object. The simplest way to create a `Date` object is like this:

```
var myDate = new Date();
```

This will create a `Date` object containing the date and time that it was created. However,

```
var myDate = new Date("1 Jan 2010");
```

will create a `Date` object containing the date 1 January 2010.

How object data are stored in variables differs from how primitive data, such as text and numbers, are stored. (Primitive data are the most basic data possible in JavaScript.) With primitive data, the variable holds the data's actual value. For example,

```
var myNumber = 23;
```

means that the variable `myNumber` will hold the data 23. However, variables assigned to objects don't hold the actual data, but rather a *reference* to the memory address where the data can be found. This doesn't mean you can get hold of the memory address — this is something only JavaScript has details of and keeps to itself in the background. All you need to remember is that when you say that a variable references an object, you mean it references a memory address. This is shown in the following example:

```
var myArrayRef = new Array(0, 1, 2);
var mySecondArrayRef = myArrayRef;
myArrayRef[0] = 100;
alert(mySecondArrayRef[0]);
```

First you set variable `myArrayRef` reference to the new array object, and then you set `mySecondArrayRef` to the same reference — for example, now `mySecondArrayRef` is set to reference the same array object. So when you set the first element of the array to 100, as shown here:

```
myArrayRef [0] = 100;
```


and display the contents of the first element of the array referenced in `mySecondArrayRef` as follows:

```
alert(mySecondArrayRef[0]);
```

you'll see it has also magically changed to 100! However, as you now know, it's not magic; it's because both variables referenced the same array object, because when it comes to objects, it's a reference to the object and not the object itself that is stored in a variable. When you did the assignment, it didn't make a copy of the array object, it simply copied the reference. Contrast that with the following:

```
var myVariable = "ABC";  
var mySecondVariable = myVariable;  
myVariable = "DEF";  
alert(mySecondVariable);
```

In this case you're dealing with a string, which is primitive data type, as are numbers. This time it's the actual data that are stored in the variable, so when you do this:

```
var mySecondVariable = myVariable;
```

`mySecondVariable` gets its own separate copy of the data in `myVariable`. So the alert at the end will still show `mySecondVariable` as holding "ABC".

To summarize this section, you create a JavaScript object using the following basic syntax:

```
var myVariable = new ObjectName(optional parameters);
```

Using an Object's Properties

Accessing the values contained in an object's properties is very simple. You write the name of the variable containing (or referencing) your object, followed by a dot, and then the name of the object's property.

For example, if you defined an `Array` object contained in the variable `myArray`, you could access its `length` property like this:

```
myArray.length
```

But what can you do with this property now that you have it? You can use it as you would any other piece of data and store it in a variable:

```
var myVariable = myArray.length;
```

Or you can show it to the user:

```
alert(myArray.length);
```

In some cases, you can even change the value of the property, like this:

```
myArray.length = 12;
```

However, unlike variables, some properties are read-only — you can get information from them, but you can't *change* information inside them.

Calling an Object's Methods

Methods are very much like functions in that they can be used to perform useful tasks, such as getting the hours from a particular date or generating a random number. Again like functions, some methods return a value, such as the `Date` object's `getHours()` method, while others perform a task, but return no data, such as the `Array` object's `sort()` method.

Using the methods of an object is very similar to using properties, in that you put the object's variable name first, then a dot, and then the name of the method. For example, to sort the elements of an `Array` in the variable `myArray`, you may use the following code:

```
myArray.sort();
```

Just as with functions, you can pass parameters to some methods by placing the parameters between the parentheses following the method's name. However, whether or not a method takes parameters, you must still put parentheses after the method's name, just as you did with functions. As a general rule, anywhere you can use a function, you can use a method of an object.

Primitives and Objects

You should now have a good idea about the difference between primitive data, such as numbers and strings, and object data, such as `Dates` and `Arrays`. However, we mentioned earlier that there is also a `String` object. Where does this fit in?

In fact there are `String`, `Number`, and `Boolean` objects corresponding to the string, number, and `Boolean` primitive data types. For example, to create a `String` object containing the text "I'm a String object" you can use the following code:

```
var myString = new String("I'm a String object");
```

The `String` object has the `length` property just as the `Array` object does. This returns the number of characters in the `String` object. For example,

```
var lengthOfString = myString.length;
```

would store the data 19 in the variable `lengthOfString` (remember that spaces are referred to as characters too).

But what if you had declared a primitive string called `mySecondString` holding the text "I'm a primitive string" like this:

```
var mySecondString = "I'm a primitive string";
```

and wanted to know how many characters could be found in this primitive string?

This is where JavaScript helps you out. Recall from previous chapters that JavaScript can handle the conversion of one data type to another automatically. For example, if you tried to add a string primitive to a number primitive, like this,

```
theResult = "23" + 23;
```

JavaScript would assume that you want to treat the number as a string and concatenate the two together, the number being converted to text automatically. The variable `theResult` would contain `"2323"` — the concatenation of 23 and 23, and not the sum of 23 and 23, which would be 46.

The same applies to objects. If you declare a primitive string and then treat it as an object, such as by trying to access one of its methods or properties, JavaScript will know that the operation you're trying to do won't work. The operation will only work with an object; for example, it would be valid with a `String` object. In this case, JavaScript converts the plain-text string into a temporary `String` object, just for that operation.

So, for your primitive string `mySecondString`, you can use the `length` property of the `String` object to find out the number of characters it contains. For example:

```
var lengthOfSecondString = mySecondString.length;
```

This would store the data 22 in the variable `lengthOfSecondString`.

The same ideas expressed here are also true for number and Boolean primitives and their corresponding `Number` and `Boolean` objects. However, these objects are not used very often, so we will not be discussing them further in this book.

The JavaScript Native Objects

So far you have just been looking at what objects are, how to create them, and how to use them. Now take a look at some of the more useful objects that are native to JavaScript — that is, those that JavaScript makes available for you to use.

You won't be looking at all of the native JavaScript objects, just some of the more commonly used ones, namely the `String` object, the `Math` object, the `Array` object, and the `Date` object. Later in the book, a whole chapter is devoted to each of the more complex objects, such as the `String` object (Chapter 8) and the `Date` object (Chapter 9).

String Objects

Like most objects, `String` objects need to be created before they can be used. To create a `String` object, you can write this:

```
var string1 = new String("Hello");  
var string2 = new String(123);  
var string3 = new String(123.456);
```

However, as you have seen, you can also declare a string primitive and use it as if it were a `String` object, letting JavaScript do the conversion to an object for you behind the scenes. For example:

```
var string1 = "Hello";
```

Using this technique is preferable so long as it's clear to JavaScript what object you expect to have created in the background. If the primitive data type is a string, this won't be a problem and JavaScript will work it out. The advantages to doing it this way are that there is no need to create a `String` object itself

Chapter 4: JavaScript — An Object-Based Language

and you avoid the troubles with comparing string objects. When you try to compare string objects with primitive string values, the actual values are compared; but with `String` objects, it's the object references that are compared.

The `String` object has a vast number of methods and properties. In this section, you'll be looking only at some of the less complex and more commonly used methods. However, in Chapter 8 you'll look at some of the trickier, but very powerful, methods associated with strings and the regular expression object (`RegExp`). Regular expressions provide a very powerful means of searching strings for patterns of characters. For example, if you want to find "Paul" where it exists as a whole word in the string "Pauline, Paul, Paula", you need to use regular expressions. However, they can be a little tricky to use, so we won't discuss them further in this chapter — we want to save some fun for later!

With most of the `String` object's methods, it helps to remember that a string is just a series of individual characters and that, as with arrays, each character has a position, or index. Also as with arrays, the first position, or index, is labeled 0 and not 1. So, for example, the string "Hello World" has the character positions shown in the following table:

Character Index	0	1	2	3	4	5	6	7	8	9	10
Character	H	e	l	l	o		W	o	r	l	d

The length Property

The `length` property simply returns the number of characters in the string. For example,

```
var myName = new String("Paul");
document.write(myName.length);
```

will write the length of the string "Paul" (that is, 4) to the page.

The `charAt()` and `charCodeAt()` Methods — Selecting a Single Character from a String

If you want to find out information about a single character within a string, you need the `charAt()` and `charCodeAt()` methods. These methods can be very useful for checking the validity of user input, something you'll see more of in Chapter 6 when you look at HTML forms.

The `charAt()` method takes one parameter: the index position of the character you want in the string. It then returns that character. `charAt()` treats the positions of the string characters as starting at 0, so the first character is at index 0, the second at index 1, and so on.

For example, to find the last character in a string, you could use this code:

```
var myString = prompt("Enter some text", "Hello World!");
var theLastChar = myString.charAt(myString.length - 1);
document.write("The last character is " + theLastChar);
```

In the first line you prompt the user for a string, with the default of "Hello World!", and store this string in the variable `myString`.

In the next line, you use the `charAt()` method to retrieve the last character in the string. You use the index position of `(myString.length - 1)`. Why? Let's take the string "Hello World!" as an example. The `length` of this string is 12, but the last character position is 11 because the indexing starts at 0. Therefore, you need to subtract one from the length of the string to get the last character's position.

In the final line, you write the last character in the string to the page.

The `charCodeAt()` method is similar in use to the `charAt()` method, but instead of returning the character itself, it returns a number that represents the decimal character code for that character in the Unicode character set. Recall that computers only understand numbers—to the computer, all your strings are just numeric data. When you request text rather than numbers, the computer does a conversion based on its internal understanding of each number and provides the respective character.

For example, to find the character code of the first character in a string, you could write this:

```
var myString = prompt("Enter some text", "Hello World!");
var theFirstCharCode = myString.charCodeAt(0);
document.write("The first character code is " + theFirstCharCode);
```

This will get the character code for the character at index position 0 in the string given by the user, and write it out to the page.

Character codes go in order, so, for example, the letter A has the code 65, B 66, and so on. Lowercase letters start at 97 (a is 97, b is 98, and so on). Digits go from 48 (for the number 0) to 57 (for the number 9). You can use this information for various purposes, as you'll see in the next example.

Try It Out Checking a Character's Case

The following is an example that detects the type of the character at the start of a given string—that is, whether the character is uppercase, lowercase, numeric, or other.

```
<html>
<head>
<script language="JavaScript" type="text/javascript">
function checkCharType(charToCheck)
{
    var returnValue = "O";
    var charCode = charToCheck.charCodeAt(0);

    if (charCode >= "A".charCodeAt(0) && charCode <= "Z".charCodeAt(0))
    {
        returnValue = "U";
    }
    else if (charCode >= "a".charCodeAt(0) && charCode <= "z".charCodeAt(0))
    {
        returnValue = "L";
    }
    else if (charCode >= "0".charCodeAt(0) && charCode <= "9".charCodeAt(0))
    {
        returnValue = "N";
    }
}
```

```
        return returnValue;
    }
</script>
<head>

<body>
<script language="JavaScript" type="text/javascript">

var myString = prompt("Enter some text","Hello World!");
switch (checkCharType(myString))
{
    case "U":
        document.write("First character was upper case");
        break;
    case "L":
        document.write("First character was lower case");
        break;
    case "N":
        document.write("First character was a number");
        break;
    default:
        document.write("First character was not a character or a number");
}
</script>
</body>
</html>
```

Type in the code and save it as `ch4_examp1.htm`.

When you load the page into your browser, you will be prompted for a string. A message will then be written to the page informing you of the type of the first character that you entered — whether it is uppercase, lowercase, a number, or something else, such as a punctuation mark.

How It Works

To start with, you define a function `checkCharType()`, which is used in the body of the page. You start this function by declaring the variable `returnValue` and initializing it to the character "O" to indicate it's some other character than a lowercase letter, uppercase letter, or numerical character.

```
function checkCharType(charToCheck)
{
    var returnValue = "O";
```

You use this variable as the value to be returned at the end of the function, indicating the type of character. It will take the values `U` for uppercase, `L` for lowercase, `N` for number, and `O` for other.

The next line in the function uses the `charCodeAt()` method to get the character code of the first character in the string stored in `charToCheck`, which is the function's only parameter. The character code is stored in the variable `charCode`.

```
var charCode = charToCheck.charCodeAt(0);
```

In the following lines you have a series of `if` statements, which check within what range of values the character code falls. You know that if it falls between the character codes for A and Z, it's uppercase, and so you assign the variable `returnValue` the value U. If the character code falls between the character codes for a and z, it's lowercase, and so you assign the value L to the variable `returnValue`. If the character code falls between the character codes for 0 and 9, it's a number, and you assign the value N to the variable `returnValue`. If the value falls into none of these ranges, then the variable retains its initialization value of O for other, and you don't have to do anything.

```
if (charCode >= "A".charCodeAt(0) && charCode <= "Z".charCodeAt(0))
{
    returnValue = "U";
}
else if (charCode >= "a".charCodeAt(0) && charCode <= "z".charCodeAt(0))
{
    returnValue = "L";
}
else if (charCode >= "0".charCodeAt(0) && charCode <= "9".charCodeAt(0))
{
    returnValue = "N";
}
```

This probably seems a bit weird at first, so let's see what JavaScript is doing with your code. When you write

```
"A".charCodeAt(0)
```

it appears that you are trying to use a method of the `String` object on a string literal, which is the same as a primitive string in that it's just characters and not an object. However, JavaScript realizes what you are doing and does the necessary conversion of literal character "A" into a temporary `String` object containing "A". Then, and only then, does JavaScript perform the `charCodeAt()` method on the `String` object it has created in the background. When it has finished, the `String` object is disposed of. Basically, this is a shorthand way of writing the following:

```
var myChar = new String("A");
myChar.charCodeAt(0);
```

In either case the first (and in this string the only) character's code is returned to you. For example, `"A".charCodeAt(0)` will return the number 65.

Finally you come to the end of the function and return the `returnValue` variable to where the function was called.

```
return returnValue;
}
```

You might wonder why you bother using the variable `returnValue` at all, instead of just returning its value. For example, you could write the code as follows:

```
if (charCode >= "A".charCodeAt(0) && charCode <= "Z".charCodeAt(0))
{
    return "U";
}
```

Chapter 4: JavaScript — An Object-Based Language

```
    }  
    else if (charCode >= "a".charCodeAt(0) && charCode <= "z".charCodeAt(0))  
    {  
        return "L";  
    }  
    else if (charCode >= "0".charCodeAt(0) && charCode <= "9".charCodeAt(0))  
    {  
        return "N";  
    }  
    return "O";
```

This would work fine, so why not do it this way? The disadvantage of this way is that it's difficult to follow the flow of execution of the function, which is not that bad in a small function like this, but can get tricky in bigger functions. With the original code you always know exactly where the function execution stops: It stops at the end with the only `return` statement. The version of the function just shown finishes when any of the `return` statements is reached, so there are four possible places where the function might end.

In the body of your page, you have some test code to check that the function works. You first use the variable `myString`, initialized to "Hello World!" or whatever the user enters into the prompt box, as your test string.

```
var myString = prompt("Enter some text","Hello World!");
```

Next, the `switch` statement uses the `checkCharType()` function that you defined earlier in its comparison expression. Depending on what is returned by the function, one of the `case` statements will execute and let the user know what the character type was.

```
switch (checkCharType(myString))  
{  
    case "U":  
        document.write("First character was upper case");  
        break;  
    case "L":  
        document.write("First character was lower case");  
        break;  
    case "N":  
        document.write("First character was a number");  
        break;  
    default:  
        document.write("First character was not a character or a number");  
}
```

That completes the example, but before we move on, it's worth noting that this example is just that — an example of using `charCodeAt()`. In practice, it would be much easier to just write

```
if (char >= "A" && char <= "Z")
```

rather than

```
if (charCode >= "A".charCodeAt(0) && charCode <= "Z".charCodeAt(0))
```

which you have used here.

The fromCharCode() Method — Converting Character Codes to a String

The method `fromCharCode()` can be thought of as the opposite of `charCodeAt()`, in that you pass it a series of comma-separated numbers representing character codes, and it converts them to a single string.

However, the `fromCharCode()` method is unusual in that it's a *static* method — you don't need to have created a `String` object to use it with, it's always available to you.

For example, the following lines put the string "ABC" into the variable `myString`:

```
var myString;  
myString = String.fromCharCode(65,66,67);
```

The `fromCharCode()` method can be very useful when used with variables. For example, to build up a string consisting of all the uppercase letters of the alphabet, you could use the following code:

```
var myString = "";  
var charCode;  
  
for (charCode = 65; charCode <= 90; charCode++)  
{  
    myString = myString + String.fromCharCode(charCode);  
}  
  
document.write(myString);
```

You use the `for` loop to select each character from A to Z in turn and concatenate this to `myString`. Note that while this is fine as an example, it is more efficient and less memory-hungry to simply write this instead:

```
var myString = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

The indexOf() and lastIndexOf() Methods — Finding a String Inside Another String

The methods `indexOf()` and `lastIndexOf()` are used for searching for the occurrence of one string inside another. A string contained inside another is usually termed a *substring*. They are useful when you have a string of information, but only want a small part of it. For example, in the trivia quiz, when someone enters a text answer, you want to check if certain keywords are present within the string.

Both `indexOf()` and `lastIndexOf()` take two parameters:

- ❑ The string you want to find
- ❑ The character position you want to start searching from (optional)

As with the `charAt()` method, character positions start at 0. If you don't include the second parameter, searching starts from the beginning of the string.

Chapter 4: JavaScript — An Object-Based Language

The return value of `indexOf()` and `lastIndexOf()` is the character position in the string at which the substring was found. Again, it's zero-based, so if the substring is found at the start of the string, then 0 is returned. If there is no match, then the value -1 is returned.

For example, to search for the substring "Paul" in the string "Hello paul. How are you Paul", you may use the code

```
<script language="JavaScript" type="text/javascript">

var myString = "Hello paul. How are you Paul";
var foundAtPosition;

foundAtPosition = myString.indexOf("Paul");
alert(foundAtPosition);

</script>
```

This code should result in a message box containing the number 24, which is the character position of "Paul". You might be wondering why it's 24, which clearly refers to the second "Paul" in the string, rather than 6 for the first "paul". Well, this is due to case sensitivity again. It's laboring the point a bit, but JavaScript takes case sensitivity very seriously, both in its syntax and when making comparisons. If you type `IndexOf()` instead of `indexOf()`, JavaScript will complain. Similarly, "paul" is not the same as "Paul". Mistakes with case are so easy to make, even for experts, that it's best to be very aware of case when programming.

You've seen `indexOf()` in action, but how does `lastIndexOf()` differ? Well, whereas `indexOf()` starts searching from the beginning of the string, or the position you specified in the second parameter, and works towards the end, `lastIndexOf()` starts at the end of the string, or the position you specified, and works towards the beginning of the string.

In the current example you first search using `indexOf()`, which finds the first "Paul" (changed to the correct case from the last example). The alert box displays this result, which is character position 6. Then you search using `lastIndexOf()`. This starts searching at the end of the string, and so the first "Paul" it comes to is the last one in the string at character position 24. Therefore, the second alert box displays the result 24.

```
<script language="JavaScript" type="text/javascript">

var myString = "Hello Paul. How are you Paul";
var foundAtPosition;

foundAtPosition = myString.indexOf("Paul");
alert(foundAtPosition);

foundAtPosition = myString.lastIndexOf("Paul");
alert(foundAtPosition);

</script>
```

Try it Out Counting Occurrences of Substrings

In this example, you look at how to use the “start character position” parameter of `indexOf()`. Here you will count how many times the word `Wrox` appears in the string.

```
<html>
<body>
<script language="JavaScript" type="text/javascript">
var myString = "Welcome to Wrox books. ";
myString = myString + "The Wrox website is www.wrox.com. ";
myString = myString + "Visit the Wrox website today. Thanks for buying Wrox";

var foundAtPosition = 0;
var wroxCount = 0;

while ( foundAtPosition != -1)
{
    foundAtPosition = myString.indexOf("Wrox",foundAtPosition);
    if (foundAtPosition != -1)
    {
        wroxCount++;
        foundAtPosition++;
    }
}

document.write("There are " + wroxCount + " occurrences of the word Wrox");

</script>
</body>
</html>
```

Save this example as `ch4_examp2.htm`. When you load the page into your browser, you should see the following sentence: There are 4 occurrences of the word `Wrox`.

How It Works

At the top of the script block, you built up a string inside the variable `myString`, which you then want to search for the occurrence of the word `Wrox`. You also define two variables: `wroxCount` will contain the number of times `Wrox` is found in the string, and `foundAtPosition` will contain the position in the string of the current occurrence of the substring `Wrox`.

You then used a `while` loop, which continues looping all the while you are finding the word `Wrox` in the string—that is, while the variable `foundAtPosition` is not equal to `-1`. Inside the `while` loop, you have this line:

```
foundAtPosition = myString.indexOf("Wrox",foundAtPosition);
```

Here you search for the next occurrence of the substring `Wrox` in the string `myString`. How do you make sure that you get the next occurrence? You use the variable `foundAtPosition` to give you the starting position of your search, because this contains the index after the index position of the last occurrence of the substring `Wrox`. You assign the variable `foundAtPosition` to the result of your search, the index position of the next occurrence of the substring `Wrox`.

Chapter 4: JavaScript — An Object-Based Language

Each time `wrox` is found (that is, each time `foundAtPosition` is not `-1`) you increase the variable `wroxCount`, which counts how many times you have found the substring, and you increase `foundAtPosition` so that you continue the search at the next position in the string.

```
if (foundAtPosition != -1)
{
    wroxCount++;
    foundAtPosition++;
}
```

Finally, you `document.write()` the value of the variable `wroxCount` to the page.

In the Chapter 3, we talked about the danger of infinite loops, and you can see that there is a danger of one here. If `foundAtPosition++` were removed, you'd keep searching from the same starting point and never move to find the next occurrence of the word `Wrox`.

The `indexOf()` and `lastIndexOf()` methods are more useful when coupled with the `substr()` and `substring()` methods, which you'll be looking at in the next section. Using a combination of these methods enables you to cut substrings out of a string.

The `substr()` and `substring()` Methods — Copying Part of a String

If you wanted to cut out part of a string and assign that cut-out part to another variable or use it in an expression, you would use the `substr()` and `substring()` methods. Both methods provide the same end result — that is, a part of a string — but they differ in the parameters they require.

The method `substring()` takes two parameters: the character start position and the character end position of the part of the string you want. The second parameter is optional; if you don't include it, all characters from the start position to the end of the string are included.

For example, if your string is `"JavaScript"` and you want just the text `"Java"`, you could call the method like so:

```
var myString = "JavaScript";
var mySubString = myString.substring(0,4);
alert(mySubString);
```

As with all the methods of the `String` object so far, the character positions start at 0. However, you might be wondering why you specified the end character as 4. This method is a little confusing because the end character is the end marker; it's not included in the substring that is cut out. You can think of the parameters as specifying the *length* of the string being returned: the parameters 0 and 4 will return (4 - 0) characters starting at and including the character at position 0. Depicted graphically it looks like this:

Character Position	0	1	2	3	4	5	6	7	8	9
Character	J	a	v	a	S	c	r	i	p	t

Like `substring()`, the method `substr()` again takes two parameters, the first being the start position of the first character you want included in your substring. However, this time the second parameter specifies the length of the string of characters that you want to cut out of the longer string. For example, you could rewrite the preceding code like this:

```
var myString = "JavaScript";
var mySubString = myString.substr(0,4);
alert(mySubString);
```

As with the `substring()` method, the second parameter is optional. If you don't include it, all the characters from the start position onward will be included.

The main reason for using one method rather than the other is that the `substring()` method is supported by IE 3+ and by NN 2+ browsers. However, the `substr()` method only works with version 4 (and later) browsers.

Let's look at the use of the `substr()` and `lastIndexOf()` methods together. In the next chapter, you'll see how you can retrieve the file path and name of the currently loaded web page. However, there is no way of retrieving the file name alone. So if, for example, your file is

`http://mywebsite/temp/myfile.htm`, you may need to extract the `myfile.htm` part. This is where `substr()` and `lastIndexOf()` are useful.

```
var fileName = window.location.href;
fileName = fileName.substr(fileName.lastIndexOf("/") + 1);
document.write("The file name of this page is " + fileName);
```

The first line sets the variable `fileName` to the current file path and name, such as `/mywebsite/temp/myfile.htm`. Don't worry about understanding this line; you'll be looking at it in the next chapter.

The second line is where the interesting action is. You can see that this code uses the return value of the `lastIndexOf()` method as a parameter for another method, something that's perfectly correct and very useful. The goal in using `fileName.lastIndexOf("/")` is to find the position of the final forward slash (`/`), which will be the last character before the name of the file. You add one to this value, because you don't want to include that character, and then pass this new value to the `substr()` method. There's no second parameter here (the length), because you don't know it. As a result, `substr()` will return all the characters right to the end of the string, which is what you want.

This example retrieves the name of the page on the local machine, because you're not accessing the page from a web server. However, don't let this mislead you into thinking that accessing files on a local hard drive from a web page is something you'll be able to do with JavaScript alone. To protect users from malicious hackers, JavaScript's access to the user's system, such as access to files, is very limited. You'll learn more about this later in the book.

The `toLowerCase()` and `toUpperCase()` Methods — Changing the Case of a String

If you want to change the case of a string, for example to remove case sensitivity when comparing strings, you need the `toLowerCase()` and `toUpperCase()` methods. It's not hard to guess what these two methods do. Both of them return a string that is the value of the string in the `String` object, but with its case converted to either upper or lower depending on the method invoked. Any non-alphabetical characters remain unchanged by these functions.

Chapter 4: JavaScript — An Object-Based Language

In the following example, you can see that by changing the case of both strings you can compare them without case sensitivity being an issue.

```
var myString = "I Don't Care About Case"

if (myString.toLowerCase() == "i don't care about case")
{
    alert("Who cares about case?");
}
```

Even though `toLowerCase()` and `toUpperCase()` don't take any parameters, you must remember to put the two empty parentheses—that is, `()`—at the end, if you want to call a method.

The Math Object

The `Math` object provides a number of useful mathematical functions and number manipulation methods. You'll be taking a look at some of them here, but you'll find the rest described in detail at the W3C site: www.w3schools.com/jsref/default.asp.

The `Math` object is a little unusual in that JavaScript automatically creates it for you. There's no need to declare a variable as a `Math` object or define a new `Math` object before being able to use it, making it a little bit easier to use.

The properties of the `Math` object include some useful math constants, such as the `PI` property (giving the value 3.14159 and so on). You access these properties, as usual, by placing a dot after the object name (`Math`) and then writing the property name. For example, to calculate the area of a circle, you may use the following code:

```
var radius = prompt("Give the radius of the circle", "");
var area = (Math.PI)*radius*radius;
document.write("The area is " + area);
```

The methods of the `Math` object include some operations that are impossible, or complex, to perform using the standard mathematical operators (+, -, *, and /). For example, the `cos()` method returns the cosine of the value passed as a parameter. You'll look at a few of these methods now.

The abs() Method

The `abs()` method returns the absolute value of the number passed as its parameter. Essentially, this means that it returns the positive value of the number. So -1 is returned as 1, -4 as 4, and so on. However, 1 would be returned as 1 because it's already positive.

For example, the following code would write the number 101 to the page.

```
var myNumber = -101;
document.write(Math.abs(myNumber));
```

The `ceil()` Method

The `ceil()` method always rounds a number up to the next largest whole number or integer. So `10.01` becomes `11`, and `-9.99` becomes `-9` (because `-9` is greater than `-10`). The `ceil()` method has just one parameter, namely the number you want rounded up.

Using `ceil()` is different from using the `parseInt()` function you saw in Chapter 2, because `parseInt()` simply chops off any numbers after the decimal point to leave a whole number, whereas `ceil()` rounds the number up.

For example, the following code writes two lines in the page, the first containing the number `102` and the second containing the number `101`:

```
var myNumber = 101.01;
document.write(Math.ceil(myNumber) + "<BR>");
document.write(parseInt(myNumber));
```

The `floor()` Method

Like the `ceil()` method, the `floor()` method removes any numbers after the decimal point, and returns a whole number or integer. The difference is that `floor()` always rounds the number down. So if you pass `10.01` you will be returned `10`, and if you pass `-9.99` you will see `-10` returned.

The `round()` Method

The `round()` method is very similar to `ceil()` and `floor()`, except that instead of always rounding up or always rounding down, it rounds up only if the decimal part is `.5` or greater, and rounds down otherwise.

For example:

```
var myNumber = 44.5;
document.write(Math.round(myNumber) + "<BR>");

myNumber = 44.49;
document.write(Math.round(myNumber));
```

This code would write the numbers `45` and `44` to the page.

Summary of Rounding Methods

As you have seen, the `ceil()`, `floor()`, and `round()` methods all remove the numbers after a decimal point and return just a whole number. However, which whole number they return depends on the method used: `floor()` returns the lowest, `ceil()` the highest, and `round()` the nearest equivalent integer. This can be a little confusing, so the following is a table of values and what whole number would be returned if these values were passed to the `parseInt()` function, and `ceil()`, `floor()`, and `round()` methods.

Parameter	parseInt() returns	ceil() returns	floor() returns	round() returns
10.25	10	11	10	10
10.75	10	11	10	11
10.5	10	11	10	11
-10.25	-10	-10	-11	-10
-10.75	-10	-10	-11	-11
-10.5	-10	-10	-11	-10

Remember that parseInt() is a native JavaScript function and not a method of the Math object, like the other methods presented in this table.

Try It Out JavaScript’s Rounding Functions Results Calculator

If you’re still not sure about rounding numbers, the following example should help. Here, you’ll look at a calculator that gets a number from the user, then writes out what the result would be when you pass that number to parseInt(), ceil(), floor(), and round().

```
<html>
<body>
<script language="JavaScript" type="text/javascript">

var myNumber = prompt("Enter the number to be rounded","");

document.write("<h3>The number you entered was " + myNumber + "</h3><br>");
document.write("<p>The rounding results for this number are</p>");
document.write("<table width=150 border=1>");
document.write("<tr><th>Method</th><th>Result</th></tr>");
document.write("<tr><td>parseInt()</td><td>" + parseInt(myNumber) + "</td></tr>");
document.write("<tr><td>ceil()</td><td>" + Math.ceil(myNumber) + "</td></tr>");
document.write("<tr><td>floor()</td><td>" + Math.floor(myNumber) + "</td></tr>");
document.write("<tr><td>round()</td><td>" + Math.round(myNumber) + "</td></tr>");
document.write("</table>")

</script>
</body>
</html>
```

Save this as ch4_examp3.htm and load it into a web browser. In the prompt box, enter a number, for example 12.354, and click OK. The results of this number being passed to parseInt(), ceil(), floor(), and round() will be displayed in the page formatted inside a table, as shown in Figure 4-1.

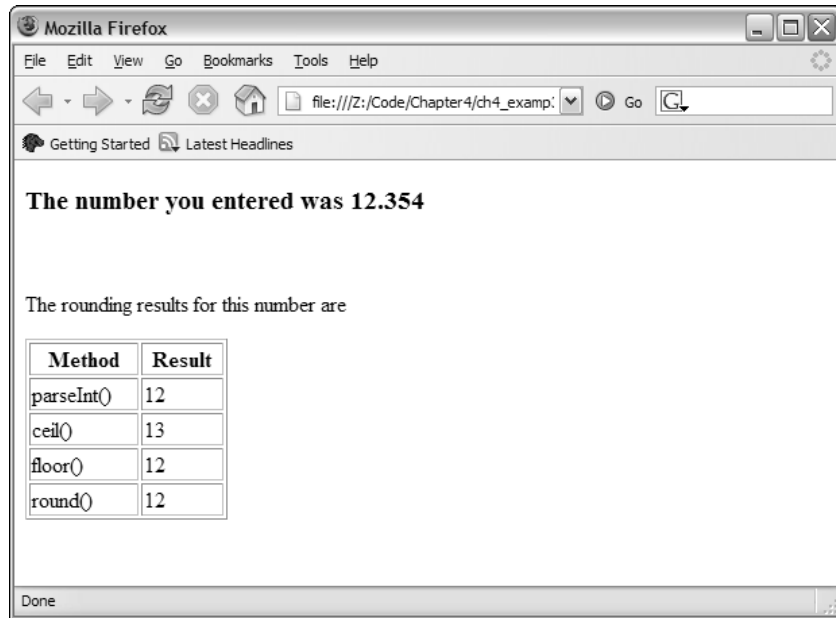


Figure 4-1

How It Works

The first task is to get the number to be rounded from the user:

```
var myNumber = prompt("Enter the number to be rounded", "");
```

Then you write out the number and some descriptive text.

```
document.write("<h3>The number you entered was " + myNumber + "</h3><br>");
document.write("<p>The rounding results for this number are</p>");
```

Notice how this time some HTML tags for formatting have been included — the main header being in `<h3>` tags, and the description of what the table means being inside a paragraph `<p>` tag.

Next you create the table of results.

```
document.write("<table width=150 border=1>");
document.write("<tr><th>Method</th><th>Result</th></tr>");
document.write("<tr><td>parseInt()</td><td>" + parseInt(myNumber) + "</td></tr>");
document.write("<tr><td>ceil()</td><td>" + Math.ceil(myNumber) + "</td></tr>");
document.write("<tr><td>floor()</td><td>" + Math.floor(myNumber) + "</td></tr>");
document.write("<tr><td>round()</td><td>" + Math.round(myNumber) + "</td></tr>");
document.write("</table>");
```

Chapter 4: JavaScript — An Object-Based Language

You create the table header first before actually displaying the results of each rounding function on a separate row. You can see how easy it is to dynamically create HTML inside the web page using just JavaScript. The principles are the same as with HTML in a page: You must make sure your tag's syntax is valid or otherwise things will appear strange or not appear at all.

Each row follows the same principle, but uses a different rounding function. Let's look at the first row, which displays the results of `parseInt()`.

```
document.write("<tr><td>parseInt()</td><td>" + parseInt(myNumber) + "</td></tr>");
```

Inside the string to be written out to the page, you start by creating the table row with the `<tr>` tag. Then you create a table cell with a `<td>` tag and insert the name of the method from which the results are being displayed on this row. Then you close the cell with `</td>` and open a new one with `<td>`. Inside this next cell you are placing the actual results of the `parseInt()` function. Although a number is returned by `parseInt()`, because you are concatenating it to a string, JavaScript automatically converts the number returned by `parseInt()` into a string before concatenating. All this happens in the background without you needing to do a thing. Finally, you close the cell and the row with `</td></tr>`.

The `random()` Method

The `random()` method returns a random floating-point number in the range between 0 and 1, where 0 is included and 1 is not. This can be very useful for displaying random banner images or for writing a JavaScript game.

Let's look at how you would mimic the roll of a single die. In the following page, 10 random numbers are written to the page. Click the browser's Refresh button to get another set of random numbers.

```
<html>
<body>
<script language="JavaScript" type="text/javascript">
var throwCount;
var diceThrow;
for (throwCount = 0; throwCount < 10; throwCount++)
{
    diceThrow = (Math.floor(Math.random() * 6) + 1);
    document.write(diceThrow + "<br>");
}

</script>
</body>
</html>
```

You want `diceThrow` to be between 1 and 6. The `random()` function returns a floating-point number between 0 and just under 1. By multiplying this number by 6, you get a number between 0 and just under 6. Then by adding 1, you get a number between 1 and just under 7. By using `floor()` to always round it down to the next lowest whole number, you can ensure that you'll end up with a number between 1 and 6.

If you wanted a random number between 1 and 100, you would just change the code so that `Math.random()` is multiplied by 100 rather than 6.

The `pow()` Method

The `pow()` method raises a number to a specified power. It takes two parameters, the first being the number you want raised to a power, and the second being the power itself. For example, to raise 2 to the power of 8 (that is, to calculate $2 * 2 * 2 * 2 * 2 * 2 * 2 * 2$), you would write `Math.pow(2, 8)` — the result being 256. Unlike some of the other mathematical methods, like `sin()`, `cos()`, and `acos()`, which are not commonly used in web programming unless it's a scientific application you're writing, the `pow()` method can often prove very useful.

Try It Out Using `pow()`

In the following example, you write a function using `pow()`, which fixes the number of decimal places in a number — a function that's missing from earlier versions of JavaScript, though it has now been added to JScript 5.5 and JavaScript 1.5, as you'll see later in this chapter. This helps demonstrate that even when a function is missing from JavaScript, you can usually use existing functions to create what you want.

```
<html>
<head>
<script language="JavaScript" type="text/javascript">

function fix(fixNumber, decimalPlaces)
{
    var div = Math.pow(10,decimalPlaces);
    fixNumber = Math.round(fixNumber * div) / div;
    return fixNumber;
}
</script>
</head>
<body>
<script language="JavaScript" type="text/javascript">

var number1 = prompt("Enter the number with decimal places you want to fix","");
var number2 = prompt("How many decimal places do you want?","");

document.write(number1 + " fixed to " + number2 + " decimal places is: ");
document.write(fix(number1,number2));

</script>
</body>
</html>
```

Save the page as `ch4_examp4.htm`. When you load the page into your browser, you will be presented with two prompt boxes. In the first, enter the number for which you want to fix the number of decimal places, for example 2.2345. In the second, enter the number of decimal places you want fixed, for example 2. Then the result of fixing the number you have entered to the number of decimal places you have chosen will be written to the page, as shown in Figure 4-2. For the example numbers, this will be 2.23.

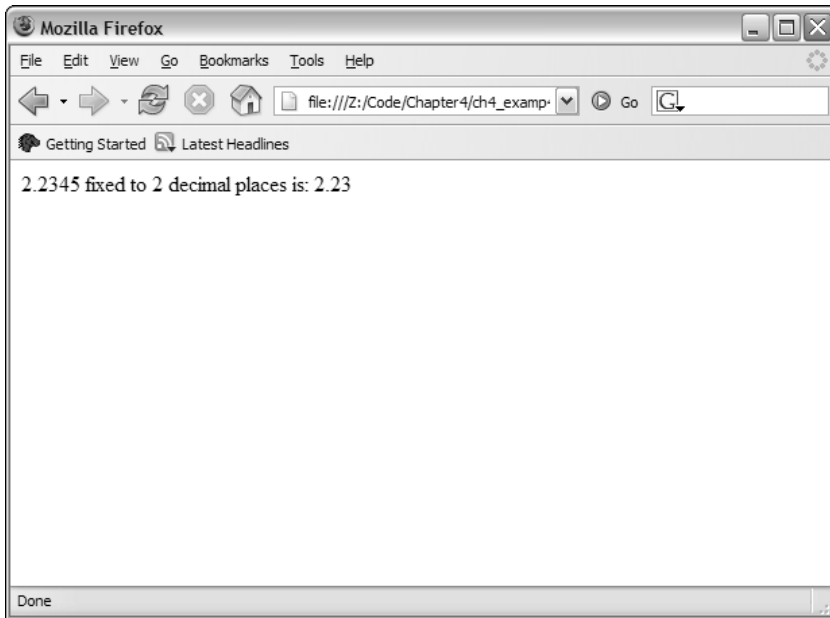


Figure 4-2

How It Works

In the head of the page you define the function `fix()`. This function will fix its `fixNumber` parameter to a maximum of its `decimalPlaces` parameter's number of digits after the decimal place. For example, fixing 34.76459 to a maximum of three decimal places will return 34.765.

The first line of code in the function sets the variable `div` to the number 10 raised to the power of the number of decimal places you want.

```
function fix(fixNumber, decimalPlaces)
{
    var div = Math.pow(10, decimalPlaces);
```

Then, in the next line, you calculate the new number.

```
fixNumber = Math.round(fixNumber * div) / div;
```

What the code `Math.round(fixNumber * div)` does is move the decimal point in the number that you are converting to after the point in the number that you want to keep. So for 2.2345, if you want to keep two decimal places, you convert it to 223.45. The `Math.round()` method rounds this number to the nearest integer (in this case 223) and so removes any undesired decimal part.

You then convert this number back into the fraction it should be, but of course only the fractional part you want is left. You do this by dividing by the same number (`div`) that you multiplied by. In this example, you divide 223 by 100, which leaves 2.23. This is 2.2345 fixed to two decimal places. This value is returned to the calling code in the line

```
    return fixNumber;
}
```

In the body of the page you use two prompt boxes to get numbers from the user. You then display the results of using these numbers in your `fix()` function to the user using `document.write()`.

Number Object

As with the `String` object, `Number` objects need to be created before they can be used. To create a `Number` object, you can write the following:

```
var firstNumber = new Number(123);
var secondNumber = new Number('123');
```

However, as you have seen, you can also declare a number as primitive and use it as if it were a `Number` object, letting JavaScript do the conversion to an object for you behind the scenes. For example:

```
var myNumber = 123.765;
```

As with the `String` object, this technique is preferable so long as it's clear to JavaScript what object you expect to have created in the background. So, for example,

```
var myNumber = "123.567";
```

will lead JavaScript to assume, quite rightly, that it's a string, and any attempts to use the `Number` object's methods will fail.

You'll look at just the `toFixed()` method of the `Number` object because that's the most useful method for everyday use.

The `toFixed()` Method

The `toFixed()` method is new to JavaScript 1.5 and JScript 5.5 — so basically it's available in Netscape 6+ and IE 5.5+ only. The method cuts a number off after a certain point. Let's say you want to display a price after sales tax. If your price is \$9.99 and sales tax is 7.5 percent, that means the after-tax cost will be \$10.73925. Well, this is rather an odd amount for a money transaction — what you really want to do is fix the number to no more than two decimal places. Let's create an example.

```
var itemCost = 9.99;
var itemCostAfterTax = 9.99 * 1.075;
document.write("Item cost is $" + itemCostAfterTax + "<br>");
itemCostAfterTax = itemCostAfterTax.toFixed(2);
document.write("Item cost fixed to 2 decimal places is " + itemCostAfterTax);
```

The first `document.write()` will output the following to the page:

```
Item cost is 10.73925
```

However, this is not the format you want; instead you want two decimal places, so on the next line enter this:

```
itemCostAfterTax = itemCostAfterTax.toFixed(2);
```

Chapter 4: JavaScript — An Object-Based Language

You use the `toFixed()` method of the `Number` object to fix the number variable that `itemCost` `AfterTax` holds to two decimal places. The method's only parameter is the number of decimal places you want your number fixed to. This line means that the next `document.write` displays this:

```
Item cost fixed to 2 decimal places is 10.74
```

The first thing you might wonder is why `10.74` and not `10.73`? Well, the `toFixed()` method doesn't just chop off the digits not required; it also rounds up or down. In this case, the number was `10.739`, which rounds up to `10.74`. If it'd been `10.732`, it would have been rounded down to `10.73`.

Note that you can only fix a number from 0 to 20 decimal places.

Because this method is only supported on newer browsers, it's a good idea to check first to see if the browser supports it, like so:

```
var varNumber = 22.234;
if (varNumber.toFixed)
{
    // Browser supports toFixed() method
    varNumber = varNumber.toFixed(2)
}
else
{
    // Browser doesn't support toFixed() method so use some other code
    var div = Math.pow(10,2);
    varNumber = Math.round(varNumber * div) / div;
}
```

Array Objects

You saw how to create and use arrays in Chapter 2, and earlier in this chapter we mentioned that they are actually objects.

As well as storing data, `Array` objects also provide a number of useful properties and methods you can use to manipulate the data in the array and find out information such as the size of the array.

Again, this is not an exhaustive look at every property and method of `Array` objects, but rather just some of the more useful ones.

The length Property — Finding Out How Many Elements Are in an Array

The `length` property gives you the number of elements within an array. You have already seen this in the trivia quiz in Chapter 3. Sometimes you know exactly how long the array is, but there are situations where you may have been adding new elements to an array with no easy way of keeping track of how many have been added.

The `length` property can be used to find the index of the last element in the array. This is illustrated in the following example:

```
var names = new Array();

names[0] = "Paul";
names[1] = "Catherine";
names[11] = "Steve";

document.write("The last name is " + names[names.length - 1]);
```

Note that you have inserted data in the elements with index positions 0, 1, and 11. The array index starts at 0, so the last element is at index `length - 1`, which is 11, rather than the value of the `length` property, which is 12.

Another situation in which the `length` property proves useful is where a JavaScript method returns an array it has built itself. For example, in Chapter 8 on advanced string handling, you'll see that the `String` object has the `split()` method, which splits text into pieces and passes back the result as an `Array` object. Because JavaScript created the array, there is no way for you to know, without the `length` property, what the index is of the last element in the array.

The `concat()` Method — Joining Arrays Together

If you want to take two separate arrays and join them together into one big array, you can use the `Array` object's `concat()` method. The `concat()` method returns a new array, which is the combination of the two arrays: the elements of the first array, then the elements of the second array. To do this, you use the method on your first array and pass the name of the second array as its parameter.

For example, say you have two arrays, `names` and `ages`, and separately they look like this:

names array			
Element Index	0	1	2
Value	Paul	Catherine	Steve

ages array			
Element Index	0	1	2
Value	31	29	34

If you combine them using `names.concat(ages)`, you will get an array like this:

Element Index	0	1	2	3	4	5
Value	Paul	Catherine	Steve	31	29	34

Chapter 4: JavaScript — An Object-Based Language

In the following code, this is exactly what you are doing:

```
var names = new Array("Paul", "Catherine", "Steve");
var ages = new Array(31, 29, 34);

var concatArray;

concatArray = names.concat(ages);
```

It's also possible to combine two arrays into one, but assign the new array to the name of the existing first array, using `names = names.concat(ages)`.

If you were to use `ages.concat(names)`, what would be the difference? Well, as you can see in the following table, the difference is that now the `ages` array elements are first, and the elements from the `names` array are concatenated on the end.

Element Index	0	1	2	3	4	5
Value	31	29	34	Paul	Catherine	Steve

The `slice()` Method — Copying Part of an Array

When you just want to copy a portion of an array, you can use the `slice()` method. Using the `slice()` method, you can slice out a portion of the array and assign that to a new variable name. The `slice()` method has two parameters:

- ❑ The index of the first element you want copied
- ❑ The index of the element marking the end of the portion you are slicing out (optional)

Just as with string copying with `substr()` and `substring()`, the start point is included in the copy, but the end point is not. Again, if you don't include the second parameter, all elements from the start index onward are copied.

Suppose you have the array `names` shown in the following table.

Index	0	1	2	3	4
Value	Paul	Sarah	Louise	Adam	Bob

If you want to create a new array with elements 1, Sarah, and 2, Louise, you would specify a start index of 1 and an end index of 3. The code would look something like this:

```
var names = new Array("Paul", "Sarah", "Louise", "Adam", "Bob");
var slicedArray = names.slice(1, 3);
```


Note that when JavaScript copies the array, it copies the new elements to an array in which they have indexes 0 and 1, and not their old indexes of 1 and 2. After slicing, the `slicedArray` looks like this:

Index	0	1
Value	Sarah	Louise

The first array, `names`, is unaffected by the slicing.

The `join()` Method — Converting an Array into a Single String

The `join()` method concatenates all the elements in an array and returns them as a string. It also enables you to specify any characters you want to insert *between* elements as they are joined together. The method has only one parameter, and that's the string you want between elements.

An example will help explaining things. Imagine that you have your weekly shopping list stored in an array, which looks something like this:

Index	0	1	2	3	4
Value	Eggs	Milk	Potatoes	Cereal	Banana

Now you want to write out your shopping list to the page using `document.write()`. You want each item to be on a different line, so this means you need to use the `
` tag between each element. First, you need to declare your array.

```
var myShopping = new Array("Eggs", "Milk", "Potatoes", "Cereal", "Banana");
```

Now, convert the array into one string with the `join()` method.

```
var myShoppingList = myShopping.join("<br>");
```

Now the variable `myShoppingList` will hold the following text:

```
"Eggs<br>Milk<br>Potatoes<br>Cereal<br>Banana"
```

which you can write out to the page with `document.write()`.

```
document.write(myShoppingList);
```

The shopping list will appear in the page with each item on a new line, as shown in Figure 4-3.

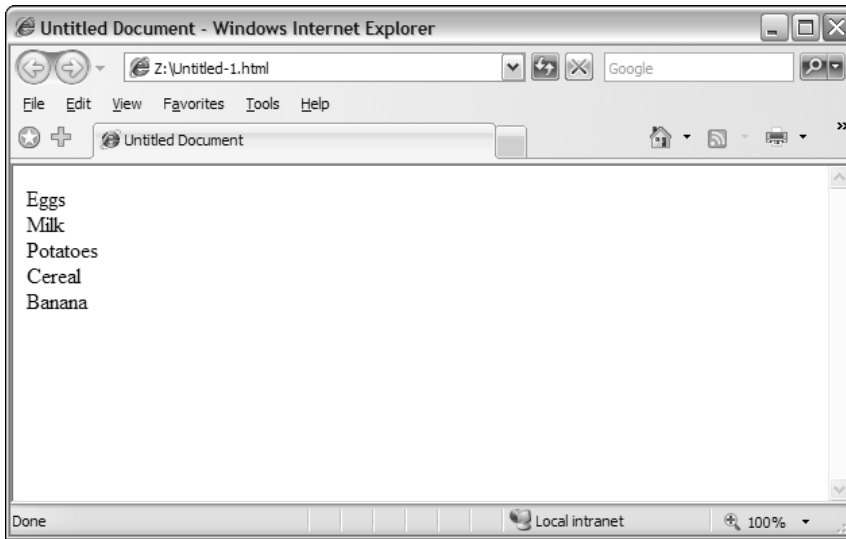


Figure 4-3

The `sort()` Method — Putting Your Array in Order

If you have an array that contains similar data, such as a list of names or a list of ages, you may want to put them in alphabetical or numerical order. This is something that the `sort()` method makes very easy. In the following code, you define your array and then put it in ascending alphabetical order using `names.sort()`. Finally, you output it so that you can see that it's in order.

```
var names = new Array("Paul", "Sarah", "Louise", "Adam", "Bob");
var elementIndex;

names.sort();
document.write("Now the names again in order" + "<BR>");

for (elementIndex = 0; elementIndex < names.length; elementIndex++)
{
    document.write(names[elementIndex] + "<BR>");
}
```

Don't forget that the sorting is case sensitive, so `Paul` will come before `paul`. Remember that JavaScript stores letters encoded in their equivalent Unicode number, and that sorting is done based on Unicode numbers rather than actual letters. It just happens that Unicode numbers match the order in the alphabet. However, lowercase letters are given a different sequence of numbers, which come after the uppercase letters. So the array with elements `Adam`, `adam`, `Zoë`, `zoë`, will be sorted to the order `Adam`, `Zoë`, `adam`, `zoë`.

Note that in your `for` statement you've used the `Array` object's `length` property in the condition statement, rather than inserting the length of the array (5), like this:

```
for (elementIndex = 0; elementIndex < 5; elementIndex++)
```

Why do this? After all, you know in advance that there are five elements in the array. Well, what would happen if you altered the number of elements in our array by adding two more names?

```
var names = new Array("Paul", "Sarah", "Louise", "Adam", "Bob", "Karen", "Steve");
```

If you had inserted 5 rather than `names.length`, your loop code wouldn't work as you want it to. It wouldn't display the last two elements unless you changed the condition part of the `for` loop to 7. By using the `length` property, you've made life easier, because now there is no need to change code elsewhere if you add array elements.

Okay, you've put things in ascending order, but what if you wanted descending order? That is where the `reverse()` method comes in.

The `reverse()` Method — Putting Your Array into Reverse Order

The final method you'll look at for the `Array` object is the `reverse()` method, which — no prizes for guessing — reverses the order of the array so that the elements at the back are moved to the front. Let's take the shopping list again as an example.

Index	0	1	2	3	4
Value	Eggs	Milk	Potatoes	Cereal	Banana

If you use the `reverse()` method

```
var myShopping = new Array("Eggs", "Milk", "Potatoes", "Cereal", "Banana");  
myShopping.reverse();
```

you end up with the array elements in this order:

Index	0	1	2	3	4
Value	Banana	Cereal	Potatoes	Milk	Eggs

To prove this you could write it to the page with the `join()` method you saw earlier.

```
var myShoppingList = myShopping.join("<br>");  
document.write(myShoppingList);
```

Try It Out Sorting an Array

When used in conjunction with the `sort()` method, the `reverse()` method can be used to sort an array so that its elements appear in reverse alphabetical or numerical order. This is shown in the following example:

```
<html>
<body>
<script language="JavaScript" type="text/javascript">

var myShopping = new Array("Eggs", "Milk", "Potatoes", "Cereal", "Banana");

var ord = prompt("Enter 1 for alphabetical order, and -1 for reverse order", 1);

if (ord == 1)
{
    myShopping.sort();
    document.write(myShopping.join("<br>"));
}
else if (ord == -1)
{
    myShopping.sort();
    myShopping.reverse();
    document.write(myShopping.join("<br>"));
}
else
{
    document.write("That is not a valid input");
}
</script>
</body>
</html>
```

Save the example as `ch4_examp5.htm`. When you load this into your browser, you will be asked to enter some input depending on whether you want the array to be ordered in forward or backward order. If you enter 1, the array will be displayed in forward order. If you enter -1, the array will be displayed in reverse order. If you enter neither of these values, you will be told that your input was invalid.

How It Works

At the top of the script block you define the array containing your shopping list. Next you define the variable `ord` to be the value entered by the user in a prompt box.

```
var ord = prompt("Enter 1 for alphabetical order, and -1 for reverse order", 1);
```

This value is used in the conditions of the `if` statements that follow. The first `if` checks whether the value of `ord` is 1 — that is, whether the user wants the array in alphabetical order. If so, the following code is executed:

```
myShopping.sort();
document.write(myShopping.join("<br>"));
```

The array is sorted and then displayed to the user on separate lines using the `join()` method. Next, in the `else if` statement, you check whether the value of `ord` is `-1` — that is, whether the user wants the array in reverse alphabetical order. If so, the following code is executed:

```
myShopping.sort();
myShopping.reverse();
document.write(myShopping.join("<br>"));
```

Here, you sort the array before reversing its order. Again the array is displayed to the user by means of the `join()` method.

Finally, if `ord` has neither the value `1` nor the value `-1`, you tell the user that his input was invalid.

```
document.write("That is not a valid input");
```

Date Objects

The `Date` object handles everything to do with date and time in JavaScript. Using it, you can find out the date and time now, store your own dates and times, do calculations with these dates, and convert the dates into strings.

The `Date` object has a lot of methods and can be a little tricky to use, which is why Chapter 9 is dedicated to the date, time, and timers in JavaScript. You'll also see in Chapter 11 how you can use dates to determine if there's been anything new added to the web site since the user last visited it. However, in this section you'll focus on how to create a `Date` object and some of its more commonly used methods.

Creating a Date Object

You can declare and initialize a `Date` object in four ways. In the first method, you simply declare a new `Date` object without initializing its value. In this case, the date and time value will be set to the current date and time on the PC on which the script is run.

```
var theDate1 = new Date();
```

Secondly, you can define a `Date` object by passing the number of milliseconds since January 1, 1970, at 00:00:00 GMT. In the following example, the date is 31 January 2000 00:20:00 GMT (that is, 20 minutes past midnight).

```
var theDate2 = new Date(949278000000);
```

It's unlikely that you'll be using this way of defining a `Date` object very often, but this is how JavaScript actually stores the dates. The other formats for giving a date are simply for our convenience.

Next, you can pass a string representing a date, or a date and time. In the following example, you have "31 January 2010".

```
var theDate3 = new Date("31 January 2010");
```

However, you could have written 31 Jan 2010, Jan 31 2010, 01-31-2010, or any of a number of valid variations you'd commonly expect when writing down a date normally — if in doubt, try it out. Note

Chapter 4: JavaScript — An Object-Based Language

that Firefox and Netscape browsers don't support the string "01-31-2010" as a valid date format. If you are writing your web pages for an international audience outside the United States, you need to be aware of the different ways of specifying dates. In the United Kingdom and many other places, the standard is day, month, year, whereas in the United States the standard is month, day, year. This can cause problems if you specify only numbers — JavaScript may think you're referring to a day when you mean a month. The easiest way to avoid such headaches is to, where possible, always use the name of the month. That way there can be no confusion.

In the fourth and final way of defining a Date object, you initialize it by passing the following parameters separated by commas: year, month, day, hours, minutes, seconds, and milliseconds. For example:

```
var theDate4 = new Date(2010,0,31,15,35,20,20);
```

This date is actually 31 January 2010 at 15:35:20 and 20 milliseconds. You can specify just the date part if you wish and ignore the time.

Something to be aware of is that in this instance January is month 0, not month 1, as you'd expect, and December is month 11. It's very easy to make a mistake when specifying a month.

Getting Date Values

It's all very nice having stored a date, but how do you get the information out again? Well, you just use the `get` methods. These are summarized in the following table.

Method	Returns
<code>getDate()</code>	The day of the month
<code>getDay()</code> and so on	The day of the week as an integer, with Sunday as 0, Monday as 1,
<code>getMonth()</code>	The month as an integer, with January as 0, February as 1, and so on
<code>getFullYear()</code>	The year as a four-digit number
<code>toString()</code>	Returns the full date based on the current time zone as a human-readable string. For example "Wed 31 Dec 2003".

For example, if you want to get the month in `ourDateObj`, you can simply write the following:

```
theMonth = myDateObject.getMonth();
```

All the methods work in a very similar way, and all values returned are based on local time, meaning time local to the machine the code is running on. It's also possible to use Universal Time, previously known as GMT, which we'll discuss in Chapter 9.

Try It Out

Using the Date Object to Retrieve the Current Date

In this example, you use the `get date` type methods you have been looking at to write the current day, month, and year to a web page.

```
<html>
<body>

<script language="JavaScript" type="text/javascript">

var months = new Array("January", "February", "March", "April", "May", "June", "July",
                        "August", "September", "October", "November", "December");
var dateNow = new Date();
var yearNow = dateNow.getFullYear();
var monthNow = months[dateNow.getMonth()];
var dayNow = dateNow.getDate();
var daySuffix;

switch (dayNow)
{
case 1:
case 21:
case 31:
    daySuffix = "st";
    break;
case 2:
case 22:
    daySuffix = "nd";
    break;
case 3:
case 23:
    daySuffix = "rd";
    break;
default:
    daySuffix = "th";
    break;
}

document.write("It is the " + dayNow + daySuffix + " day ");
document.write("in the month of " + monthNow);
document.write(" in the year " + yearNow);

</script>
</body>
</html>
```

Save the code as `ch4_examp6.htm`. If you load up the page, you should see a correctly formatted sentence telling you what the current date is.

How It Works

The first thing you do in the code is declare an array and populate it with the months of a year. Why do this? Well, there is no method of the `Date` object that'll give you the month by name instead of as a number. However, this poses no problem; you just declare an array of months and use the month number as the array index to select the correct month name.

```
var months = new Array("January", "February", "March", "April", "May", "June", "July",
                        "August", "September", "October", "November", "December");
```

Chapter 4: JavaScript — An Object-Based Language

Next you create a new `Date` object and by not initializing it with your own value, you allow it to initialize itself to the current date and time.

```
var dateNow = new Date();
```

Following this you set the `yearNow` variable to the current year, as returned by the `getFullYear()` method.

```
var yearNow = dateNow.getFullYear();
```

Note that `getFullYear()` only became available with version 4 browsers, such as IE 4 and NN 4.06 and above. Prior to this, there was only the `getYear()` method, which on some browsers returned only a two-digit year.

You then populate your `monthNow` variable with the value contained in the array element with an index of the number returned by `getMonth()`. Remember that `getMonth()` returns the month as an integer value, starting with 0 for January — this is a bonus because arrays also start at 0, so no adjustment is needed to find the correct array element.

```
var monthNow = months[dateNow.getMonth()];
```

Finally, the current day of the month is put into variable `dayNow`.

```
var dayNow = dateNow.getDate();
```

Next you use a `switch` statement, which you learned about in the last chapter. This is a useful technique for adding the correct suffix to the date that you already have. After all, your application will look more professional if you can say "it is the 1st day", rather than "it is the 1 day". This is a little tricky, however, because the suffix you want to add depends on the number that precedes it. So, for the first, twenty-first, and thirty-first days of the month, you have this:

```
switch (dayNow)
{
  case 1:
  case 21:
  case 31:
    daySuffix = "st";
    break;
```

For the second and twenty-second days, you have this:

```
case 2:
case 22:
  daySuffix = "nd";
  break;
```

and for the third and twenty-third days, you have this:

```
case 3:
case 23:
  daySuffix = "rd";
  break;
```


Finally, you need the `default` case for everything else. As you will have guessed by now, this is simply `"th"`.

```
default:
    daySuffix = "th";
    break;
}
```

In the final lines you simply write the information to the HTML page, using `document.write()`.

Setting Date Values

To change part of the date in a `Date` object, you have a group of `set` functions, which pretty much replicate the `get` functions described earlier, except that you are setting, not getting, the values. These functions are summarized in the following table.

Method	Description
<code>setDate()</code>	The date of the month is passed in as the parameter to set the date
<code>setMonth()</code>	The month of the year is passed in as an integer parameter, where 0 is January, 1 is February, and so on
<code>setFullYear()</code>	This sets the year to the four-digit integer number passed in as a parameter

Note that for security reasons, there is no way for web-based JavaScript to change the current date and time on a user's computer.

So, to change the year to 2009, the code would be as follows:

```
myDateObject.setFullYear(2009);
```

Setting the date and month to the twenty-seventh of February looks like this:

```
myDateObject.setDate(27);
myDateObject.setMonth(1);
```

One minor point to note here is that there is no direct equivalent of the `getDay()` method. After the year, date, and month have been defined, the day is automatically set for you.

Calculations and Dates

Take a look at the following code:

```
var myDate = new Date("1 Jan 2010");
myDate.setDate(32);
document.write(myDate);
```

Surely there is some error — since when has January had 32 days? The answer is that of course it doesn't, and JavaScript knows that. Instead JavaScript sets the date to 32 days from the first of January — that is, it sets it to the first of February.

Chapter 4: JavaScript — An Object-Based Language

The same also applies to the `setMonth()` method. If you set it to a value greater than 11, the date automatically rolls over to the next year. So if you use `setMonth(12)`, that will set the date to January of the next year, and similarly `setMonth(13)` is February of the next year.

How can you use this feature of `setDate()` and `setMonth()` to your advantage? Well, let's say you want to find out what date it will be 28 days from now. Given that different months have different numbers of days and that you could roll over to a different year, it's not as simple a task as it might first seem. Or at least that would be the case if it were not for `setDate()`. The code to achieve this task is as follows:

```
var nowDate = new Date();
var currentDay = nowDate.getDate();
nowDate.setDate(currentDay + 28);
```

First you get the current system date by setting the `nowDate` variable to a new `Date` object with no initialization value. In the next line you put the current day of the month into a variable called `currentDay`. Why? Well, when you use `setDate()` and pass it a value outside of the maximum number of days for that month, it starts from the first of the month and counts that many days forward. So, if today's date is the January 15 and you use `setDate(28)`, it's not 28 days from the fifteenth of January, but 28 days from the first of January. What you want is 28 days from the current date, so you need to add the current date to the number of days ahead you want. So you want `setDate(15 + 28)`. In the third line you set the date to the current date, plus 28 days. You stored the current day of the month in `currentDay`, so now you just add 28 to that to move 28 days ahead.

If you want the date 28 days prior to the current date, you just pass the current date minus 28. Note that this will most often be a negative number. You need to change only one line, and that's the third one, which you change to

```
nowDate.setDate(currentDay - 28);
```

You can use exactly the same principles for `setMonth()` as you have used for `setDate()`.

Getting Time Values

The methods you use to retrieve the individual pieces of time data work much like the `get` methods for date values. The methods you use here are:

- ☐ `getHours()`
- ☐ `getMinutes()`
- ☐ `getSeconds()`
- ☐ `getMilliseconds()`
- ☐ `toString()`

These methods return respectively the hours, minutes, seconds, milliseconds, and full time of the specified `Date` object, where the time is based on the 24-hour clock: 0 for midnight and 23 for 11 p.m. The last method is similar to the `toString()` method in that it returns an easily readable string, except that in this case it contains the time (for example, "13:03:51 UTC").

Note that the `getMilliseconds()` method is available only in IE 4+ and NN 4.06+ browsers.

Try It Out Writing the Current Time into a Web Page

Let's look at an example that writes out the current time to the page.

```
<html>
<body>
<script language="JavaScript" type="text/javascript">

var greeting;

var nowDate = new Date();
var nowHour = nowDate.getHours();
var nowMinute = nowDate.getMinutes();
var nowSecond = nowDate.getSeconds();

if (nowMinute < 10)
{
    nowMinute = "0" + nowMinute;
}

if (nowSecond < 10)
{
    nowSecond = "0" + nowSecond;
}

if (nowHour < 12)
{
    greeting = "Good Morning";
}
else if (nowHour < 17)
{
    greeting = "Good Afternoon";
}
else
{
    greeting = "Good Evening";
}

document.write("<h4>" + greeting + " and welcome to my website</h4>");
document.write("According to your clock the time is ");
document.write(nowHour + ":" + nowMinute + ":" + nowSecond);

</script>
</body>
</html>
```

Save this page as `ch4_examp7.htm`. When you load it into a web browser, it writes a greeting based on the time of day as well as the current time, as shown in Figure 4-4.

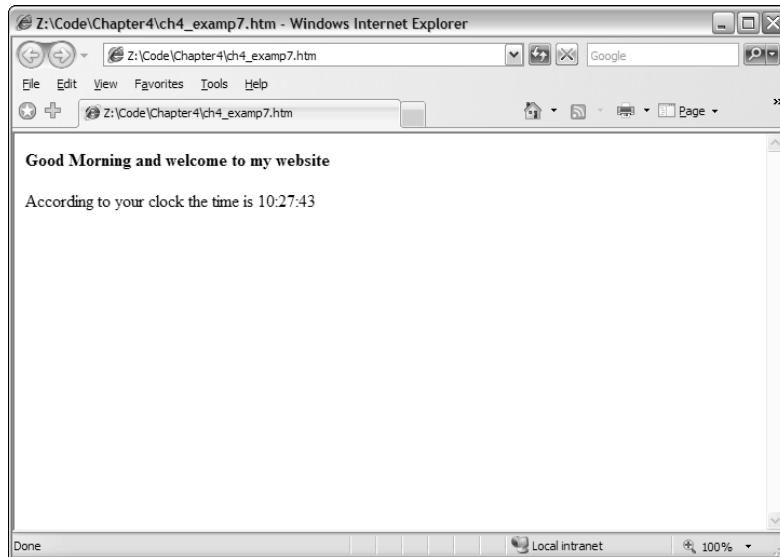


Figure 4-4

How It Works

The first two lines of code declare two variables—`greeting` and `nowDate`.

```
var greeting;  
var nowDate = new Date();
```

The `greeting` variable will be used shortly to store the welcome message on the web site, whether this is "Good Morning", "Good Afternoon", or "Good Evening". The `nowDate` variable is initialized to a new `Date` object. Note that the constructor for the `Date` object is empty, so JavaScript will store the current date and time in it.

Next, you get the information on the current time from `nowDate` and store it in various variables. You can see that getting time data is very similar to getting date data, just using different methods.

```
var nowHour = nowDate.getHours();  
var nowMinute = nowDate.getMinutes();  
var nowSecond = nowDate.getSeconds();
```

You may wonder why the following lines are included in the example:

```
if (nowMinute < 10)  
{  
    nowMinute = "0" + nowMinute;  
}  
  
if (nowSecond < 10)  
{  
    nowSecond = "0" + nowSecond;  
}
```

These lines are there just for formatting reasons. If the time is nine minutes past 10, then you expect to see something like 10:09. You don't expect 10:9, which is what you would get if you used the `getMinutes()` method without adding the extra zero. The same goes for seconds. If you're just using the data in calculations, you don't need to worry about formatting issues — you do here because you're inserting the time the code executed into the web page.

Next, in a series of `if` statements, you decide (based on the time of day) which greeting to create for displaying to the user.

```
if (nowHour < 12)
{
    greeting = "Good Morning";
}
else if (nowHour < 17)
{
    greeting = "Good Afternoon";
}
else
{
    greeting = "Good Evening";
}
```

Finally, you write out the greeting and the current time to the page.

```
document.write("<h4>" + greeting + " and welcome to my website</h4>");
document.write("According to your clock the time is ");
document.write(nowHour + ":" + nowMinute + ":" + nowSecond);
```

You'll see in Chapter 9 on dates, times, and timers how you can write a continuously updating time to the web page, making it look like a clock.

Setting Time Values

When you want to set the time in your `Date` objects, you have a series of methods similar to those used for getting the time:

- ☐ `setHours()`
- ☐ `setMinutes()`
- ☐ `setSeconds()`
- ☐ `setMilliseconds()`

These work much like the methods you use to set the date, in that if you set any of the time parameters to an illegal value, JavaScript assumes you mean the next or previous time boundary. If it's 9:57 and you set minutes to 64, the time will be set to 10:04 — that is, 64 minutes from 9:00.

This is demonstrated in the following code:

```
var nowDate = new Date();
nowDate.setHours(9);
nowDate.setMinutes(57);
```

```
alert(nowDate);

nowDate.setMinutes(64);
alert(nowDate);
```

First you declare the `nowDate` variable and assign it to a new `Date` object, which will contain the current date and time. In the following two lines you set the hours to 9 and the minutes to 57. You show the date and time using an `alert` box, which should show a time of 9:57. The minutes are then set to 64 and again an alert box is used to show the date and time to the user. Now the minutes have rolled over the hour so the time shown should be 10:04.

If the hours were set to 23 instead of 9, setting the minutes to 64 would not just move the time to another hour but also cause the day to change to the next date.

JavaScript Classes

In this section you'll be looking at some quite tricky and advanced stuff. It's not essential stuff, so you may want to move on and come back to it later.

You've seen that JavaScript provides a number of objects built into the language and ready for us to use. It's a bit like a house that's built already and you can just move on in. However, what if you want to create your own house, to design it for your own specific needs? In that case you'll use an architect to create technical drawings and plans that provide the template for the new house — the builders use the plans to tell them how to create the house.

So what does any of this have to do with JavaScript and objects? Well, JavaScript enables you to be an architect and create the templates for your own objects to your own specification, to fill your specific needs. Let's say, for example, you were creating a cinema booking system. JavaScript doesn't come with any built-in cinema booking objects, so you'd have to design your own. What you need to do is create objects modeled around the real world. So for a simple cinema booking system, you might have an object representing customers' booking details and an object for the cinema where the bookings have been made. As well as being able to store information, you can create your own methods for an object. So for a booking system, you might want an "add new booking" method or a method that gets the details of all the bookings currently made.

Where you have no need to store data but simply want functionality, such as the `fix()` function you saw before, it's generally easier just to have a code library rather than to create a special object.

Just as a builder of a house needs an architect's plans to know what to build and how it should be laid out, you need to provide blueprints telling JavaScript how your object should look. For example, you need to define its methods and provide the code for those methods. The key to this is JavaScript's support for the definition of classes. Classes are essentially templates for an object, as the architect's drawings are the template used to build a house. Before you can use your new object type, you need to define its class, methods, and properties. The important distinction is that when you define your class, no object based on that class is created. It's only when you create an instance of your class using the `new` keyword that an object of that class type, based on your class blueprint or prototype, is created.

A class consists of three things:

- ❑ A constructor
- ❑ Method definitions
- ❑ Properties

A constructor is a method that is called every time one of your objects based on this class is created. It's useful when you want to initialize properties or the object in some way. You need to create a constructor even if you don't pass any parameters to it or it contains no code. (In that case it'd just be an empty definition.) As with functions, a constructor can have zero or more parameters.

You used methods when you used JavaScript's built-in objects; now you get the chance to use classes to define your own methods performing specific tasks. Your class will specify what methods you have and the code that they execute. Again, you have used properties of built-in objects before and now get to define your own. You don't need to declare your class's properties. You can simply go ahead and use properties in your class without letting JavaScript know in advance.

Let's create a simple class based on the real-world example of a cinema booking system.

Defining a Class

Let's start by creating a class for a customer's booking. This class will be called the `CustomerBooking` class. The first thing you need to do is create the class constructor.

The constructor for your class is shown here:

```
function CustomerBooking (bookingId, customerName, film, showDate)
{
  this.customerName = customerName;
  this.bookingId = bookingId;
  this.showDate = showDate;
  this.film = film;
}
```

Your first thought might be that what you have here is simply a function, and you'd be right. It's not until you start defining the `CustomerBooking` class properties and methods that it becomes a class. This is in contrast to some programming languages, which have a more formal way of defining classes.

When you look at the code, the important thing to note is that the constructor function's name must match that of the class you are defining — in this case the `CustomerBooking` class. That way, when a new instance of your class as an object (termed an *object instance*) is created, this function will be called automatically. Note that you have four parameters for your constructor function, and that these are used inside the class itself. However, note that you use the `this` keyword. For example:

```
this.customerName = customerName;
```

Inside a constructor function or within a class method, the `this` keyword will refer to that object instance of your class. Here you refer to the `customerName` property of this class object, and you set it to

Chapter 4: JavaScript — An Object-Based Language

equal the `customerName` parameter. If you have used other object-oriented programming languages, you might wonder where you defined this `customerName` property. The answer is that you didn't; simply by assigning a property a value, JavaScript creates it for you. There is no check that the property exists; JavaScript creates it as it needs to. The same is true if you use the object with a property never mentioned in your class definition. All this free property creation might sound great, but it has drawbacks, the main one being that JavaScript won't tell you if you accidentally misspell a property name; it'll just create a new property with the misspelled name, something that can make it difficult to track bugs. One way around this problem is to create methods that get a property's value and enable you to set a property's value. Now this may sound like hard work, but it can reduce bugs or at least make them easier to spot. Let's create a few property `get/set` methods for the `CustomerBooking` class.

```
CustomerBooking.prototype.getCustomerName = function()
{
    return this.customerName;
}

CustomerBooking.prototype.setCustomerName = function(customerName)
{
    this.customerName = customerName;
}

CustomerBooking.prototype.getShowDate = function()
{
    return this.showDate;
}

CustomerBooking.prototype.setShowDate = function(showDate)
{
    this.showDate = showDate;
}

CustomerBooking.prototype.getFilm = function()
{
    return this.film;
}

CustomerBooking.prototype.setFilm = function(film)
{
    this.film = film;
}

CustomerBooking.prototype.getBookingId = function()
{
    return this.bookingId;
}

CustomerBooking.prototype.setBookingId = function(bookingId)
{
    this.bookingId = bookingId;
}
```


Now you have defined a `set` and `get` method for each of your class's four properties: `bookingId`, `film`, `customerName`, and `showDate`. Let's look at how you created one of the methods, the `getCustomerName()` method.

```
CustomerBooking.prototype.getCustomerName = function()
{
    return this.customerName;
}
```

The first thing you notice is that this is a very odd way of defining a function. On the left you set the class's prototype property's `getCustomerName` to equal a function, which you then define immediately afterwards. In fact, JavaScript supplies most objects with a `prototype` property, which allows new properties and methods to be created. So whenever you want to create a method for your class, you simply write the following:

```
className.prototype.methodName = function(method parameter list)
{
    // method code
}
```

You've created your class, but how do you now create new objects based on that class? Well, you look at this in the next section.

Creating and Using Class Object Instances

You create instances of your classes in the same way you created instances of built-in JavaScript classes: using the `new` keyword. So to create a new instance of your `CustomerBooking` class, you'd write this:

```
var firstBooking = new
    CustomerBooking(1234, "Robert Smith", "Raging Bull", "25 July 2004 18:20");

var secondBooking = new
    CustomerBooking(1244, "Arnold Palmer", "Toy Story", "27 July 2004 20:15");
```

Here, as with a `String` object, you have created two new objects and stored them in variables, `firstBooking` and `secondBooking`, but this time it's a new object based on your class.

Let's call the `getCustomerName()` method of each of the two objects and write the results to the page.

```
document.write("1st booking person's name is " +
    firstBooking.getCustomerName() + "<br>");
document.write("2nd booking person's name is " +
    secondBooking.getCustomerName());
```

And you'll see the following written into the page from information contained in your class objects:

```
1st booking person's name is Robert Smith
2nd booking person's name is Arnold Palmer
```

Chapter 4: JavaScript — An Object-Based Language

Now let's put this together in a page.

```
<html>
<body>

<script language="JavaScript" type="text/javascript">

// CustomerBooking class

function CustomerBooking(bookingId, customerName, film, showDate)
{
    this.customerName = customerName;
    this.bookingId = bookingId;
    this.showDate = showDate;
    this.film = film;
}

CustomerBooking.prototype.getCustomerName = function()
{
    return this.customerName;
}

CustomerBooking.prototype.setCustomerName = function(customerName)
{
    this.customerName = customerName;
}

CustomerBooking.prototype.getShowDate = function()
{
    return this.showDate;
}

CustomerBooking.prototype.setShowDate = function(showDate)
{
    this.showDate = showDate;
}

CustomerBooking.prototype.getFilm = function()
{
    return this.film;
}

CustomerBooking.prototype.setFilm = function(film)
{
    this.film = film;
}

CustomerBooking.prototype.getBookingId = function()
{
    return this.bookingId;
}

CustomerBooking.prototype.setBookingId = function(bookingId)
{

```

```
    this.bookingId = bookingId;
  }

  var firstBooking = new CustomerBooking(1234,
                                          "Robert Smith", "Raging Bull", "25 July 2004 18:20");
  var secondBooking = new CustomerBooking(1244,
                                          "Arnold Palmer", "Toy Story", "27 July 2004 20:15");
  document.write("1st booking persons name is " +
                 firstBooking.getCustomerName() + "<br>");
  document.write("2nd booking persons name is " +
                 secondBooking.getCustomerName());

</script>

</body>
</html>
```

At the top of the page is your `<script>` tag, inside of which is the code that defines your class. You must include class definition code in every page that uses your class to create objects. For convenience, you may therefore decide to put your class definitions in a separate file and import that file into each page that uses the class. You can do this using the `<script>` tag, but instead of putting the code inside the open and close tags, you'll use the script tag's `src` attribute to point to the file containing the JavaScript. For example, if you create a file called `MyCinemaBookingClasses.js` and put your class code in there, you can import it into a page as shown here:

```
<script language="JavaScript" src="MyCinemaBookingClasses.js"></script>
```

The `src` attribute points to the URL of your class, which in this case assumes that the class's `.js` file is in the same directory as your page.

An Array of Items

So far you have a class for items that you can put a single booking into, but no class representing all the bookings taken by a cinema. So how can you create a cinema class that supports the storage of zero or more items? The answer is using an array, which we discuss in Chapter 3.

Let's start by defining your class, which you'll call the `cinema` class, and add to the script block with your `CustomerBooking` class.

```
// cinema class

function cinema()
{
    this.bookings = new Array();
}
```

Here you define the constructor. Inside the constructor, you initialize the `bookings` property that will hold all the `CustomerBooking` class objects.

Chapter 4: JavaScript — An Object-Based Language

Next you need to add a way of making bookings for the cinema; for this you create the `addBooking()` method.

```
cinema.prototype.addBooking = function(bookingId, customerName, film, showDate)
{
    this.bookings[bookingId] = new CustomerBooking(bookingId,
                                                    customerName, film, showDate);
}
```

The method takes four parameters, the details needed to create a new booking. Then, inside the method, you create a new object of type `CustomerBooking`. A reference to this object is stored inside your `bookings` array, using the unique `bookingId` to associate the place in which the new object is stored.

Let's look at how you can access the items in the array. In the following method, called `getBookingsTable()`, you go through each booking in the cinema and create the HTML necessary to display all the bookings in a table.

```
cinema.prototype.getBookingsTable = function()
{
    var booking;
    var bookingsTableHTML = "<table border=1>";

    for (booking in this.bookings)
    {
        bookingsTableHTML += "<tr><td>";
        bookingsTableHTML += this.bookings[booking].getBookingId();
        bookingsTableHTML += "</td>";

        bookingsTableHTML += "<td>";
        bookingsTableHTML += this.bookings[booking].getCustomerName();
        bookingsTableHTML += "</td>";

        bookingsTableHTML += "<td>";
        bookingsTableHTML += this.bookings[booking].getFilm();
        bookingsTableHTML += "</td>";

        bookingsTableHTML += "<td>";
        bookingsTableHTML += this.bookings[booking].getShowDate();
        bookingsTableHTML += "</td>";
        bookingsTableHTML += "</tr>";
    }
    bookingsTableHTML += "</table>";
    return bookingsTableHTML;
}
```

You can access each booking by its unique `bookingId`, but what you want to do is simply loop through all the bookings for the cinema, so you use a `for...in` loop, which will loop through each item in the `items` array. Each time the loop executes, `booking` will be set by JavaScript to contain the `bookingId` of the next booking; it doesn't contain the item itself but its associated keyword.

Since you have the associated keyword, you can access the item objects in the array like this:

```
this.bookings[booking]
```

Remember that this refers to the object instance of your class. You then use the `CustomerBooking` object's `get` methods to obtain the details for each booking. Finally, on the last line, you return the HTML — with your summary of all the bookings — to the calling code.

Let's put this all together in a page and save the page as `ch4_examp8.htm`.

```
<html>
<body>

<h2>Summary of bookings</h2>

<script language="JavaScript" type="text/javascript">

// CustomerBooking class

function CustomerBooking(bookingId, customerName, film, showDate)
{
    this.customerName = customerName;
    this.bookingId = bookingId;
    this.showDate = showDate;
    this.film = film;
}

CustomerBooking.prototype.getCustomerName = function()
{
    return this.customerName;
}

CustomerBooking.prototype.setCustomerName = function(customerName)
{
    this.customerName = customerName;
}

CustomerBooking.prototype.getShowDate = function()
{
    return this.showDate;
}

CustomerBooking.prototype.setShowDate = function(showDate)
{
    this.showDate = showDate;
}

CustomerBooking.prototype.getFilm = function()
{
    return this.film;
}

CustomerBooking.prototype.setFilm = function(film)
{
    this.film = film;
}

CustomerBooking.prototype.getBookingId = function()
{

```

Chapter 4: JavaScript — An Object-Based Language

```
    return this.bookingId;
}

CustomerBooking.prototype.setBookingId = function(bookingId)
{
    this.bookingId = bookingId;
}

// cinema class

function cinema()
{
    this.bookings = new Array();
}

cinema.prototype.addBooking = function(bookingId, customerName, film, showDate)
{
    this.bookings[bookingId] = new CustomerBooking(bookingId,
                                                    customerName, film, showDate);
}

cinema.prototype.getBookingsTable = function()
{
    var booking;
    var bookingsTableHTML = "<table border=1>";

    for (booking in this.bookings)
    {
        bookingsTableHTML += "<tr><td>";
        bookingsTableHTML += this.bookings[booking].getBookingId();
        bookingsTableHTML += "</td>";

        bookingsTableHTML += "<td>";
        bookingsTableHTML += this.bookings[booking].getCustomerName();
        bookingsTableHTML += "</td>";

        bookingsTableHTML += "<td>";
        bookingsTableHTML += this.bookings[booking].getFilm();
        bookingsTableHTML += "</td>";

        bookingsTableHTML += "<td>";
        bookingsTableHTML += this.bookings[booking].getShowDate();
        bookingsTableHTML += "</td>";
        bookingsTableHTML += "</tr>";
    }
    bookingsTableHTML += "</table>";
    return bookingsTableHTML;
}

var londonOdeon = new cinema();
londonOdeon.addBooking(342, "Arnold Palmer", "Toy Story", "15 July 2009 20:15");
londonOdeon.addBooking(335, "Louise Anderson", "The Shawshank Redemption", "27 July
2009 11:25");
londonOdeon.addBooking(566, "Catherine Hughes",
```

```
        "Never Say Never", "27 July 2009 17:55");  
londonOdeon.addBooking(324, "Beci Smith",  
        "Shrek", "29 July 2009 20:15");  
  
document.write(londonOdeon.getBookingsTable());  
  
</script>  
  
</body>  
</html>
```

Your new code is the lines

```
var londonOdeon = new cinema();  
londonOdeon.addBooking(342, "Arnold Palmer", "Toy Story", "15 July 2009 20:15");  
londonOdeon.addBooking(335, "Louise Anderson",  
        "The Shawshank Redemption", "27 July 2009 11:25");  
londonOdeon.addBooking(566, "Catherine Hughes",  
        "Never Say Never", "27 July 2009 17:55");  
londonOdeon.addBooking(324, "Beci Smith", "Shrek", "29 July 2009 20:15");  
  
document.write(londonOdeon.getBookingsTable());
```

These create a new `cinema` object and store a reference to it in the variable `londonOdeon`. You then create four new bookings using the `cinema` class's `addBooking()` method. On the final line, you write the HTML returned by the `getBookingsTable()` method to the page.

Your page should now look like that shown in Figure 4-5.

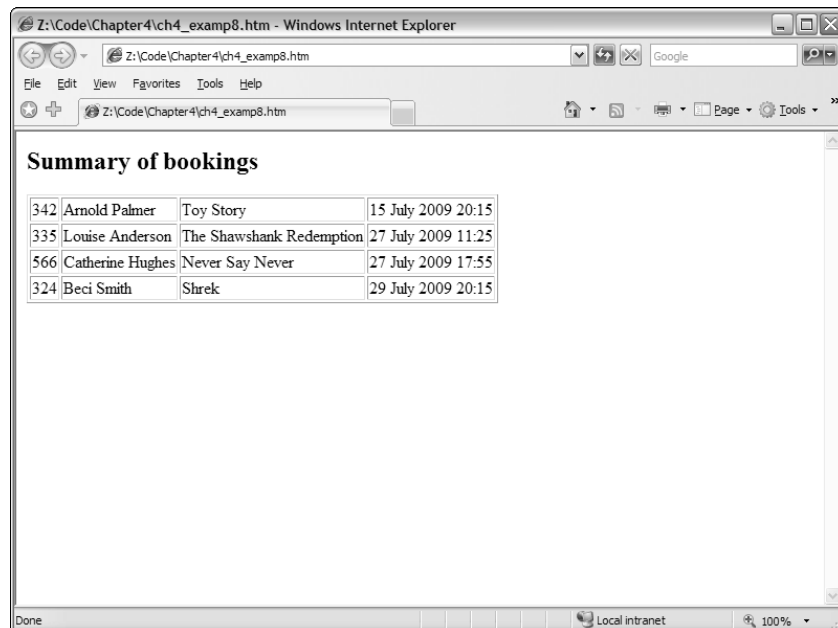


Figure 4-5

Chapter 4: JavaScript — An Object-Based Language

The cinema booking system you have created is very basic to say the least! However, it gives you an idea of how JavaScript classes can be used to help make code more maintainable and how they can be used to model real-world problems and situations.

Summary

In this chapter you've taken a look at the concept of objects and seen how vital they are to an understanding of JavaScript, which represents virtually everything with objects. You also looked at some of the various native objects that the JavaScript language provides to add to its functionality.

You saw that:

- ❑ JavaScript is object-based — it represents things, such as strings, dates, and arrays, using the concept of objects.
- ❑ Objects have properties and methods. For example, an `Array` object has the `length` property and the `sort()` method.
- ❑ To create a new object, you simply write `new ObjectType()`. You can choose to initialize an object when you create it.
- ❑ To set an object's property's value or get that value, you simply write `ObjectName.ObjectProperty`.
- ❑ Calling the methods of an object is similar to calling functions. Parameters may be passed, and return values may be passed back. Accessing the methods of an object is identical to accessing a property, except that you must remember to add parentheses at the end, even when there are no parameters. For example, you would write `ObjectName.ObjectMethod()`.
- ❑ The `String` object provides lots of handy functionality for text and gives you ways of finding out how long the text is, searching for text inside the string, and selecting parts of the text.
- ❑ The `Math` object is created automatically and provides a number of mathematical properties and methods. For example, to obtain a random number between 0 and 1, you use the method `Math.random()`.
- ❑ The `Array` object provides ways of manipulating arrays. Some of the things you can do are find the length of an array, sort its elements, and join two arrays together.
- ❑ The `Date` object provides a way of storing, calculating with, and later accessing dates and times.
- ❑ JavaScript enables you to create your own type of objects using classes. These can be used to model real-world situations and for making code easier to create and more maintainable, though they do require extra effort at the start.

In the next chapter, you'll turn your attention to the web browser itself and, particularly, the various objects that it makes available for your JavaScript programming. You'll see that the use of browser objects is key to creating powerful web pages.

Exercise Questions

Suggested solutions to these questions can be found in Appendix A.

Question 1

Using the `Date` object, calculate the date 12 months from now and write this into a web page.

Question 2

Obtain a list of names from the user, storing each name entered in an array. Keep getting another name until the user enters nothing. Sort the names in ascending order and then write them out to the page, with each name on its own line.

Question 3

You saw earlier in the chapter when looking at the `pow()` method how you could use it inventively to fix a number to a certain number of decimal places. However, there is a flaw in the function you created. A proper `fix()` function should return 2.1 fixed to three decimal places as

```
2.100
```

However, your `fix()` function instead returns it as

```
2.1
```

Change the `fix()` function so that the additional zeros are added where necessary.

5

Programming the Browser

Over the past three chapters, you've examined the core JavaScript language. You've seen how to work with variables and data, perform operations on those data, make decisions in your code, loop repeatedly over the same section of code, and even how to write your own functions. In the preceding chapter you moved on to learn how JavaScript is an object-based language, and you saw how to work with the native JavaScript objects. However, you are not interested only in the language itself—you want to find out how to write script for the web browser. Using this ability, you can start to create more impressive web pages.

Not only is JavaScript object-based, but the browser is also made up of objects. When JavaScript is running in the browser, you can access the browser's objects in exactly the same way that you used JavaScript's native objects in the last chapter. But what kinds of objects does the browser provide?

The browser makes available a remarkable number of objects. For example, there is a `window` object corresponding to the window of the browser. You have already been using two methods of this object, namely the `alert()` and `prompt()` methods. For simplicity, we previously referred to these as functions, but they are in fact methods of the browser's `window` object.

Another object made available by the browser is the page itself, represented by the `document` object. Again, you have already used methods and properties of this object. Recall from Chapter 1 that you used the `document` object's `backgroundColor` property to change the background color of the page. You have also been using the `write()` method of the `document` object to write information to the page.

A variety of other objects exist, representing a lot of the HTML that you write in the page. For example, there is an `img` object for each `` tag that you use to insert an image into your document.

The collection of objects that the browser makes available to you for use with JavaScript is generally called the *Browser Object Model (BOM)*.

You will often see this termed the Document Object Model (DOM). However, throughout this book, we'll use the term DOM to refer to the W3C's standard Document Object Model, which is discussed in Chapter 13.

Chapter 5: Programming the Browser

All this added functionality of JavaScript comes with a potential downside. Which collections of objects are made available to you is highly dependent on the brand and version of the browser that you are using. Some objects are made available in some browsers and not in others, whereas other objects have different properties and methods in different browsers. The good news is that browsers are following the W3C's guidelines much more than they used to. This means that if you follow the W3C guidelines, your code is more likely to work with different browsers. Note, however, some browser writers have interpreted the W3C standards in a different way. You will see much more about the differences between the BOMs of IE and Firefox browsers in Chapter 12.

However, in this chapter you will concentrate on the core functionality of the BOM, the objects that are common to all browsers. You can achieve a lot in JavaScript by just sticking to such objects. You can find more information on them online at http://www.w3schools.com/dhtml/dhtml_domreference.asp and http://msdn.microsoft.com/library/default.asp?url=/workshop/author/dhtml/dhtml_node_entry.asp.

Introduction to the Browser Objects

In this section, we introduce the objects of the BOM that are common to all browsers.

In Chapter 4, you saw that JavaScript has a number of native objects that you have access to and can make use of. Most of the objects are those that you need to create yourself, such as the `String` and `Date` objects. Others, such as the `Math` object, exist without you needing to create them and are ready for use immediately when the page starts loading.

When JavaScript is running in a web page, it has access to a large number of other objects made available by the web browser. Rather like the `Math` object, these are created for you rather than your needing to create them explicitly. As mentioned, the objects, their methods, properties, and events are all mapped out in the BOM.

The BOM is very large and potentially overwhelming at first. However, you'll find that initially you won't be using more than 10 percent of the available objects, methods, and properties in the BOM. You'll start in this chapter by looking at the more commonly used parts of the BOM, shown in Figure 5-1. These parts of the BOM are, to a certain extent, common across all browsers. Later chapters will build on this so that by the end of the book you'll be able to really make the BOM work for you.

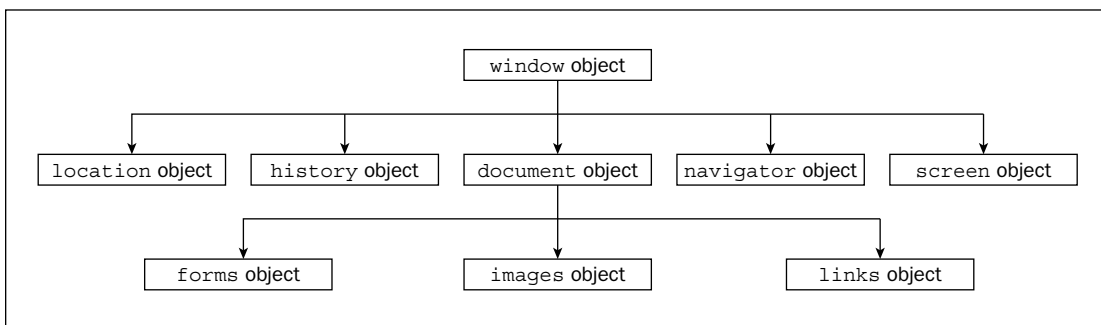


Figure 5-1

The BOM has a hierarchy. At the very top of this hierarchy is the `window` object. You can think of this as representing the frame of the browser and everything associated with it, such as the scrollbars, navigator bar icons, and so on.

Contained inside our window frame is the page. The page is represented in the BOM by the `document` object. You can see these two objects represented in Figure 5-2.

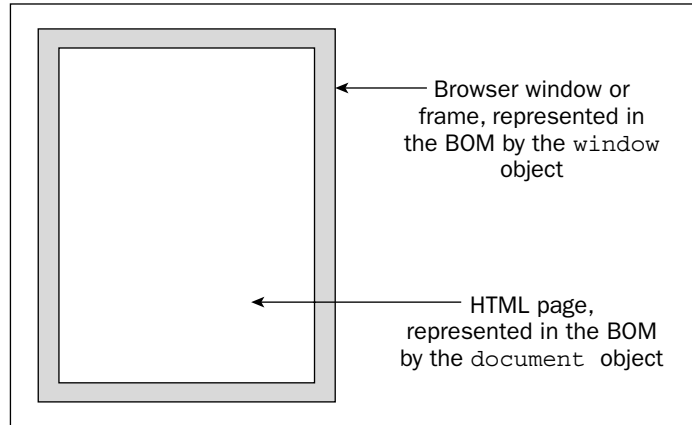


Figure 5-2

Now let's look at each of these objects in more detail.

The window Object — Our Window onto the Page

The `window` object represents the browser's frame or window, in which your web page is contained. To some extent, it also represents the browser itself and includes a number of properties that are there simply because they don't fit anywhere else. For example, via the properties of the `window` object, you can find out what browser is running, the pages the user has visited, the size of the browser window, the size of the user's screen, and much more. You can also use the `window` object to access and change the text in the browser's status bar, change the page that is loaded, and even open new windows.

The `window` object is a *global object*, which means you don't need to use its name to access its properties and methods. In fact, the global functions and variables (the ones accessible to script anywhere in a page) are all created as properties of the global object. For example, the `alert()` function you have been using since the beginning of the book is, in fact, the `alert()` method of the `window` object. Although you have been using this simply as

```
alert("Hello!");
```

you could write

```
window.alert("Hello!");
```

However, since the `window` object is the global object, it is perfectly correct to use the first version.

Chapter 5: Programming the Browser

Some of the properties of the `window` object are themselves objects. Those common to both IE and NN include the `document`, `navigator`, `history`, `screen`, and `location` objects. The `document` object represents your page, the `history` object contains the history of pages visited by the user, the `navigator` object holds information about the browser, the `screen` object contains information about the display capabilities of the client, and the `location` object contains details on the current page's location. You'll look at these important objects individually later in the chapter.

Using the window Object

Let's start with a nice, simple example in which you change the default text shown in the browser's status bar. The status bar (usually in the bottom left of the browser window) is usually used by the browser to show the status of any document loading into the browser. For example, on IE and Firefox, after a document has loaded, you'll normally see `Done` in the status bar. Let's change that so it says `Hello and Welcome`.

To change the default message in the window's status bar, you need to use the `window` object's `defaultStatus` property. To do this you can write the following:

```
window.defaultStatus = "Hello and Welcome";
```

Or, because the `window` is the global object, you can just write this:

```
defaultStatus = "Hello and Welcome";
```

Either way works, and both are valid; however, writing `window` in front makes it clear exactly where the `defaultStatus` property came from. Otherwise it might appear that `defaultStatus` is a variable name. This is particularly true for less common properties and methods, such as `defaultStatus`. You'll find yourself becoming so familiar with more common ones, such as `document` and `alert()`, that you don't need to put `window` in front to remind you of their context.

Let's put the code in a page.

```
<html>
<head>
<script language="JavaScript" type="text/javascript">
window.defaultStatus = "Hello and Welcome";
</script>
</head>
</html>
```

Save the page as `ch5_examp1.htm` and load it into your browser. You should see the specified message in the status bar.

At this point, it's worth highlighting the point that within a web page you shouldn't use names for your functions or variables that conflict with names of BOM objects or their properties and methods. If you do, you may not get an error, but instead get unexpected results. For example, in the following code you declare a variable named `defaultStatus`. You then try to set the `defaultStatus` property of the `window` object to `Welcome to my website`. However, this won't change the default message in the status bar; instead the value in the `defaultStatus` variable will be changed.

```
var defaultStatus;  
defaultStatus = "Welcome to my website";
```

In this situation you need to use a different variable name.

As with all the BOM objects, you can look at lots of properties and methods for the `window` object. However, in this chapter you'll concentrate on the `history`, `location`, `navigator`, `screen`, and `document` properties. All five of these properties contain objects (the `history`, `location`, `navigator`, `screen`, and `document` objects), each with its own properties and methods. In the next few pages, you'll look at each of these objects in turn and find out how they can help you make full use of the BOM.

The *history* Object

The `history` object keeps track of each page that the user visits. This list of pages is commonly called the *history stack* for the browser. It enables the user to click the browser's Back and Forward buttons to revisit pages. You have access to this object via the `window` object's `history` property.

Like the native JavaScript `Array` object, the `history` object has a `length` property. You can use this to find out how many pages are in the history stack.

As you might expect, the `history` object has the `back()` and `forward()` methods. When they are called, the location of the page currently loaded in the browser is changed to the previous or next page that the user has visited.

The `history` object also has the `go()` method. This takes one parameter that specifies how far forward or backward in the history stack you want to go. For example, if you wanted to return the user to the page before the previous page, you'd write this:

```
history.go(-2);
```

To go forward three pages, you'd write this:

```
history.go(3);
```

Note that `go(-1)` and `back()` are equivalent, as are `go(1)` and `forward()`.

The *location* Object

The `location` object contains lots of potentially useful information about the current page's location. Not only does it contain the URL (Uniform Resource Locator) for the page, but also the server hosting the page, the port number of the server connection, and the protocol used. This information is made available through the `location` object's `href`, `hostname`, `port`, and `protocol` properties. However, many of these values are only really relevant when you are loading the page from a server and not, as you are doing in the present examples, loading the page directly from a local hard drive.

In addition to retrieving the current page's location, you can also use the methods of the `location` object to change the location and refresh the current page.

Chapter 5: Programming the Browser

You can navigate to another page in two ways. You can either set the `location` object's `href` property to point to another page, or you can use the `location` object's `replace()` method. The effect of the two is the same; the page changes location. However, they differ in that the `replace()` method removes the current page from the history stack and replaces it with the new page you are moving to, whereas using the `href` property simply adds the new page to the top of the history stack. This means that if the `replace()` method has been used and the user clicks the Back button in the browser, the user can't go back to the original page loaded. If the `href` property has been used, the user can use the Back button as normal.

For example, to replace the current page with a new page called `myPage.htm`, you'd use the `replace()` method and write the following:

```
window.location.replace("myPage.htm");
```

This will load `myPage.htm` and replace any occurrence of the current page in the history stack with `myPage.htm`.

To load the same page and to add it to the history of pages navigated to, you use the `href` property:

```
window.location.href = "myPage.htm";
```

and the page currently loaded is added to the history. In both of the preceding cases, `window` is in front of the expression, but as the `window` object is global throughout the page, you could have written the following:

```
location.replace("myPage.htm");
```

or

```
location.href = "myPage.htm";
```

The navigator Object

The `navigator` object is another object that is a property of the `window` object and is available in both IE and Firefox browsers. Its name is more historical than descriptive. Perhaps a better name would be the "browser object," because the `navigator` object contains lots of information about the browser and the operating system in which it's running.

Probably the most common use of the `navigator` object is for handling browser differences. Using its properties, you can find out which browser, version, and operating system the user has. You can then act on that information and make sure the user is directed to pages that will work with his browser. The last section in this chapter is dedicated to this important subject, so we will not discuss it further here.

The screen Object

The `screen` object property of the `window` object contains a lot of information about the display capabilities of the client machine. Its properties include the `height` and `width` properties, which indicate the vertical and horizontal range of the screen, respectively, in pixels.

Another property of the `screen` object, which you will be using in an example later, is the `colorDepth` property. This tells you the number of bits used for colors on the client's screen.

The document Object — The Page Itself

Along with the `window` object, the `document` object is probably one of the most important and commonly used objects in the BOM. Via this object you can gain access to the properties and methods of some of the objects defined by HTML tags inside your page.

Unfortunately, it's here that the BOMs of different browsers can differ greatly. In this chapter you will concentrate on those properties and methods of the `document` object that are common to all browsers. More advanced manipulation of the browser document object will appear in Chapters 12 and 13.

The `document` object has a number of properties associated with it, which are also arrays. The main ones are the `forms`, `images`, and `links` arrays. IE supports a number of other array properties, such as the `all` array property, which is an array of all the tags represented by objects in the page. However, you'll be concentrating on using objects that have cross-browser support, so that you are not limiting your web pages to just one browser.

You'll be looking at the `images` and `links` arrays shortly. A third array, the `forms[]` array, will be one of the topics of the next chapter when you look at forms in web browsers. First, though, you'll look at a nice, simple example of how to use the `document` object's methods and properties.

Using the document Object

You've already come across some of the `document` object's properties and methods, for example the `write()` method and the `bgColor` property.

Try It Out

Setting Colors According to the User's Screen Color Depth

In this example you set the background color of the page according to how many colors the user's screen supports. This is termed *screen color depth*. If the user has a display that supports just two colors (black and white), there's no point in you setting the background color to bright red. You accommodate different depths by using JavaScript to set a color the user can actually see.

```
<html>
<body>
<script language="JavaScript" type="text/javascript">
switch (window.screen.colorDepth)
{
    case 1:
    case 4:
        document.bgColor = "white";
        break;
    case 8:
    case 15:
    case 16:
        document.bgColor = "blue";
        break;
    case 24:
    case 32:
        document.bgColor = "skyblue";
        break;
    default:
        document.bgColor = "white";
}
```

```
}
    document.write("Your screen supports " + window.screen.colorDepth +
        "bit color");
</script>
</body>
</html>
```

Save the page as `ch5_examp2.htm`. When you load it into your browser, the background color of the page will be determined by your current screen color depth. Also, a message in the page will tell you what the color depth currently is.

You can test that the code is working properly by changing the colors supported by your screen. On Windows, you can do this by right-clicking on the desktop and choosing the Properties option. Under the Settings tab, there is a section called “Color quality” in which you can change the number of colors supported. By refreshing the browser, you can see what difference this makes to the color of the page.

On Netscape and Firefox browsers, it’s necessary to shut down and restart the browser to observe any effect.

How It Works

As you saw earlier, the `window` object has the `screen` object property. One of the properties of this object is the `colorDepth` property, which returns a value of 1, 4, 8, 15, 16, 24, or 32. This represents the number of bits assigned to each pixel on your screen. (A pixel is just one of the many dots that your screen is made up of.) To work out how many colors you have, you just calculate the value of 2 to the power of the `colorDepth` property. For example, a `colorDepth` of 1 means that there are two colors available, a `colorDepth` of 8 means that there are 256 colors available, and so on. Currently, most people have a screen color depth of at least 8, but usually of 16 or 24, with 32 increasingly common.

The first task of your script block is to set the color of the background of the page based on the number of colors the user can actually see. You do this in a big `switch` statement. The condition that is checked for in the `switch` statement is the value of `window.screen.colorDepth`.

```
switch (window.screen.colorDepth)
```

You don’t need to set a different color for each `colorDepth` possible, because many of them are similar when it comes to general web use. Instead, you set the same background color for different, but similar, `colorDepth` values. For a `colorDepth` of 1 or 4, you set the background to white. You do this by declaring the `case 1:` statement, but you don’t give it any code. If the `colorDepth` matches this `case` statement, it will fall through to the `case 4:` statement below, where you do set the background color to white. You then call a `break` statement, so that the case matching will not fall any further through the `switch` statement.

```
{
    case 1:
    case 4:
        document.bgColor = "white";
        break;
```

You do the same with `colorDepth` values of 8, 15, and 16, setting the background color to blue as follows:

```
case 8:
case 15:
case 16:
    document.bgColor = "blue";
    break;
```

Finally, you do the same for `colorDepth` values of 24 and 32, setting the background color to sky blue.

```
case 24:
case 32:
    document.bgColor = "skyblue";
    break;
```

You end the `switch` statement with a default case, just in case the other case statements did not match. In this default case, you again set the background color to white.

```
default:
    document.bgColor = "white";
}
```

In the next bit of script, you use the `document` object's `write()` method, something you've been using in these examples for a while now. You use it to write to the document—that is, the page—the number of bits the color depth is currently set at, as follows:

```
document.write("Your screen supports " + window.screen.colorDepth +
               "bit color")
```

You've already been using the `document` object in the examples throughout the book so far. You used its `bgColor` property in Chapter 1 to change the background color of the page, and you've also made good use of its `write()` method in the examples to write HTML and text out to the page.

Now let's look at some of the slightly more complex properties of the `document` object. These properties have in common the fact that they all contain arrays. The first one you look at is an array containing an object for each image in the page.

The images Array

As you know, you can insert an image into an HTML page using the following tag:

```

```

The browser makes this image available for you to script in JavaScript by creating an `img` object for it with the name `myImage`. In fact, each image on your page has an `img` object created for it.

Each of the `img` objects in a page is stored in the `images[]` array. This array is a property of the `document` object. The first image on the page is found in the element `document.images[0]`, the second in `document.images[1]`, and so on.

Chapter 5: Programming the Browser

If you want to, you can assign a variable to reference an `img` object in the `images[]` array. It can make code easier to read. For example, if you write

```
var myImage2 = document.images[1];
```

the `myImage2` variable will contain a reference to the `img` object inside the `images[]` array at index position 1. Now you can write `myImage2` instead of `document.images[1]` in your code, with exactly the same effect.

You can also access `img` objects in the `images` array by name. For example, the `img` object created by the `` tag, which has the name `myImage`, can be accessed in the `document` object's `images` array property like this:

```
document.images["myImage"]
```

Because the `document.images` property is an array, it has the properties of the native JavaScript `Array` object, such as the `length` property. For example, if you want to know how many images there are on the page, the code `document.images.length` will tell you.

Try It Out Image Selection

The `img` object itself has a number of useful properties. The most important of these is its `src` property. By changing this you can change the image that's loaded. The next example demonstrates this.

```
<html>
<body>
<img name=img1 src="" border=0 width=200 height=150>
<script language="JavaScript" type="text/javascript">
  var myImages = new Array("usa.gif","canada.gif","jamaica.gif","mexico.gif");
  var imgIndex = prompt("Enter a number from 0 to 3","");
  document.images["img1"].src = myImages[imgIndex];
</script>
</body>
</html>
```

Save this as `ch5_examp3.htm`. You will also need four image files, called `usa.gif`, `canada.gif`, `jamaica.gif`, and `mexico.gif`. You can create these yourself or obtain the ones provided with the code download for the book.

When this page is loaded into the browser, a prompt box asks you to enter a number from 0 to 3. A different image will be displayed depending on the number you enter.

How It Works

At the top of the page you have your HTML `` tag. Notice that the `src` attribute is left empty and is given the name value `img1`.

```
<img name=img1 src="" border=0 width=200 height=150>
```

Next you come to the script block where the image to be displayed is decided. On the first line you define an array containing a list of image sources. In this example, the images are in the same directory

as the HTML file, so a path is not specified. If yours are not, make sure you enter the full path (for example, `C:\myImages\mexico.gif`).

Then you ask the user for a number from 0 to 3, which will be used as the array index to access the image source in the `myImages` array.

```
var imgIndex = prompt("Enter a number from 0 to 3","");
```

Finally you set the `src` property of the `img` object to the source text inside the `myImages` array element with the index number provided by the user.

```
document.images["img1"].src = myImages[imgIndex];
```

Don't forget that when you write `document.images["img1"]`, you are accessing the `img` object stored in the `images` array. You've used the image's name, as defined in the name attribute of the `` tag, but you could have used `document.images[0]`. It's an index position of 0, because it's the first (and only) image on this page.

The links Array

For each hyperlink tag `<A>` defined with an `href` attribute, the browser creates an `A` object. The most important property of the `A` object is the `href` property, corresponding to the `href` attribute of the tag. Using this, you can find out where the link points to, and you can change this, even after the page has loaded.

The collection of all `A` objects in a page is contained within the `links[]` array, much as the `img` objects are contained in the `images[]` array, as you saw earlier.

Connecting Code to Web Page Events

Chapter 4 introduces objects that are defined by their methods and properties. However, objects also have events associated with them. We did not mention this before, because native JavaScript objects do not have these events, but the objects of the BOM do.

So what are these events?

Events occur when something in particular happens. For example, the user clicking on the page, clicking on a hyperlink, or moving his mouse pointer over some text all cause events to occur. Another example, which is used quite frequently, is the load event for the page.

Why are you interested in events?

Take as an example the situation in which you want to make a menu pop up when the user clicks anywhere in your web page. Assuming that you can write a function that will make the pop-up menu appear, how do you know *when* to make it appear, or in other words, *when* to call the function? You somehow need to intercept the event of the user clicking in the document, and make sure your function is called when that event occurs.

Chapter 5: Programming the Browser

To do this, you need to use something called an *event handler*. You associate this with the code that you want to execute when the event occurs. This provides you with a way of intercepting events and making your code execute when they have occurred. You will find that adding an event handler to your code is often known as “connecting your code to the event.” It’s a bit like setting an alarm clock—you set the clock to make a ringing noise when a certain event happens. With alarm clocks, the event is when a certain time is reached.

Event handlers are made up of the word `on` and the event that they will handle. For example, the click event has the `onclick` event handler, and the load event has the `onload` event handler.

A number of ways exist to connect your code to an event using event handlers. In this chapter you’ll look at a two of the easiest ways of adding events, ways that have been around since Netscape 2 and so are supported even by older browsers, as well as by current ones. In Chapter 12 you’re going to look at newer and standards-friendly ways of adding events.

Event Handlers as Attributes

The first and most common method is to add the event handler’s name and the code you want to execute to the HTML tag’s attributes.

Let’s create a simple HTML page with a single hyperlink, given by the element `<A>`. Associated to this element is the `A` object. One of the events the `A` object has is the click event. The click event fires, not surprisingly, when the user clicks the hyperlink.

```
<html>
<body>
<A href="somepage.htm" name="linkSomePage">
  Click Me
</A>
</body>
</html>
```

As it stands, this page does nothing a normal hyperlink doesn’t do. You click it, and it navigates the window to another page, called `somepage.htm`, which would need to be created. There’s been no event handler added to the link—yet!

As mentioned earlier, one very common and easy way of connecting the event to your code is to add it directly to the tag of the object whose event you are capturing. In this case, it’s the click event of the `A` object, as defined by the `<A>` tag. On clicking the link, you want to capture the event and connect it to your code. You need to add the event handler, in this case `onclick`, as an attribute to your `<A>` tag. You set the value of the attribute to the code you want to have executed when the event occurs.

Let’s rewrite the `<A>` tag to do this as follows:

```
<A href="somepage.htm" name="linkSomePage" onclick="alert('You Clicked?')">
  Click Me
</A>
```

You can see that you have added `onclick="alert('You Clicked?')"` to the definition of the `<A>` tag. Now, when the link is clicked, you see an alert box. After this, the hyperlink does its usual stuff and takes you to the page defined in the `href` attribute.

This is fine if you have only one line of code to connect to the event handler, but what if you want a number of lines to execute when the link is clicked?

Well, all you need to do is define the function you want to execute and call it in the `onclick` code. Let's do that now.

```
<html>
<body>
<script language="JavaScript">
function linkSomePage_onclick()
{
    alert('You Clicked?');
    return true;
}
</script>
<A href="somepage.htm" name="linkSomePage"
    onclick="return linkSomePage_onclick()">
    Click Me
</A>
</body>
</html>
```

Within the script block you have created a function, just a standard function, and given it a descriptive name to help you when reading the code. Here we're using `ObjectName_event()` as the function name. That way you can instantly see what object on the page this relates to and which event is being connected to. So, in the preceding example, the function is called `linkSomePage_onclick()`, because you are referring to the `onclick` event handler for the `A` object with name `linkSomePage`. Note that this naming convention is simply something created by the author; it's not compulsory, and you can use whatever convention you prefer as long as you are consistent.

The `onclick` attribute is now connected to some code that calls the function `linkSomePage_onclick()`. Therefore, when the user clicks the hyperlink, this function will be executed.

You'll also see that the function returns a value, `true` in this case. Also, where you define your `onclick` attribute, you return the return value of the function by using the `return` statement before the function name. Why do this?

The value returned by `onclick="return linkSomePage_onclick()"` is used by JavaScript to decide whether the normal action of the link — that is, going to a new page — should occur. If you return `true`, the action continues, and you go to `somepage.htm`. If you return `false`, the normal chain of events (that is, going to `somepage.htm`) does not happen. You say that the action associated with the event is canceled. Try changing the function to this:

```
function linkSomePage_onclick()
{
    alert('This link is going nowhere');
    return false;
}
```

Now you'll find that you just get a message, and no attempt is made to go to `somepage.htm`.

Chapter 5: Programming the Browser

Not all objects and their events make use of the return value, so sometimes it's redundant. Also, it's not always the case that returning `false` cancels the action. For reasons of browser history rather than logic, it's sometimes `true` that cancels the action. Generally speaking, it's best to return `true` and deal with the exceptions as you find them.

Some events are not directly linked with the user's actions as such. For example, the `window` object has the `load` event, which fires when a page is loaded, and the `unload` event, which fires when the page is unloaded (that is, when the user either closes the browser or moves to another page).

Event handlers for the `window` object actually go inside the `<body>` tag. For example, to add an event handler for the `load` and `unload` events, you'd write the following:

```
<body language="JavaScript" onload="myOnLoadfunction()"
onunload="myOnUnloadFunction()">
```

Notice that you have specified the `language` attribute of the `<body>` tag as `JavaScript`. This is because the `<body>` tag is not contained within a JavaScript script block defined with the `<script>` tag. As usual, since JavaScript is the default scripting language, this can be left off.

Event Handlers as Properties

Now let's look at the second way to connect to events.

With this method, you first need to define the function that will be executed when the event occurs. Then you need to set that object's event handler property to the function you defined.

This is illustrated in the following example:

```
<html>
<body>
<script language="JavaScript" type="text/javascript">
function linkSomePage_onclick()
{
    alert('This link is going nowhere');
    return false;
}
</script>
<A href="somepage.htm" name="linkSomePage">
    Click Me
</A>
<script language="JavaScript" type="text/javascript">
    window.document.links[0].onclick = linkSomePage_onclick;
</script>
</body>
</html>
```

Save this as `ch5_examp4.htm`.

You define the function `linkSomePage_onclick()`, much as you did before. As before, you can return a value indicating whether you want the normal action of that object to happen.

Next you have the `<A>` tag, whose object's event you are connecting to. You'll notice there is no mention of the event handler or the function within the attributes of the tag.

The connection is made between the object's event and our function on the final lines of script, as shown in the following code:

```
<script language="JavaScript" type="text/javascript">
    document.links[0].onclick = linkSomePage_onclick;
</script>
```

As you saw before, `document.links[0]` returns the `A` object corresponding to the first link in your web page, which is your `linkSomePage` hyperlink. You set this object's `onclick` property to reference your function—this makes the connection between the object's event handler and your function. Note that no parentheses are added after the function name. Now whenever you click the link, your function gets executed.

The first method of connecting code to events is easier, so why would you ever want to use the second?

Perhaps the most common situation in which you would want to do this is one in which you want to capture an event for which there is no HTML tag to write your event handler as an attribute. It is also useful if you want the code attached to an event handler to be changed dynamically.

Try It Out Displaying a Random Image when the Page Loads

Let's look at another example in which you connect to a hyperlink's click event to randomly change the image loaded in a page.

```
<html>
<head>
<script language="JavaScript" type="text/javascript">
var myImages = new Array("usa.gif", "canada.gif", "jamaica.gif", "mexico.gif");

function changeImg(imgNumber)
{
    var imgClicked = document.images[imgNumber];
    var newImgNumber = Math.round(Math.random() * 3);
    while (imgClicked.src.indexOf(myImages[newImgNumber]) != -1)
    {
        newImgNumber = Math.round(Math.random() * 3);
    }
    imgClicked.src = myImages[newImgNumber];
    return false;
}

</script>
</head>
<body>
    
    
</body>
</html>
```

Chapter 5: Programming the Browser

Save the page as `ch5_exam5.htm`. Again, you will need four image files for the example, which you can create or retrieve from the code download available with this book.

Load the page into your browser. You should see a page like that shown in Figure 5-3.

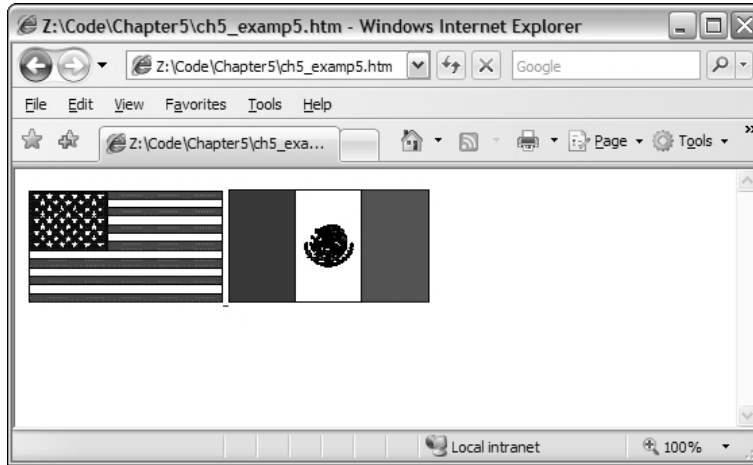


Figure 5-3

If you click an image, you'll see it change to a different image, which is selected randomly.

How It Works

The first line in the script block at the top of the page defines a variable with page-level scope. This is an array that contains your list of image sources.

```
var myImages = new Array("usa.gif", "canada.gif", "jamaica.gif", "mexico.gif");
```

Next you have the `changeImg()` function, which will be connected to the `onclick` event handler of an `` tag surrounding each of your images. You are using the same function for both images' `onclick` event handlers and indeed can connect one function to as many event handlers as you like. You pass this function one parameter—the index in the `images` array of the `img` object related to that click event—so that you know which image you need to act on.

In the first line of the function, you use the passed parameter to declare a new variable that points to the `img` object in the `images[]` array corresponding to the image that was clicked, as follows:

```
function changeImg(imgNumber)
{
    var imgClicked = document.images[imgNumber];
```

Following this, you set the `newImgNumber` variable to a random integer between 0 and 3. The `Math.random()` method provides a random number between 0 and 1, and you multiply that by three to get a number between 0 and 3. This number is converted to an integer (0, 1, 2, or 3) by means of

`Math.round()`. This integer will provide the index for the image `src` that you will select from the `myImages` array.

```
var newImgNumber = Math.round(Math.random() * 3);
```

The next lines are a `while` loop, the purpose of which is to ensure that you don't select the same image as the current one. If the string contained in `myImages[newImgNumber]` is found inside the `src` property of the current image, you know it's the same and that you need to get another random number. You keep looping until you get a new image, at which point `myImages[newImgNumber]` will not be found in the existing `src` and `-1` will be returned by the `indexOf()` method, breaking out of the loop.

```
while (imgClicked.src.indexOf(myImages[newImgNumber]) != -1)
{
    newImgNumber = Math.round(Math.random() * 3);
}
```

Finally, you set the `src` property of the `img` object to the new value contained in your `myImages` array. You return `false` to stop the link from trying to navigate to another page; remember that the HTML link is only there to provide a means of capturing an `onclick` event handler.

```
imgClicked.src = myImages[newImgNumber];
return false;
}
```

Next you connect the `onclick` event of the first `` tag to the `changeImg()` function:

```

```

And now to the second `` tag:

```

```

Passing `1` in the `changeImg()` function lets the function know that image `1`, the second image in the page, needs to be changed.

Browser Version Checking Examples

Many browsers, versions of those browsers, and operating systems are out there on the Internet, each with its own version of the BOM and its own particular quirks. It's therefore important that you make sure your pages will work correctly on all browsers, or at least *degrade gracefully*, such as by displaying a message suggesting that the user upgrade her browser.

Although you can go a long way with cross-browser-compatible code, there may come a time when you want to add extra features that only one browser supports. The solution is to write script that checks the browser name, version, and, if necessary, operating system, and executes script that is compatible with these variants.

You can check for browser details in two main ways. The first is to see if the object and property you use in your code are actually available in the user's browser. Let's say for example that our code relies on the

Chapter 5: Programming the Browser

document object's `all` property, which is available to browsers like IE version 4+, but not to any Firefox or Netscape browsers. If you write

```
If (document.all)
{
    // our code using the document.all property
}
```

the `if` statement's condition will evaluate to `true` if the property returns a valid value; if the property is not supported, its value will be `undefined`, and the `if` statement will evaluate to `false`. To check whether a particular method is supported, you can do this:

```
if (document.getElementById)
{
    // code
}
else
{
    // Alternative code
}
```

You've "tested" the existence of the method as you did with properties. Just remember not to include the opening or closing brackets after the method even if it normally has a number of parameters. The `getElementById` method, for example, has one parameter.

The next example shows how to use object checking to ensure that you execute the right code for the right browser; this technique is not foolproof but can be very useful.

Try It Out Checking for Supported Browser Properties

```
<html>
<body>

<script language="JavaScript" type="text/javascript">
var browser = "Unknown";
var version = "0";

// NN4+
if (document.layers)
{
    browser = "NN";
    version = "4.0";

    if (navigator.securityPolicy)
    {
        version = "4.7+";
    }
}
else if (document.all)
{
    browser = "IE"
    version = "4"
```

```

}

// IE5+
if (window.clipboardData)
{
    browser = "IE"
    version = "5+"
}
// Firefox/NN6+
else if (window.sidebar)
{
    browser = "Firefox";
    version = "1+";
}
document.write(browser + " " + version);

</script>

</body>
</html>

```

Save this example as `ch5_examp6.htm`.

How It Works

The page looks at which BOM and JavaScript properties the browser supports and, based on that, makes a rough guess as to the browser type. So on the first lines you check to see if the browser's `document` object has the `layers` property.

```

if (document.layers)
{
    browser = "NN";
    version = "4.0";

    if (navigator.securityPolicy)
    {
        version = "4.7+";
    }
}

```

Because only NN 4 supports the `layers` property, you can assume this is an NN 4 browser or at least a browser that supports the NN 4 `layers` feature. The inner `if` statement looks to see if the `navigator` object supports the `securityPolicy` property; this property is supported by only Netscape Navigator version 4 and in particular only by sub-version 4.7 and later. However, the slight danger here is that a rare browser not from Netscape or Microsoft might support the `securityPolicy` property but not other features of NN 4.7+ browsers. It's wise to test with as many varieties of browser as you expect to have visit the web site and see what happens. If one fails, you can see which property it fails on and write code to check for it.

Next you have a test for the `document` object's `all` property, a property supported by IE 4+ and Opera 7+.

Chapter 5: Programming the Browser

```
else if (document.all)
{
    browser = "IE"
    version = "4"
}
```

To see if the browser is an IE 5+ browser, check the window object's `clipboardData` property, which is currently supported only by IE 5+.

```
// IE5+
if (window.clipboardData)
{
    browser = "IE"
    version = "5+"
}
```

The final bit of checking is for the `sidebar` property supported by Firefox, which deals with the sidebar tool.

```
// Firefox/NN6+
else if (window.sidebar)
{
    browser = "Firefox";
    version = "1+";
}
```

On the last line you write out the results of the BOM property checking.

```
document.write(browser + " " + version);
```

Hopefully, this example demonstrates how to use object checking to see if a particular feature is supported by a browser. In the example, you haven't actually used the various features; it's simply a way of demonstrating how to check for browser-specific objects. When writing your own code, be sure to double-check whether a particular feature you're using is supported by all the browsers you expect to visit your web site. If some of the browsers you expect to visit don't support a particular feature, then test for the feature and write alternative code for the browsers that don't support it.

You'll be seeing much more advanced object checking in Chapter 12.

No Script at All

Sometimes people switch off JavaScript in their browsers, or use a browser that doesn't support JavaScript, though that's quite rare these days. To cover this situation, you can use the `<noscript>` tag. Any HTML inside the `<noscript>` tag will be displayed only to browsers that don't support JavaScript or on which JavaScript has been disabled:

```
<html>
<body>
<noscript>
This website requires JavaScript to be enabled.
</noscript>
</body>
<html>
```

Browser Checking Using the Navigator Object

The second method of checking browser details is using the `navigator` object property of the `window` object. In particular, you use the `appName` and `userAgent` properties of the `navigator` object. The main problem with this method is that a less common browser may well declare itself to be a particular version of Internet Explorer or Firefox but not actually support all the JavaScript or BOM objects, properties, or methods of that browser. Therefore this method of “browser sniffing” has fallen out of favor and is not the recommended way of checking for compatibility. It’s really a last resort when all other methods have failed, such as when two different browsers support the same object and property but implement them so that they work in two different ways. Object checking wouldn’t help you in that circumstance, so you’d have to fall back on using the `navigator` object.

The `appName` property returns the model of the browser, such as Microsoft Internet Explorer or Netscape.

The `userAgent` property returns a string containing various bits of information, such as the browser version, operating system, and browser model. However, the value returned by this property varies from browser to browser, so you have to be very, very careful when using it. For example, you can’t assume that it starts with the browser version. It does under NN, but under IE the browser version is embedded in the middle of the string.

Try It Out Checking for and Dealing with Different Browsers

In this example, you create a page that uses the aforementioned properties to discover the client’s browser and browser version. The page can then take action based upon the client’s specifications.

```
<html>
<head>
<script language="JavaScript" type="text/javascript">

function getBrowserName()
{
    var lsBrowser = navigator.userAgent;
    if (lsBrowser.indexOf("MSIE") >= 0)
    {
        lsBrowser = "MSIE";
    }
    else if (lsBrowser.indexOf("Netscape") >= 0)
    {
        lsBrowser = "Netscape";
    }
    else if (lsBrowser.indexOf("Firefox") >= 0)
    {
        lsBrowser = "Firefox";
    }
    else if (lsBrowser.indexOf("Safari") >= 0)
    {
        lsBrowser = "Safari";
    }
    else if (lsBrowser.indexOf("Opera") >= 0)
    {
        lsBrowser = "Opera";
    }
    else
```

```
{
    lsBrowser = "UNKNOWN";
}
return lsBrowser;
}

function getBrowserVersion()
{
    var findIndex;
    var browserVersion = 0;
    var browser = getBrowserName();

    browserVersion = navigator.userAgent;
    findIndex = browserVersion.indexOf(browser) + browser.length + 1;
    browserVersion = parseFloat(browserVersion.substring(findIndex, findIndex + 3));
    return browserVersion;
}

</script>
</head>
<body>
<script language="JavaScript" type="text/javascript">

var browserName = getBrowserName();
var browserVersion = getBrowserVersion();

if (browserName == "MSIE")
{
    if (browserVersion < 5.5)
    {
        document.write("Your version of Internet Explorer is too old");
    }
    else
    {
        document.write("Your version of Internet Explorer is fully supported");
    }
}
else if (browserName == 'Firefox')
{
    document.write("Firefox is fully supported");
}
else if (browserName == 'Netscape')
{
    if (browserVersion < 6)
    {
        document.write("Your version of Netscape is too old");
    }
    else
    {
        document.write("Your version of Netscape is fully supported");
    }
}
}
```



```
else
{
    document.write("<h2>Sorry this browser version is not supported</h2>");
}

</script>
<noscript>
    <h2>This website requires a browser supporting scripting</h2>
</noscript>
</body>
</html>
```

Save this script as `ch5_examp7.htm`.

If the browser is Firefox, Internet Explorer 5.5+, or Netscape Navigator 6+, a message appears telling the user that the browser is supported. If it's an earlier version of IE or Navigator, the user will see a message telling him his version of that browser is not supported.

If it's not one of those browsers, he'll see a message saying his browser is unsupported. This is not particularly friendly, so in practice you could have available a plain and simple version of the page without scripting — something with as much functionality as possible without JavaScript.

If the browser doesn't support JavaScript or the user has turned off support, then he'll see a message telling him the web site needs JavaScript to work.

How It Works

The script block in the head of the page defines two important functions. The `getBrowserName()` function finds out the name of the browser and the `getBrowserVersion()` function finds out the browser version.

The key to the browser checking code is the value returned by the `navigator.userAgent` property. Here are a few example user agent strings from current browsers:

1. Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322; .NET CLR 2.0.40607)
2. Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 2.0.40607)
3. Mozilla/5.0 (Windows; U; Windows NT 5.1; en-GB; rv:1.8.0.4) Gecko/20060508 Firefox/1.5.0.4
4. Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7.5) Gecko/20060127 Netscape/8.1
5. Opera/9.00 (Windows NT 5.0; U; en)

Here each line of the `userAgent` string has been numbered. Looking closely at each line it's not hard to guess which browser each agent string relates to. In order:

1. Microsoft Internet Explorer 6
2. Microsoft Internet Explorer 7 (beta 3)
3. Firefox 1.5
4. Netscape 8.1
5. Opera 9.0

Chapter 5: Programming the Browser

Using this information, let's start on the first function, `getBrowserName()`. First you get the name of the browser, as found in `navigator.userAgent`, and store it in the variable `lsBrowser`. This will also be used as the variable to store the return value for the function.

```
function getBrowserName()
{
    var lsBrowser = navigator.userAgent;
```

The string returned by this property tends to be quite long and does vary slightly sometimes. However, by checking for the existence of certain keywords, such as MSIE or Netscape, you can determine the browser name. Start with the following lines:

```
    if (lsBrowser.indexOf("MSIE") >= 0)
    {
        lsBrowser = "MSIE";
    }
```

These lines search the `lsBrowser` string for Microsoft. If the `indexOf` value of this substring is 0 or greater, you know you have found it, and so you set the return value to MSIE.

The following `else if` statement does the same, except that it is modified for Netscape.

```
    else if (lsBrowser.indexOf("Netscape") >= 0)
    {
        lsBrowser = "Netscape";
    }
```

This principle carries on for another three `if` statements, in which you also check for Firefox, Safari, and Opera. If you have a browser you want to check for, this is the place to add its `if` statement. Just view the string it returns in `navigator.userAgent` and look for its name or something that uniquely identifies it.

If none of the `if` statements match, you return UNKNOWN as the browser name.

```
    else
    {
        lsBrowser = "UNKNOWN";
    }
```

The value of `lsBrowser` is then returned to the calling code.

```
    return lsBrowser;
}
```

You now turn to the final function, `getBrowserVersion()`.

The browser version details often appear in the `userAgent` string right after the name of the browser. For these reasons, your first task in the function is to find out which browser you are dealing with. You declare and initialize the `browser` variable to the name of the browser, using the `getBrowserName()` function you just wrote.

```
function getBrowserVersion()
{
    var findIndex;
    var browserVersion = 0;
    var browser = getBrowserName();
```

If the browser is MSIE (Internet Explorer), you need to use the `userAgent` property again. Under IE, the `userAgent` property always contains MSIE followed by the browser version. So what you need to do is search for MSIE, then get the number following that.

You set `findIndex` to the character position of the browser name plus the length of the name, plus one. Doing this ensures you to get the character after the name and after the following space or / character that follows the name and is just before the version number. `browserVersion` is set to the floating-point value of that number, which you obtain using the `substring()` method. This selects the character starting at `findIndex`, your number, and whose end is one before `findIndex`, plus three. This ensures that you just select three characters for the version number.

```
browserVersion = navigator.userAgent;
findIndex = browserVersion.indexOf(browser) + browser.length + 1;
browserVersion = parseFloat(browserVersion.substring(findIndex,findIndex + 3));
```

If you look back to the `userAgent` strings, you see that IE7's is as follows:

```
Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 2.0.40607)
```

So `findIndex` will be set to the character index of the number 7 following the browser name. `browserVersion` will be set to three characters from and including the 7, giving the version number as 7.0.

At the end of the function you return `browserVersion` to the calling code, as shown here:

```
    return browserVersion;
}
```

You've seen the supporting functions, but how do you make use of them? Well, in the following code, which executes as the page is loaded, you obtain two bits of information—browser name and version—and use these to filter which browser the user is running.

```
var browserName = getBrowserName();
var browserVersion = getBrowserVersion();

if (browserName == "MSIE")
{
    if (browserVersion < 5.5)
    {
        document.write("Your version of Internet Explorer is too old");
    }
    else
    {
        document.write("Your version of Internet Explorer is fully supported");
    }
}
```

Chapter 5: Programming the Browser

The first of the `if` statements is shown in the preceding code and checks to see if the user has IE. If she does, it then checks to see if the version is one earlier, less, than version 5.5. If it is, she is told the IE she has is too old. If it is IE5.5+, she is told it's fully supported.

You do this again for Firefox and Netscape. The Firefox version isn't checked in this example, but you can check it if you want to:

```
else if (browserName == 'Firefox')
{
    document.write("Firefox is fully supported");
}
else if (browserName == 'Netscape')
{
    if (browserVersion < 6)
    {
        document.write("Your version of Netscape is too old");
    }
    else
    {
        document.write("Your version of Netscape is fully supported");
    }
}
else
{
    document.write("<h2>Sorry this browser version is not supported</h2>")
}
```

On the final part of the `if` statements is the `else` statement that covers all other browsers and tells the user the browser is not supported.

Finally, there are some `<noscript>` tags for early browsers and for users who have chosen to disable JavaScript. These will display a message informing the user that she needs a JavaScript-enabled browser.

```
<noscript>
  <h2>This website requires a browser supporting scripting</h2>
</noscript>
```

As mentioned earlier, although this script will work fine at the moment, it's possible that browsers will change their `userAgent` strings and you'll need to update the function to keep track of this. Also, some browsers pretend to be other browsers even if they don't function 100 percent the same, which can leave your code showing errors.

For these reasons, stick to the object checking method, which was detailed earlier in the chapter and will be covered more fully in Chapter 12.

Summary

You've covered a lot in this chapter, but now you have all the grounding you need to move on to more useful things, such as forms and user input, and later to more advanced areas of text and date manipulation.

- ❑ You turned your attention to the browser, the environment in which JavaScript exists. Just as JavaScript has native objects, so do web browsers. The objects within the web browser, and the hierarchy they are organized in, are described by something called the Browser Object Model (BOM). This is essentially a map of a browser's objects. Using it, you can navigate your way around each of the objects made available by the browser, together with their properties, methods, and events.
- ❑ The first of the main objects you looked at was the `window` object. This sits at the very top of the BOM's hierarchy. The `window` object contains a number of important sub-objects, including the `location` object, the `navigator` object, the `history` object, the `screen` object, and the `document` object.
- ❑ The `location` object contains information about the current page's location, such as its file name, the server hosting the page, and the protocol used. Each of these is a property of the `location` object. Some properties are read-only, but others, such as the `href` property, not only enable us to find the location of the page, but also can be changed so that we can navigate the page to a new location.
- ❑ The `history` object is a record of all the pages the user has visited since opening his or her browser. Sometimes pages are not noted (for example, when the `location` object's `replace()` method is used for navigation). You can move the browser forward and backward in the history stack and discover what pages the user has visited.
- ❑ The `navigator` object represents the browser itself and contains useful details of what type of browser, version, and operating system the user has. These details enable you to write pages dealing with various types of browsers, even where they may be incompatible.
- ❑ The `screen` object contains information about the display capabilities of the user's computer.
- ❑ The `document` object is one of the most important objects. It's an object representation of our page and contains all the elements, also represented by objects, within that page. The differences between Netscape and Microsoft browsers are particularly prominent here. If you want cross-browser-compatible pages, you will find you are quite limited as to which elements you can access.
- ❑ The `document` object contains three properties that are actually arrays. These are the `links[]`, `images[]`, and `forms[]` arrays. Each contains all the objects created by the `<A>`, ``, and `<form>` tags on the page, and it's our way of accessing those tags.
- ❑ The `images[]` array contains an `img` object for each `` element on the page. You found that even after the page has loaded, you can change the properties of images. For example, you can make the image change when clicked. The same principles for using the `images[]` array apply to the `links[]` array.
- ❑ You next saw that BOM objects have events as well as methods and properties. You handle these events in JavaScript by using event handlers, which you connect to code that you want to have executed when the event occurs. The events available for use depend on the object you are dealing with.

- ❑ Connecting a function that you have written to an event handler is simply a matter of adding an attribute to the element corresponding to the particular object you are interested in. The attribute has the name of the event handler you want to capture and the value of the function you want to connect to it.
- ❑ In some instances, such as for the `document` object, a second way of connecting event handlers to code is necessary. Setting the object's property with the name of the event handler to our function produces the same effect as if you did it using the event handler as an attribute.
- ❑ In some instances, returning values from event functions enables you to cancel the action associated with the event. For example, to stop a clicked link from navigating to a page, you return `false` from the event handler's code.
- ❑ Finally, you looked at how you can check what type of browser the user has so that you can make sure the user sees only those pages or parts of a page that his browser is compatible with. The `navigator` object provides you with the details you need, in particular the `appName` and `userAgent` properties. You can also check specific BOM properties to see if they are supported before using them. If a browser doesn't support a specific property needed for your code to work, you can either write alternative code or let the user know he needs to upgrade his browser.

That's it for this chapter. In the next chapter you move on to more exciting form scripting, where you can add various controls to your page to help you gather information from the user.

Exercise Questions

Suggested solutions to these questions can be found in Appendix A.

Question 1

Create a page with a number of links. Then write code that fires on the `window onload` event, displaying the `href` of each of the links on the page.

Question 2

Create two pages, one called `IEOnly.htm` and the other called `FFOnly.htm`. Each page should have a heading telling you what page is loaded, for example

```
<H2>Welcome to the Internet Explorer only page</H2>
```

Using the functions for checking browser type, connect to the `window` object's `onload` event handler and detect what browser the user has. Then, if it's the wrong page for that browser, redirect to the other page.

Question 3

Insert an image in the page with the `` tag. When the mouse pointer rolls over the image, it should switch to a different image. When the mouse pointer rolls out (leaves the image), it should swap back again.

6

HTML Forms — Interacting with the User

Web pages would be very boring if you could not interact with or obtain information from the user, such as text, numbers, or dates. Luckily, with JavaScript this is possible. You can use this information within the web page, or it can be posted to the web server where you can manipulate it and store it in a database if you wish. In this chapter you'll concentrate on using the information within the web browser, which is called *client-side processing*. In Chapters 14 and 15 you'll see how to send this information to a web server and store it in a database, which is called *server-side processing*.

You're quite accustomed to various user interface elements. For example, the Windows operating system has a number of standard elements, such as buttons you can click; lists, drop-down list boxes, and radio buttons you can select from; and boxes you can check. The same applies with any graphical user interface (GUI) operating system, whether it's a Mac, Unix, or Linux system. These elements are the means by which we now interface with applications. The good news is that you can include many of these types of elements in your web page — and even better, it's very easy to do so. When you have such an element — say, a button — inside your page, you can then tie code to its events. For example, when the button is clicked, you can fire off a JavaScript function you've created.

It's important to note at this point that the elements we discuss in this chapter are the common elements made available by HTML, and not ActiveX elements, Java Applets, or plug-ins. You'll look at some of these in Chapter 15.

All of the HTML elements used for interaction should be placed inside an HTML form. In Netscape 4+, it's compulsory for the elements to be inside a form; otherwise they won't be displayed. It's also compulsory if you submit the form to a server. Let's start by taking a look at HTML forms and how you interact with them in JavaScript.

HTML Forms

Forms provide you with a way of grouping together HTML interaction elements with a common purpose. For example, a form may contain elements that enable the input of a user's data for registering on a web site. Another form may contain elements that enable the user to ask for a car insurance quote. It's possible to have a number of separate forms in a single page. You don't need to worry about pages containing multiple forms until you have to submit information to a web server—then you need to be aware that the information from only one of the forms on a page can be submitted to the server at one time.

To create a form, you use the `<form>` and `</form>` tags to declare where it starts and where it ends. The `<form>` tag has a number of attributes, such as the `action` attribute, which determines where the form is submitted to; the `method` attribute, which determines how the information is submitted; and the `target` attribute, which determines the frame to which the response to the form is loaded.

Generally speaking, for client-side scripting where you have no intention of submitting information to a server, these attributes are not necessary. They will come into play in a later chapter when you look at programming server pages. For now the only attribute you need to set in the `<form>` tag is the `name` attribute, so that you can reference the form.

So, to create a blank form, the tags required would look something like this:

```
<form name="myForm">
</form>
```

You won't be surprised to hear that these tags create a `Form` object, which you can use to access the form. You can access this object in two ways.

First, you can access the object directly using its name—in this case `document.myForm`. Alternatively, you can access the object through the `document` object's `forms[]` array property. Remember that the last chapter included a discussion of the `document` object's `images[]` array and how you can manipulate it like any other array. The same applies to the `forms[]` array, except that instead of each element in the array holding an `IMG` object, it now holds a `Form` object. For example, if our `Form` is the first `Form` in the page, you reference it using `document.forms[0]`.

Many of the attributes of the `<form>` tag can be accessed as properties of the `Form` object. In particular, the `name` property of the `Form` object mirrors the `name` attribute of the `<form>` tag.

Try It Out The forms Array

Let's have a look at an example that uses the `forms` array. Here you have a page with three forms on it. Using the `forms[]` array, you access each `Form` object in turn and show the value of its `name` property in a message box.

```
<html>
<head>
<script language="JavaScript" type="text/javascript">
function window_onload()
{
    var numberForms = document.forms.length;
```



```
var formIndex;
for (formIndex = 0; formIndex < numberForms; formIndex++)
{
    alert (document.forms[formIndex].name);
}
}
</script>
</head>
<body language=JavaScript type="text/javascript" onload="window_onload()">
<form name="form1">
<p>This is inside form1</p>
</form>
<form name="form2">
<p>This is inside form2</p>
</form>
<form name="form3">
<p>This is inside form3</p>
</form>
</body>
</html>
```

Save this as `ch6_examp1.htm`. When you load it into your browser, you should see three alert boxes, each of which shows the name of a form.

How It Works

Within the body of the page you define three forms. Each form is given a name and contains a paragraph of text.

Within the definition of the `<body>` tag, the `window_onload()` function is connected to the window object's `onload` event handler.

```
<body language=JavaScript onload="return window_onload()">
```

This means that when the page is loaded, your `window_onload()` function will be called.

The `window_onload()` function is defined in a script block in the head of the page. Within this function you loop through the `forms[]` array. Just like any other JavaScript array, the `forms[]` array has a `length` property, which you can use to determine how many times you need to loop. Actually, because you know how many forms there are, you can just write the number in. However, this example uses the `length` property, since that makes it easier to add to the array without having to change the function. Generalizing your code like this is a good practice to get into.

The function starts by getting the number of `Form` objects within the `forms` array and storing that number in the variable `numberForms`.

```
function window_onload()
{
    var numberForms = document.forms.length;
```

Next you define a variable, `formIndex`, to be used in your `for` loop. After this comes the `for` loop itself.

```
var formIndex;
for (formIndex = 0; formIndex < numberForms; formIndex++)
{
    alert(document.forms[formIndex].name);
}
```

Remember that because the indexes for arrays start at 0, your loop needs to go from an index of 0 to an index of `numberForms - 1`. You enable this by initializing the `formIndex` variable to 0, and setting the condition of the `for` loop to `formIndex < numberForms`.

Within the `for` loop's code, you pass the index of the form you want (that is, `formIndex`) to `document.forms[]`, which gives you the `Form` object at that array index in the `forms` array. To access the `Form` object's `name` property, you put a dot at the end of the name of the property, `name`.

Other Form Object Properties and Methods

The HTML form controls commonly found in forms, which you will look at in more detail shortly, also have corresponding objects. One way to access these is through the `elements[]` property of the `Form` object. This is an array just like the `forms[]` array property of the `document` object that you saw earlier. The `elements[]` array contains all the objects corresponding to the HTML interaction elements within the form, with the exception of the little-used `<input type=image>` element. As you'll see later, this property is very useful for looping through each of the elements in a form. For example, you can loop through each element to check that it contains valid data prior to submitting a form.

Being an array, the `elements[]` property of the `Form` object has the `length` property, which tells you how many elements are in the form. The `Form` object also has the `length` property, which also gives you the number of elements in the form. Which of these you use is up to you because both do the same job, although writing `document.myForm.length` is shorter, and therefore quicker to type and less lengthy to look at in code, than `document.myForm.elements.length`.

When you submit data from a form to a server, you normally use the Submit button, which you will come to shortly. However, the `Form` object also has the `submit()` method, which does nearly the same thing. It differs in that it does not call the `onsubmit` event handler for the `submit` event of the `Form` object.

Recall that in the last chapter you saw how return values passed back from an event handler's code can affect whether the normal course of events continues or is canceled. You saw, for example, that returning `false` from a hyperlink's `onclick` event handler causes the link's navigation to be canceled. Well, the same principle applies to the `Form` object's `onsubmit` event handler, which fires when the user submits the form. If you return `true` to this event handler, the form submission goes ahead; if you return `false`, the submission is canceled. This makes the `onsubmit` event handler's code a great place to do form validation—that is, to check that what the user has entered into the form is valid. For example, if you ask for the user's age and she enters `mind your own business`, you can spot that this is text rather than a valid number and stop her from continuing.

In addition to there being a Reset button, which we'll discuss later in the chapter, the `Form` object has the `reset()` method, which clears the form, or restores default values if these exist.

Creating blank forms is not exactly exciting or useful, so now let's turn our attention to the HTML elements that provide interaction functionality inside our forms.

HTML Elements in Forms

About ten elements are commonly found within `<form>` elements. The most useful are shown in Figures 6-1, 6-2, 6-3, and 6-4, ordered into general types. We give each its name and, in parentheses, the HTML needed to create it, though note this is not the full HTML but only a portion.

Text Input Elements

Text Box
(`<INPUT type="text">`)

Password Box
(`<INPUT type="Password">`)
TextArea
(`<TEXTAREA></TEXTAREA>`)

Here is some text I typed in to this control

Figure 6-1

Tick Box Elements

Check boxes
(`<INPUT type= "checkbox">`)

☐
☐

Radio buttons
(`<INPUT type="radio">`)

☐
☐
☐

Figure 6-2

Select Elements

Drop Down List
(`<SELECT size="1"><OPTION value="First List Item">First List Item</OPTION></SELECT>`)

List Boxes
(`<SELECT size="4"><OPTION value="First List Item">First List Item</OPTION><OPTION value="">Second List Item</OPTION><OPTION value="">Third List Item</OPTION></SELECT>`)

Figure 6-3

Buttons

Standard Button
(`<INPUT type="button">`)

Submit Button
(`<INPUT type="submit">`)

Reset Button
(`<INPUT type="reset">`)

Figure 6-4

As you can see, most of the `<form>` elements are created by means of the `<input>` tag. One of the `<input>` tag's attributes is the `type` attribute. It's this attribute that decides which of the HTML elements this tag will be. Examples of values for this attribute include `button` (to create a button) and `text` (to create a text box).

Each form element inside the web page is made available to you as — yes, you guessed it — an object. As with all the other objects you have seen, each element's object has its own set of distinctive properties, methods, and events. You'll be taking a look at each form element in turn and how to use its particular properties, methods, and events, but before you do that, let's look at properties and methods that the objects of the form elements have in common.

Common Properties and Methods

One property that all the objects of the form elements have in common is the `name` property. You can use the value of this property to reference that particular element in your script. Also, if you are sending the information in the form to a server, the element's `name` property is sent along with any value of the form element, so that the server knows what the value relates to.

Most form element objects also have the `value` property, which returns the value of the element. For example, for a text box, the `value` property returns the text that the user has entered in the text box. Also, setting the value of the `value` property enables you to put text inside the text box. However, the use of the `value` property is specific to each element, so you'll look at what it means as you look at each individual element.

All form element objects also have the `form` property, which returns the `Form` object in which the element is contained. This can be useful in cases where you have a generic routine that checks the validity of data in a form. For example, when the user clicks a Submit button, you can pass the `Form` object referenced by the `form` property to your data checker, which can use it to loop through each element on the form in turn, checking that the data in the element are valid. This is handy if you have more than one form defined on the page or where you have a generic data checker that you cut and paste to different pages — this way you don't need to know the form's name in advance.

Sometimes it's useful to know what type of element you're dealing with, particularly where you're looping through the elements in a form using the `elements[]` array property of the `Form` object. This information can be retrieved by means of the `type` property, which each element's object has. This property returns the type of the element (for example, `button` or `text`).

All form element objects also have the `focus()` and `blur()` methods. *Focus* is a concept you might not have come across yet. If an element is the center of the *focus*, any key presses made by the user will be passed directly to that element. For example, if a text box has focus, pressing keys will enter values into the text box. Also, if a button has the focus, pressing the Enter key will cause the button's `onclick` event handler code to fire, just as if a user had clicked the button with his mouse.

The user can set which element currently has the *focus* by clicking on it or by using the Tab key to select it. However, you as the programmer can also decide which element has the focus by using the form element's object's `focus()` method. For example, if you have a text box for the user to enter his age and he enters an invalid value, such as a letter rather than a number, you can tell him that his input is invalid and send him back to that text box to correct his mistake.

Blur, which perhaps could be better called "lost focus," is the opposite of focus. If you want to remove a form element from being the focus of the user's attention, you can use the `blur()` method. When used with a form element, the `blur()` method usually results in the focus shifting to the page containing the form.

In addition to the `focus()` and `blur()` methods, all the form element's objects have the `onfocus` and `onblur` event handlers. These are fired, as you'd expect, when an element gets or loses the focus, respectively, due to user action or the `focus()` and `blur()` methods. The `onblur` event handler can be a good place to check the validity of data in the element that has just lost the focus. If the data are invalid, you can set the focus back to the element and let the user know why the data he entered are wrong.

One thing to be careful of is using the `focus()` and `blur()` methods in the `onfocus` or `onblur` event handler code. There is the danger of an infinite loop occurring. For example, consider two elements, each of whose `onfocus` events passes the focus to the other element. Then, if one element gets the focus, its `onfocus` event will pass the focus to the second element, whose `onfocus` event will pass the focus back to the first element, and so on until the only way out is to close the browser down. This is not likely to please your users!

Also be very wary of using the `focus()` and `blur()` methods to put focus back in a problem field if that field or others depend on some of the user's input. For example, say you have two text boxes, one in which you want the user to enter her city and the other in which you want her to enter her state. Also say that the input into the state text box is checked to make sure that the specified city is in that state. If the state does not contain the city, you put the focus back on the state text box so that the user can change the name of the state. However, if the user actually input the wrong city name and the right state name, she may not be able to go back to the city text box to rectify the problem.

Button Form Elements

We're starting our look at form elements with the standard button element because it's probably the most commonly used and is fairly simple. The HTML tag to create a button is the `<input>` tag. For example, to create a button called `myButton`, which has the words "Click Me" on its face, the `<input>` tag would need to be as follows:

```
<input type="button" name="myButton" value="Click Me">
```

The `type` attribute is set to `button`, and the `value` attribute is set to the text you want to appear on the face of the button. You can leave the `value` attribute off, but you'll end up with a blank button, which will leave your users guessing as to its purpose.

This element creates an associated `Button` object; in this example it is called `myButton`. This object has all the common properties and methods described earlier, including the `value` property. This property enables you to change the text on the button face using JavaScript, though this is probably not something you'll need to do very often. What the button is really all about is the `click` event.

You connect to the button's `onclick` event handler just as you did with the `onclick` events of other HTML tags such as the `<A>` tag. All you need to do is define a function that you want to have executed when the button is clicked (say, `button_onclick()`) and then add the `onclick` event handler as an attribute of the `<input>` tag as follows:

```
<input type="button" onclick="button_onclick()">
```

Try It Out Counting Button Clicks

In the following example you use the methods described previously to record how often a button has been clicked.

```
<html>
<head>
<script language="JavaScript" type="text/javascript">
var numberOfClicks = 0;
```

```
function myButton_onclick()
{
    numberOfClicks++;
    window.document.form1.myButton.value = 'Button clicked ' + numberOfClicks +
    ' times';
}
</script>
</head>
<body>
<form name=form1>
    <input type='button' name='myButton' value='Button clicked 0 times'
    onclick="myButton_onclick()" >
</form>
</body>
</html>
```

Save this page as `ch6_examp2.htm`. If you load this page into your browser, you will see a button with “Button clicked 0 times” on it. If you repeatedly press this button, you will see the number of button clicks recorded on the text of the button.

How It Works

You start the script block in the head of the page by defining a global variable, accessible anywhere inside your page, called `numberOfClicks`. You record the number of times the button has been clicked in this variable, and use this information to update the button’s text.

The other piece of code in the script block is the definition of the function `myButton_onclick()`. This function is connected to the `onclick` event handler in the `<input>` tag in the body of the page. This tag is for a button element called `myButton` and is contained within a form called `form1`.

```
<form name=form1>
    <input type='button' name='myButton' value='Button clicked 0 times'
    onclick="myButton_onclick()" >
</form>
```

Let’s look at the `myButton_onclick()` function a little more closely. First, the function increments the value of the variable `numberOfClicks` by one.

```
function myButton_onclick()
{
    numberOfClicks++;
```

Next, you update the text on the button face using the `Button` object’s `value` property.

```
    window.document.form1.myButton.value = 'Button clicked ' + numberOfClicks +
    ' times';
}
```

The function in this example is specific to this form and button, rather than a generic function you’ll be using in other situations. Therefore the code in this example refers to the form and button directly using `window.document.form1.myButton`. Remember that the `window` object has a property containing the document object, which itself holds all the elements in a page, including the `<form>` element, and that the button is embedded inside your form.

Try It Out onmouseup and onmousedown

Two less commonly used events supported by the `Button` object are the `onmousedown` and `onmouseup` events. You can see these two events in action in the next example.

```
<html>
<head>
<script language="JavaScript" type="text/javascript">
function myButton_onmouseup()
{
    document.form1.myButton.value = "Mouse Goes Up"
}
function myButton_onmousedown()
{
    document.form1.myButton.value = "Mouse Goes Down"
}
</script>
</head>
<body>
<form name=form1>
    <input type='button' name='myButton' value=' Mouse Goes Up '
        onmouseup="myButton_onmouseup()"
        onmousedown="myButton_onmousedown()" ">
</form>
</body>
</html>
```

Save this page as `ch6_examp3.htm` and load it into your browser. If you click the button with your left mouse button and keep it held down, you'll see the text on the button change to "Mouse Goes Down." As soon as you release the button, the text changes to "Mouse Goes Up."

How It Works

In the body of the page you define a button called `myButton` within a form called `form1`. Within the attributes of the `<input>` tag you attach the function `myButton_onmouseup()` to the `onmouseup` event handler, and the function `myButton_onmousedown()` to the `onmousedown` event handler.

```
<form name=form1>
    <input type='button' name='myButton' value=' Mouse Goes Up '
        onmouseup="myButton_onmouseup()"
        onmousedown="myButton_onmousedown()" ">
</form>
```

The `myButton_onmouseup()` and `myButton_onmousedown()` functions are defined in a script block in the head of the page. Each function consists of just a single line of code, in which you use the `value` property of the `Button` object to change the text that is displayed on the button's face.

An important point to note is that events like `onmouseup` and `onmousedown` are triggered only when the mouse pointer is actually over the element in question. For example, if you click and hold down the mouse button over your button, then move the mouse away from the button before releasing the mouse button, you'll find that the `onmouseup` event does not fire and the text on the button's face does not change. In this instance it would be the `document` object's `onmouseup` event handler code that would fire, if you'd connected any code to it.

Don't forget that, like all form element objects, the `Button` object also has the `onfocus` and `onblur` events, though they are rarely used in the context of buttons.

The Submit and Reset Buttons

Two additional button types are the Submit and Reset buttons. You define these buttons just as you do a standard button, except that the `type` attribute of the `<input>` tag is set to `submit` or `reset` rather than to `button`. For example, the Submit and Reset buttons in Figure 6-4 were created using the following code:

```
<input type="submit" value="Submit" name="submit1">  
<input type="reset" value="Reset" name="reset1">
```

These buttons have special purposes, which are not related to script.

When the Submit button is clicked, the form data from the form that the button is inside gets sent to the server automatically, without the need for any script.

When the Reset button is clicked, all the elements in a form are cleared and returned to their default values (the values they had when the page was first loaded).

The Submit and Reset buttons have corresponding objects called `Submit` and `Reset`, which have exactly the same properties, methods, and events as a standard `Button` object.

Text Elements

The standard text element enables users to enter a single line of text. This information can then be used in JavaScript code or submitted to a server for server-side processing.

A text box is created by means of the `<input>` tag, much as your button was, but with the `type` attribute set to `text`. Again, you can choose not to include the `value` attribute, but if you do include it this value will appear inside the text box when the page is loaded.

In the following example the `<input>` tag has two additional attributes, `size` and `maxlength`. The `size` attribute determines how many characters wide the text box is, and `maxlength` determines the maximum number of characters the user can enter in the box. Both attributes are optional and use defaults determined by the browser.

For example, to create a text box 10 characters wide, with a maximum character length of 15, and initially containing the words `Hello World`, your `<input>` tag would be as follows:

```
<input type="text" name="myTextBox" size=10 maxlength=15 value="Hello World">
```

The `Text` object that this element creates has a `value` property, which you can use in your scripting to set or read the text contained inside the text box. In addition to the common properties and methods we discussed earlier, the `Text` object also has the `select()` method, which selects or highlights all the text inside the text box. This may be used if the user has entered an invalid value, and you can set the `focus` to the text box and select the text inside it. This then puts the user's cursor in the right place to correct the data and makes it very clear to the user where the invalid data is. The `value` property of `Text` objects always returns a string data type, even if number characters are being entered. If you use the

Chapter 6: HTML Forms — Interacting with the User

value as a number, JavaScript normally does a conversion from a string data type to a number data type for you, but this is not always the case. For example, JavaScript won't do the conversion if the operation you're doing is valid for a string. If you have a form with two text boxes and you add the values returned from these, JavaScript concatenates rather than adds the two values, so 1 plus 1 will be 11 and not 2. To fix this, you need to convert all the values involved to a numerical data type, for example by using `parseInt()` or `parseFloat()` or `Number()`. However, if you subtract the two values, an operation only valid for numbers, JavaScript says "Aha, this can only be done with numbers, so I'll convert the values to a number data type." Therefore, 1 minus 1 will be returned as 0 without your having to use `parseInt()` or `parseFloat()`. This is a tricky bug to spot, so it's best to get into the habit of converting explicitly to avoid problems later.

In addition to the common event handlers, such as `onfocus` and `onblur`, the `Text` object has the `onchange`, `onselect`, `onkeydown`, `onkeypress`, and `onkeyup` event handlers.

The `onselect` event fires when the user selects some text in the text box.

More useful is the `onchange` event, which fires when the element loses focus if (and only if) the value inside the text box is different from the value it had when it got the focus. This enables you to do things like validity checks that occur only if something has changed.

As mentioned before, the `onfocus` and `onblur` events can be used for validating user input. However, they also have another purpose, and that's to make a text box read-only. In IE 4.0+ and NN 6 you can use the `READONLY` attribute of the `<input>` tag or the `readOnly` property of the `Text` object to prevent the contents from being changed. However, these techniques won't work in NN 4.x. You can get around this using the `blur()` method. All you need to do is add an `onfocus` event handler to the `<input>` tag defining the text box and connect it to some code that blurs the focus from the text box with the `blur()` method.

```
<input type="text" name=txtReadOnly value="Look but don't change"
      onfocus="window.document.form1.txtReadOnly.blur()"
      READONLY=true>
```

The `onkeypress`, `onkeydown`, and `onkeyup` events fire, as their names suggest, when the user presses a key, when the user presses a key down, and when a key pressed down is let back up respectively.

Try It Out A Simple Form with Validation

Let's put all the information on text boxes and buttons together into an example. In this example you have a simple form consisting of two text boxes and a button. The top text box is for the user's name, and the second is for her age. You do various validity checks. You check the validity of the age text box when it loses focus. However, the name and age text boxes are only checked to see if they are empty when the button is clicked.

```
<html>
<head>
<script language="JavaScript" type="text/javascript">
function butCheckForm_onclick()
{
    var myForm = document.form1;
    if (myForm.txtAge.value == "" || myForm.txtName.value == "")
```

```
{
    alert("Please complete all the form");
    if (myForm.txtName.value == "")
    {
        myForm.txtName.focus();
    }
    else
    {
        myForm.txtAge.focus();
    }
}
else
{
    alert("Thanks for completing the form " + myForm.txtName.value);
}
}
function txtAge_onblur()
{
    var txtAge = document.form1.txtAge;
    if (isNaN(txtAge.value) == true)
    {
        alert("Please enter a valid age");
        txtAge.focus();
        txtAge.select();
    }
}
function txtName_onchange()
{
    window.status = "Hi " + document.form1.txtName.value;
}
</script>
</head>
<body>
<form name=form1>
    Please enter the following details:
    <p>
    Name:
    <br>
    <input type="text" name=txtName onchange="txtName_onchange()">
    <br>
    Age:
    <br>
    <input type="text" name=txtAge onblur="txtAge_onblur()" size=3 maxlength=3>
    <br>
    <input type="button" value="Check Details" name=butCheckForm
        onclick="butCheckForm_onclick()">
</form>
</body>
</html>
```

After you've entered the text, save the file as `ch6_examp4.htm` and load it into your web browser.

In the text box shown in Figure 6-5, type your name. When you leave the text box you'll see `Hi yourname` appear in the status bar at the bottom of the window.

Chapter 6: HTML Forms — Interacting with the User

Enter an invalid value into the age text box, such as aaaa, and when you try to leave the box, it'll tell you of the error and send you back to correct it.

Finally, click the Check Details button and both text boxes will be checked to see that you have completed them. If either is empty, you'll get a message telling you to complete the whole form, and it'll send you back to the box that's empty.

If everything is filled in correctly, you'll get a message thanking you, as shown in Figure 6-5.

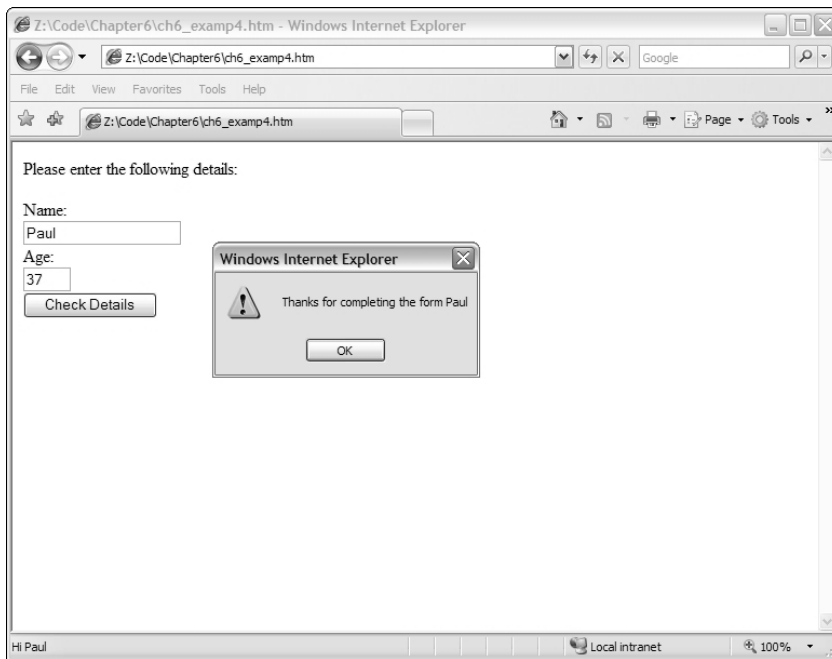


Figure 6-5

Note that this example does not work properly on Firefox or Netscape browsers; we'll discuss why shortly.

How It Works

Within the body of the page, you create the HTML tags that define your form. Inside your form, which is called `form1`, you create three form elements with the names `txtName`, `txtAge`, and `butCheckForm`.

```
<form name=form1>
  Please enter the following details:
  <p>
  Name:
  <br>
  <input type="text" name=txtName onchange="txtName_onchange()" >
  <br>
  Age:
```

```
<br>
<input type="text" name=txtAge onblur="txtAge_onblur()" size=3 maxlength=3>
<br>
<input type="button" value="Check Details" name=butCheckForm
      onclick="butCheckForm_onclick()">
</form>
```

You'll see that for the second text box (the `txtAge` text box), you have included the `size` and `maxlength` attributes inside the `<input>` tag. Setting the `size` attribute to 3 gives the user an idea of how much text you are expecting, and setting the `maxlength` attribute to 3 helps ensure that you don't get overly large numbers entered for the age value!

The first text box's `onchange` event handler is connected to the function `txtName_onchange()`, the second text box's `onblur` event handler is connected to the function `txtAge_onblur()`, and the button's `onclick` event handler is connected to the function `butCheckForm_onclick()`. These functions are defined in a script block in the head of the page. You will look at each of them in turn, starting with `butCheckForm_onclick()`.

The first thing you do is define a variable, `myForm`, and set it to reference the `Form` object created by your `<form>` tag later in the page.

```
function butCheckForm_onclick()
{
    var myForm = document.form1;
```

Doing this reduces the size of your code each time you want to use the `form1` object. Instead of `document.form1` you can just type `myForm`. It makes your code a bit more readable and therefore easier to debug, and it saves typing. When you set a variable to be equal to an existing object, you don't (in this case) actually create a new `form1` object; instead you just point your variable to the existing `form1` object. So when you type `myForm.name`, JavaScript checks your variable, finds it's actually storing the location in memory of the object `form1`, and uses that object instead. All this goes on behind the scenes so you don't need to worry about it and can just use `myForm` as if it were `document.form1`.

After getting your reference to the `Form` object, you then use it in an `if` statement to check whether the value in the text box named `txtAge` or the text box named `txtName` actually contains any text.

```
if (myForm.txtAge.value == "" || myForm.txtName.value == "")
{
    alert("Please complete all the form");
    if (myForm.txtName.value == "")
    {
        myForm.txtName.focus();
    }
    else
    {
        myForm.txtAge.focus();
    }
}
```

If you do find an incomplete form, you alert the user. Then in an inner `if` statement, you check which text box was not filled in. You set the focus to the offending text box, so that the user can start filling it in

Chapter 6: HTML Forms — Interacting with the User

straightaway without having to move the focus to it herself. It also lets the user know which text box our program requires her to fill in. To avoid annoying your users, make sure that text in the page tells them which fields are required.

If the original outer `if` statement finds that the form is complete, it lets the user know with a thank-you message.

```
else
{
    alert("Thanks for completing the form " + myForm.txtName.value);
}
```

In this sort of situation, it's probably more likely to submit the form to the server than to let the user know with a thank-you message. You can do this using the `Form` object's `submit()` method, or with a normal Submit button.

The next of your three functions is `txtAge_onblur()`, which is connected to the `onblur` event of your `txtAge` text box. The purpose of this function is to check that the string value the user entered into the age box actually consists of number characters.

```
function txtAge_onblur()
{
    var txtAge = document.form1.txtAge;
```

Again at the start of the function, you declare a variable and set it to reference an object; this time it's the `Text` object created for the `txtAge` text box that you define further down the page. Now, instead of having to type `document.form1.txtAge` every time, you just type `txtAge`, and it does the same thing. It certainly helps save those typing fingers, especially since it's a big function with multiple use of the `txtAge` object.

The following `if` statement checks to see whether what has been entered in the `txtAge` text box can be converted to a number. You use the `isNaN()` function to do this for you. If the value in the `txtAge` text box is not a number, it's time to tell the user and set the focus back to the offending element with the `focus()` method of the corresponding `Text` object. Additionally, this time you highlight the text by using the `Text` object's `select()` method. This makes it even clearer to the user what he has to fix, and he can rectify the problem without needing to delete text first.

```
if (isNaN(txtAge.value) == true)
{
    alert("Please enter a valid age");
    txtAge.focus();
    txtAge.select();
}
```

You could go further and check that the number inside the text box is actually a valid age—for example, 191 is not a valid age, nor is 255 likely to be. You just need to add another `if` statement to check for these possibilities.

This function is connected to the `onblur` event handler of the `txtAge` text box, but why didn't you use the `onchange` event handler, with its advantage that it only rechecks the value when the value has actually been changed? The `onchange` event would not fire if the box was empty both before focus was passed to it and after focus was passed away from it. However, leaving the checking of the form completion until just before the form is submitted is probably best because some users prefer to fill in information out of order and come back to some form elements later.

The final function is for the `txtName` text box's `onchange` event. Its use here is a little flippant and intended primarily as an example of the `onchange` event.

```
function txtName_onchange()
{
    window.status = "Hi " + document.form1.txtName.value;
}
```

When the `onchange` event fires (when focus is passed away from the name text box and its contents have changed), you take the value of the `txtName` box and put it into the window's status bar at the bottom of the window. It simply says *Hi yourname*. You access the status bar using the window object's `status` property, although you could just enter the following:

```
status = "Hi " + document.form1.txtName.value;
```

Here `window` is in front just to make it clear what you are actually accessing. It would be very easy when reading the code to mistake `status` for a variable, so in this situation, although it isn't strictly necessary, putting `window` in front does make the code easier to read, understand, and therefore debug.

Problems with Firefox, Netscape, and the blur Event

The previous example will fail with Firefox and Netscape if you enter a name in the name text box and then an invalid age into the age box (for example, if you enter `abc` and then click the Check Form button). With IE the `blur` event fires and displays an alert box if the age is invalid, but the button's click event doesn't fire. However, in FF/NN, both events fire with the result that the invalid age alert is hidden by the "form completed successfully" alert box.

In addition, if you enter an invalid age for both IE and Firefox/Netscape browsers and then switch to a different program altogether, the "invalid age" alert box appears, which is annoying for the user. It could be that the user was opening up another program to check the details.

Although this is a fine example, it is not great for the real world. A better option would be to check the form when it's finally submitted and not while the user is entering data. Or, alternatively, you can check the data as they are entered but not use an alert box to display errors. Instead you could write out a warning in red next to the erroneous input control, informing the user of the invalid data, and then also get your code to check the form when it's submitted. In the dynamic HTML chapters you'll see how to write to the page after it's been loaded.

The Password Text Box

The only real purpose of the password box is to enable users to type in a password on a page and to have the password characters hidden, so that no one can look over the user's shoulder and discover his or her password. However, when sent to the server, the text in the password is sent as plain text—there is no encryption or any attempt at hiding the text—so it's not a secure way of passing information.

Chapter 6: HTML Forms — Interacting with the User

Defining a password box is identical to defining a text box, except that the `type` attribute is `password`.

```
<input name="password1" type="password">
```

This form element creates an associated `Password` object, which is identical to the `Text` object in its properties, methods, and events.

The Hidden Text Box

The hidden text box can hold text and numbers just like a normal text box, with the difference being that it's not visible to the user. A hidden element? It may sound as useful as an invisible painting, but in fact it proves to be very useful.

To define a hidden text box, you have the following HTML:

```
<input type="hidden" name=myHiddenElement>
```

The hidden text box creates a `Hidden` object. This is available in the `elements` array property of the `Form` object and can be manipulated in JavaScript like any other object, although you can actually set its value only through its HTML definition or through JavaScript. As with a normal text box, its value is submitted to the server when the user submits the form.

So why are hidden text boxes useful? Imagine you have a lot of information that you need to obtain from the user, but to avoid having a page stuffed full of elements and looking like the control panel of the space shuttle, you decide to obtain the information over more than one page. The problem is, how do you keep a record of what was entered in previous pages? Easy — you use hidden text boxes and put the values in there. Then, in the final page, all the information is submitted to the server — it's just that some of it is hidden. Anyway, you'll see more about this in Chapter 16.

textarea Element

The `textarea` element allows multi-line input of text. Other than this, it acts very much like the text box element.

However, unlike the text box, the `textarea` element has its own tag, the `<textarea>` tag. It also has two additional attributes: `COLS` and `ROWS`. The `COLS` attribute defines how many characters wide the text area will be, and the `ROWS` attribute defines how many character rows there will be. You set the text inside the element by putting it between the start and close tags, rather than by using the `value` attribute. So if you want a `textarea` element 40 characters wide by 20 rows deep with initial text `Hello World` on the first line and `Line 2` on the second line, you define it as follows:

```
<textarea name=myTextArea cols=40 rows=20>Hello World  
Line 2  
</textarea>
```

Another attribute of the `<textarea>` tag is the `wrap` attribute, which determines what happens when the user types to the end of a line. The default value for this is `soft`, so the user does not have to press Return at the end of a line, though this can vary from browser to browser. To turn wrapping on, you can use one of two values: `soft` (this includes just the `wrap` attribute on its own) and `hard`. As far as client-side processing goes, both do the same thing: they switch wrapping on. However, when you come to

server-side processing, they do make a difference in terms of which information is sent to the server when the form is posted.

If you set the `wrap` attribute on by setting it to `soft`, wrapping will occur on the client side, but the carriage returns won't be posted to the server, just the text. If the `wrap` attribute is set to `hard`, any carriage returns caused by wrapping will be converted to hard returns — it will be as if the user had pressed the Enter key, and these returns will be sent to the server. Also, you need to be aware that the carriage-return character is determined by the operating system that the browser is running on — for example, in Windows a carriage return is `\r\n`, whereas on a Macintosh the carriage return is `\r` and on Unix a carriage return is `\n`. To turn off wrapping client-side, set `wrap` to `off`.

The `Textarea` object created by the `<textarea>` tag has the same properties, methods, and events as the `Text` object you saw previously, except that the text area doesn't have the `maxlength` attribute. Note that there is a `value` property even though the `<textarea>` tag does not have a `value` attribute. The `value` property simply returns the text between the `<textarea>` and `</textarea>` tags. The events supported by the `Textarea` object include the `onkeydown`, `onkeypress`, `onkeyup`, and `onchange` event handlers.

Try It Out Event Watching

To help demonstrate how the `keydown`, `keypress`, `keyup`, and `change` events work (in particular, the order in which they fire), you'll create an example that tells you what events are firing.

```
<html>
<head>
<script language="JavaScript" type="text/javascript">

function DisplayEvent(eventName)
{
    var myMessage = window.document.form1.textarea2.value;
    myMessage = myMessage + eventName;
    window.document.form1.textarea2.value = myMessage;
}
</script>
</head>

<body>
<form name=form1>
    <textarea rows=15 cols=40 name=textarea1
        onchange="DisplayEvent('onchange\n');"
        onkeydown="DisplayEvent('onkeydown\n');"
        onkeypress="DisplayEvent('onkeypress\n');"
        onkeyup="DisplayEvent('onkeyup\n\n');"></textarea>
    <textarea rows=15 cols=40 name=textarea2></textarea>
    <br><br>
    <input type="button" value="Clear Event TextArea"
        name=button1 onclick="window.document.form1.textarea2.value=''">
</form>
</body>
</html>
```

Chapter 6: HTML Forms — Interacting with the User

Save this page as `ch6_examp5.htm`. Load the page into your browser and see what happens when you type any letter into the first text area box. You should see the events being fired listed in the second text area box (`onkeydown`, `onkeypress`, and `onkeyup`), as shown in Figure 6-6. When you click outside the first text area box, you'll see the `onchange` event fire.

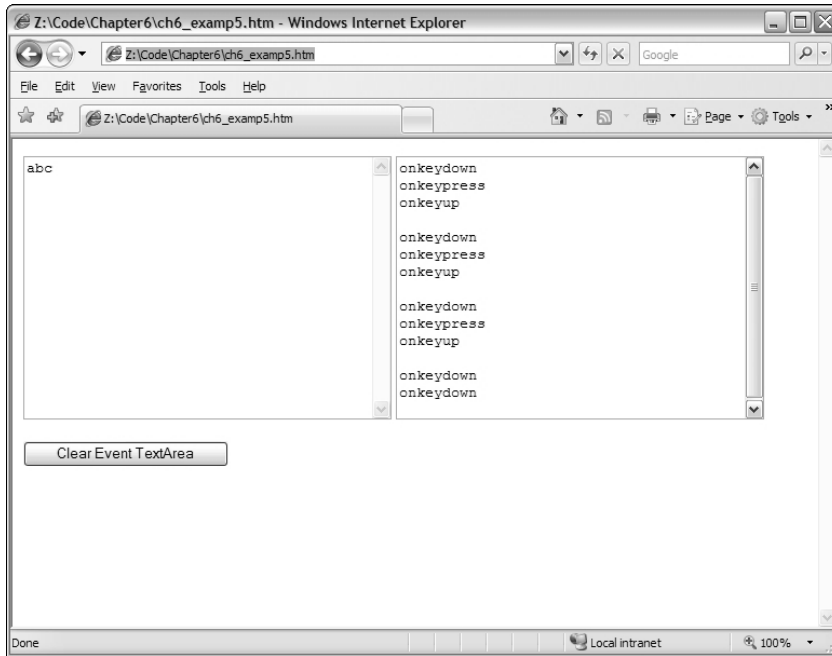


Figure 6-6

Experiment with the example to see what events fire and when.

How It Works

Within a form called `form1` in the body of the page, you define two text areas and a button. The first text area is the one whose events you are going to monitor. You attach code that calls the `DisplayEvent()` function to each of the `onchange`, `onkeydown`, `onkeypress`, and `onkeyup` event handlers. The value passed to the function reflects the name of the event firing.

```
<textarea rows=15 cols=40 name=textarea1
  onchange="DisplayEvent('onchange\n');"
  onkeydown="DisplayEvent('onkeydown\n');"
  onkeypress="DisplayEvent('onkeypress\n');"
  onkeyup="DisplayEvent('onkeyup\n\n');"></textarea>
```

Next you have an empty text area the same size as the first.

```
<textarea rows=15 cols=40 name=textarea2></textarea>
```

Finally you have your button element.

```
<input type="button" value="Clear Event TextArea"  
      NAME=button1 onclick="window.document.form1.textarea2.value=' '>
```

Notice that the `onclick` event handler for the button is not calling a function, but just executing a line of JavaScript. Although you normally call functions, it's not compulsory; if you have just one line of code to execute, it's easier just to insert it rather than create a function and call it. In this case, the `onclick` event handler is connected to some code that sets the contents of the second text area to empty (' ').

Now let's look at the `DisplayEvent()` function. This is defined in a script block in the head of the page. It adds the name of the event handler that has been passed as a parameter to the text already contained in the second text area.

```
function DisplayEvent(eventName)  
{  
    var myMessage = window.document.form1.textarea2.value;  
    myMessage = myMessage + eventName;  
    window.document.form1.textarea2.value = myMessage;  
}
```

Check Boxes and Radio Buttons

The discussions of check boxes and radio buttons are together because their objects have identical properties, methods, and events. A check box enables the user to check and uncheck it. It is similar to the paper surveys you may get where you are asked to "check the boxes that apply to you." Radio buttons are basically a group of check boxes, with the property that only one can be checked at once. Of course, they also look different, and their group nature means that they are treated differently.

Creating check boxes and radio buttons requires our old friend the `<input>` tag. Its `type` attribute is set to "checkbox" or "radio" to determine which box or button is created. To set a check box or a radio button to be checked when the page is loaded, you simply insert the keyword `checked` into the `<input>` tag. This is handy if you want to set a default option like, for example, those "Check this box if you want our junk mail" forms you often see on the Net, which are usually checked by default, forcing you to uncheck them. So to create a check box that is already checked, your `<input>` tag will be the following:

```
<input type="checkbox" name=chkDVD checked value="DVD">
```

To create a checked radio button, the `<input>` tag would be as follows:

```
<input type="radio" name=radCPUSpeed checked value="1 GHz">
```

As previously mentioned, radio buttons are group elements. In fact, there is little point in putting just one on a page, because the user won't be able to choose between any alternative boxes.

To create a group of radio buttons, you simply give each radio button the same `name`. This creates an array of radio buttons going by that name that you can access, as you would with any array, using its index.

Chapter 6: HTML Forms — Interacting with the User

For example, to create a group of three radio buttons, your HTML would be as follows:

```
<input type="radio" name=radCPUSpeed checked value="800 MHz">
<input type="radio" name=radCPUSpeed value="1 GHz">
<input type="radio" name=radCPUSpeed value="1.5 GHz">
```

You can put as many groups of radio buttons in a form as you want, by just giving each group its own unique name. Note that you have only used one `CHECKED` attribute, since only one of the radio buttons in the group can be checked. If you had used the `CHECKED` attribute in more than one of the radio buttons, only the last of these would have actually been checked.

Using the `value` attribute of the check box and radio button elements is not the same as with previous elements you’ve looked at. First, it tells you nothing about the user’s interaction with an element, because it’s predefined in your HTML or by your JavaScript. Whether a check box or radio button is checked or not, it still returns the same value. Second, when a form is posted to a server, only the values of the checked check boxes and radio buttons are sent. So, if you have a form with 10 check boxes and the user submits the form with none of these checked, then nothing is sent to the server except a blank form. You’ll learn more about this when you look at server-side scripting in Chapter 16.

Each check box has an associated `Checkbox` object, and each radio button in a group has a separate `Radio` object. As mentioned earlier, with radio buttons of the same name you can access each `Radio` object in a group by treating the group of radio buttons as an array, with the name of the array being the name of the radio buttons in the group. As with any array, you have the `length` property, which will tell you how many radio buttons are in the group.

For determining whether a user has actually checked or unchecked a check box, you need to use the `checked` property of the `Checkbox` object. This property returns `true` if the check box is currently checked and `false` if not.

Radio buttons are slightly different. Because radio buttons with the same name are grouped together, you need to test each `Radio` object in the group in turn to see if it has been checked. Only one of the radio buttons in a group can be checked, so if you check another one in the group, the previously checked one will become unchecked, and the new one will be checked in its place.

Both `Checkbox` and `Radio` have the event handlers `onclick`, `onfocus`, and `onblur`, and these operate as you saw for the other elements, although they can also be used to cancel the default action, such as clicking the check box or radio button.

Try It Out Check Boxes and Radio Buttons

Let’s look at an example that makes use of all the properties, methods, and events we have just discussed. The example is a simple form that enables a user to build a computer system. Perhaps it could be used in an e-commerce situation, to sell computers with the exact specifications determined by the customer.

```
<html>
<head>
<script language="JavaScript" type="text/javascript">

var radCpuSpeedIndex = 0;

function radCPUSpeed_onclick(radIndex)
```

```

{
    var returnValue = true;
    if (radIndex == 1)
    {
        returnValue = false;
        alert("Sorry that processor speed is currently unavailable");
        // Next line works around a bug in IE that doesn't cancel the
        // Default action properly
        document.form1.radCpuSpeed[radCpuSpeedIndex].checked = true;
    }
    else
    {
        radCpuSpeedIndex = radIndex;
    }
    return returnValue;
}
function butCheck_onclick()
{
    var controlIndex;
    var element;
    var numberOfControls = document.form1.length;
    var compSpec = "Your chosen processor speed is ";
    compSpec = compSpec + document.form1.radCpuSpeed[radCpuSpeedIndex].value;
    compSpec = compSpec + "\nWith the following additional components\n";
    for (controlIndex = 0; controlIndex < numberOfControls; controlIndex++)
    {
        element = document.form1[controlIndex];
        if (element.type == "checkbox")
        {
            if (element.checked == true)
            {
                compSpec = compSpec + element.value + "\n";
            }
        }
    }
    alert(compSpec);
}
</script>
</head>
<body>
<form name=form1>
<p> Tick all of the components you want included on your computer <br><br>
<table> <tr>
    <td>DVD-ROM</td>
    <td><input type="checkbox" name="chkDVD" value="DVD-ROM"></td>
</tr> <tr>
    <td>CD-ROM</td>
    <td><input type="checkbox" name="chkCD" value="CD-ROM"></td>
</tr> <tr>
    <td>Zip Drive</td>
    <td><input type="checkbox" name="chkZip" value="ZIP Drive"></td>
</tr>
</table>
<p>

```

Chapter 6: HTML Forms — Interacting with the User

```
Select the processor speed you require <table>
<tr>
  <td><input type="radio" name="radCPUSpeed" checked
    onclick="return radCPUSpeed_onclick(0)" value="3.8 GHz"></td>
  <td><input type="radio" name="radCPUSpeed"
    onclick="return radCPUSpeed_onclick(1)" value="4.8 GHz"></td>
  <td>4.8 GHz</td>
  <td><input type="radio" name="radCPUSpeed"
    onclick="return radCPUSpeed_onclick(2)" value="6 GHz"></td>
  <td>6 GHz</td> </tr>
</table>
</p> <input type="button" value="Check Form" name="butCheck"
  onclick="return butCheck_onclick()" >
</form>
</body>
</html>
```

Save the page as `ch6_exam6.htm` and load it into your web browser. You should see a form like the one shown in Figure 6-7.

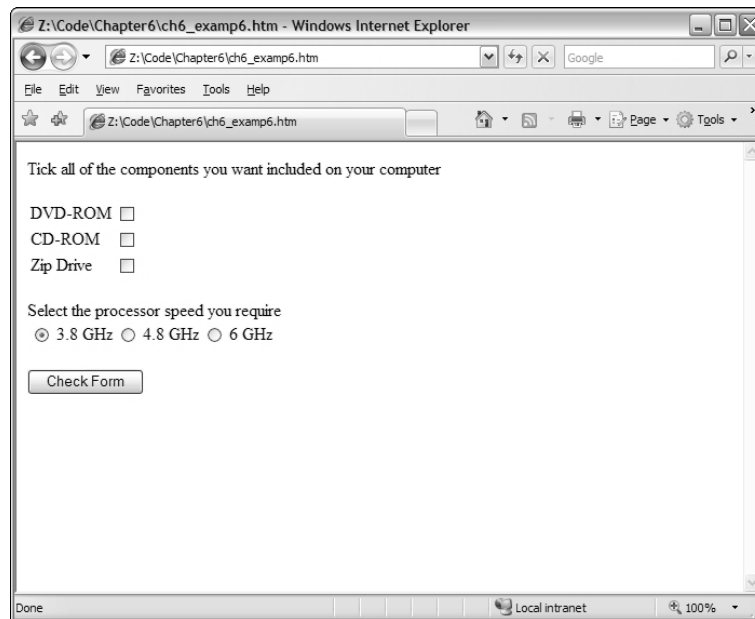


Figure 6-7

Check some of the check boxes, change the processor speed, and click the Check Form button. A message box will appear, listing the components and processor speed you selected. For example, if you select a DVD-ROM and a Zip drive and a 6 GHz processor speed, you will see something like what is shown in Figure 6-8.



Figure 6-8

Note that the 4.8 GHz processor is out of stock, so if you choose that, a message box will appear telling you that it's out of stock, and the 4.8 GHz processor speed radio button won't be selected. The previous setting will be restored when the user dismisses the message box.

How It Works

Let's first look at the body of the page, where you define the check boxes and radio buttons and a standard button inside a form called `form1`. You start with the check boxes. They are put into a table simply for formatting purposes. No functions are called, and no events are linked to.

```
<table>
<tr>
  <td>DVD-ROM</td>
  <td><input type="checkbox" name=chkDVD value="DVD-ROM"></td>
</tr>
<tr>
  <td>CD-ROM</td>
  <td><input type="checkbox" name=chkCD value="CD-ROM"></td>
</tr>
<tr>
  <td>Zip Drive</td>
  <td><input type="checkbox" name=chkZip value="ZIP Drive"></td>
</tr>
</table>
```

Next come the radio buttons for selecting the required CPU speed, and these are a little more complex. Again they are put into a table for formatting purposes.

```
<table>
<tr>
  <td><input type="radio" name=radCPUSpeed checked
    onclick="return radCPUSpeed_onclick(0)" value="3.8 GHz"></td>
  <td>3.8 GHz</td>
  <td><input type="radio" name=radCPUSpeed
    onclick="return radCPUSpeed_onclick(1)" value="4.8 GHz"></td>
  <td>4.8 GHz</td>
  <td><input type="radio" name=radCPUSpeed
    onclick="return radCPUSpeed_onclick(2)" value="6 GHz"></td>
  <td>6 GHz</td>
</tr>
</table>
```

Chapter 6: HTML Forms — Interacting with the User

The radio button group name is `radCPUSpeed`. Here the first one is set to be checked by default by the inclusion of the word `CHECKED` inside the `<input>` tag's definition. It's a good idea to ensure that you have one radio button checked by default, because if you do not and the user doesn't select a button, the form will be submitted with no value for that radio group.

You're making use of the `onclick` event of each `Radio` object. For each button you're connecting to the same function, `radCPUSpeed_onclick()`, but for each radio button, you are passing a value—the index of that particular button in the `radCPUSpeed` radio button group array. This makes it easy to determine which radio button was selected. You'll look at this function a little later, but first let's look at the standard button that completes your form.

```
<input type="button" value="Check Form" name=butCheck
      onclick="return butCheck_onclick()">
```

This button's `onclick` event handler is connected to the `butCheck_onclick()` function and is for the user to click when she has completed the form.

So you have two functions, `radCPUSpeed_onclick()` and `butCheck_onclick()`. These are both defined in the script block in the head of the page. Let's look at this script block now. It starts by declaring a variable `radCpuSpeedIndex`. This will be used to store the currently selected index of the `radCPUSpeed` radio button group.

```
var radCpuSpeedIndex = 0;
```

Next you have the `radCPUSpeed_onclick()` function, which is called by the `onclick` event handler in each radio button. Your function has one parameter, namely the index position in the `radCPUSpeed[]` array of the radio object selected.

```
function radCPUSpeed_onclick(radIndex)
{
    var returnValue = true;
```

The first thing you do in the function is declare the `returnValue` variable and set it to `true`. You'll be returning this as your return value from the function. In this case the return value is important because it decides whether the radio button remains checked as a result of the user clicking it. If you return `false`, that cancels the user's action, and the radio button remains unchecked. In fact no radio button becomes checked, which is why you keep track of the index of the checked radio button so you can track which button was the previously checked one. To allow the user's action to proceed, you return `true`.

As an example of this in action, you have an `if` statement on the next line. If the radio button's index value passed is 1 (that is, if the user checked the box for a 4.8 GHz processor), you tell the user that it's out of stock and cancel the clicking action by setting `returnValue` to `false`.

```
if (radIndex == 1)
{
    returnValue = false;
    alert("Sorry that processor speed is currently unavailable");
    document.form1.radCPUSpeed[radCpuSpeedIndex].checked = true;
}
```


As previously mentioned, canceling the clicking action results in no radio buttons being checked. To rectify this, you set the previously checked box to be checked again in the following line:

```
document.form1.radCPUSpeed[radCpuSpeedIndex].checked = true;
```

What you are doing here is using the Array object for the `radCpuSpeed` radio group. Each array element actually contains an object, namely each of your three `Radio` objects. You use the `radCpuSpeedIndex` variable as the index of the `Radio` object that was last checked, since this is what it holds.

Finally, in the `else` statement you set `radCpuSpeedIndex` to the new checked radio button's index value.

```
else
{
    radCpuSpeedIndex = radIndex;
}
```

In the last line of the function, the value of `returnValue` is returned to where the function was called and will either cancel or allow the clicking action.

```
return returnValue;
}
```

Your second function, `butCheck_onclick()`, is the one connected to the button's `onclick` event. In a real e-commerce situation, this button would be the place where you'd check your form and then submit it to the server for processing. Here you use the form to show a message box confirming which boxes you have checked, as if you didn't already know!

At the top you declare the four local variables, which will be used in the function. The variable `numberOfControls` is set to the form's `length` property, which is the number of elements on the form. The variable `compSpec` is used to build up the string that you'll display in a message box.

```
function butCheck_onclick()
{
    var controlIndex;
    var element;
    var numberOfControls = document.form1.length;
    var compSpec = "Your chosen processor speed is ";
    compSpec = compSpec + document.form1.radCPUSpeed[radCpuSpeedIndex].value;
    compSpec = compSpec + "\nWith the following additional components\n";
```

In the following line you add the value of the radio button the user has selected to your message string:

```
compSpec = compSpec + document.form1.radCPUSpeed[radCpuSpeedIndex].value;
```

The global variable `radCpuSpeedIndex`, which was set by the radio button group's `onclick` event, contains the array index of the selected radio button.

Chapter 6: HTML Forms — Interacting with the User

An alternative way of finding out which radio button was clicked would be to loop through the radio button group's array and test each radio button in turn to see if it was checked. The code would look something like this:

```
var radIndex;
for (radIndex = 0; radIndex < document.form1.radCPUSpeed.length; radIndex++)
{
    if (document.form1.radCPUSpeed[radIndex].checked == true)
    {
        radCpuSpeedIndex = radIndex;
        break;
    }
}
```

But to get back to the actual code, you'll notice a few new-line (`\n`) characters thrown into the message string for formatting reasons.

Next you have your big `for` statement.

```
for (controlIndex = 0; controlIndex < numberOfControls; controlIndex++)
{
    element = document.form1[controlIndex];
    if (element.type == "checkbox")
    {
        if (element.checked == true)
        {
            compSpec = compSpec + element.value + "\n";
        }
    }
}
alert(compSpec);
}
```

It's here that you loop through each element on the form using `document.form1[controlIndex]`, which returns a reference to the element object stored at the `controlIndex` index position.

You'll see that in this example the `element` variable is set to reference the object stored in the `form1[]` array at the index position stored in variable `controlIndex`. Again, this is for convenient shorthand purposes; now to use that particular object's properties or methods, you just type `element`, a period, and then the method or property name, making your code easier to read and debug, which also saves on typing.

You only want to see which check boxes have been checked, so you use the `type` property, which every HTML element object has, to see what element type you are dealing with. If the `type` is `checkbox`, you go ahead and see if it's a checked check box. If so, you append its value to the message string in `compSpec`. If it is not a check box, it can be safely ignored.

Finally, you use the `alert()` method to display the contents of your message string.

The select Elements

Although they look quite different, the drop-down list and the list boxes are actually both elements created with the `<select>` tag, and strictly speaking they are both select elements. The select element has one or more options in a list that you can select from; each of these options is defined by means of the `<option>` tag. Your list of `<option>` tags goes in between the `<select>` and `</select>` tags.

The `size` attribute of the `<select>` tag is used to specify how many of the options are visible to the user.

For example, to create a list box five rows deep and populate it with seven options, your `<select>` tag would look like this:

```
<select name=theDay size=5>
  <option value=0 selected>Monday
  <option value=1>Tuesday
  <option value=2>Wednesday
  <option value=3>Thursday
  <option value=4>Friday
  <option value=5>Saturday
  <option value=6>Sunday
</select>
```

Notice that the Monday `<option>` tag also contains the word `selected`; this will make this option the default selected one when the page is loaded. The values of the options have been defined as numbers, but text would be equally valid.

If you want this to be a drop-down list, you just need to change the `size` attribute in the `<select>` tag to 1, and presto, it's a drop-down list.

If you want to let the user choose more than one item from a list at once, you simply need to add the `multiple` attribute to the `<select>` definition.

The `<select>` tag creates a `Select` object. This object has an `options[]` array property, and this array is made up of `Option` objects, one for each `<option>` element inside the `<select>` element associated with the `Select` object. For instance, in the preceding example, if the `<select>` element was contained in a form called `theForm` with the following:

```
document.theForm.theDay.options[0]
```

you would access the option created for Monday.

How can you tell which option has been selected by the user? Easy: You use the `Select` object's `selectedIndex` property. You can use the index value returned by this property to access the selected option using the `options[]` array.

The `Option` object also has `index`, `text`, and `value` properties. The `index` property returns the index position of that option in the `options[]` array. The `text` property is what's displayed in the list, and the `value` property is the value defined for the option, which would be posted to the server if the form were submitted.

Chapter 6: HTML Forms — Interacting with the User

If you want to find out how many options there are in a select element, you can use the `length` property of either the `Select` object itself or of its `options[]` array property.

Let's see how you could loop through the `options[]` array for the preceding select box:

```
var theDayElement = window.document.form1.theDay;
document.write("There are " + theDayElement.length + " options<br>");
var optionCounter;
for (optionCounter = 0; optionCounter < theDayElement.length; optionCounter++)
{
    document.write("Option text is " + theDayElement.options[optionCounter].text)
    document.write(" and its value is ");
    document.write(theDayElement.options[optionCounter].value);
    document.write("<br>")
}
```

First you set the variable `theDayElement` to reference the `Select` object. Then you write the number of options to the page, in this case 7.

Next you use a `for` loop to loop through the `options[]` array, displaying the text of each option, such as *Monday*, *Tuesday*, and so on, and its value, such as 0, 1, and so on. If you create a page based on this code, it must be placed after the `<select>` tag has been defined.

It's also possible to add options to a select element after the page has finished loading. You'll look at how this is done next.

Adding New Options

To add a new option to a select element, you simply create a new `Option` object using the `new` operator and then insert it into the `options[]` array of the `Select` object at an empty index position.

When you create a new `Option` object, there are two parameters to pass—the first is the text you want to appear in the list, and the second the value to be assigned to the option.

```
var myNewOption = new Option("TheText", "TheValue");
```

You then simply assign this `Option` object to an empty array element, for example:

```
document.theForm.theSelectObject.options[0] = myNewOption;
```

If you want to remove an option, you simply set that part of the `options[]` array to `null`. For example, to remove the element you just inserted, you need the following:

```
document.theForm.theSelectObject.options[0] = null;
```

When you remove an `Option` object from the `options[]` array, the array is reordered so that the array index value of each of the options above the removed one has its index value decremented by one.

When you insert a new option at a certain index position, be aware that it will overwrite any `Option` object that is already there.

Try It Out Adding and Removing List Options

Use the list-of-days example you saw previously to demonstrate adding and removing list options.

```

<html>
<head>
<script language="JavaScript" type="text/javascript">
function butRemoveWed_onclick()
{
    if (document.form1.theDay.options[2].text == "Wednesday")
    {
        document.form1.theDay.options[2] = null;
    }
    else
    {
        alert('There is no Wednesday here!');
    }
}
function butAddWed_onclick()
{
    if (document.form1.theDay.options[2].text != "Wednesday")
    {
        var indexCounter;
        var days = document.form1.theDay;
        var lastoption = new Option();
        days.options[6] = lastoption;
        for (indexCounter = 6; indexCounter > 2; indexCounter--)
        {
            days.options[indexCounter].text = days.options[indexCounter - 1].text;
            days.options[indexCounter].value = days.options[indexCounter - 1].value;
        }
        var option = new Option("Wednesday",2);
        days.options[2] = option;
    }
    else
    {
        alert('Do you want to have TWO Wednesdays?????');
    }
}
</script>
</head>
<body>
<form name=form1>
<select name=theDay size=5>
    <option value=0 selected>Monday
    <option value=1>Tuesday
    <option value=2>Wednesday
    <option value=3>Thursday
    <option value=4>Friday
    <option value=5>Saturday
    <option value=6>Sunday
</select>
<BR>
<input type="button" value="Remove Wednesday" name=butRemoveWed

```

Chapter 6: HTML Forms — Interacting with the User

```
        onclick="butRemoveWed_onclick()" ">
<input type="button" value="Add Wednesday" name=butAddWed
        onclick="butAddWed_onclick()" ">
<BR>
</form>
</body>
</html>
```

Save this as `ch6_examp7.htm`. If you type the page in and load it into your browser, you should see the form shown in Figure 6-9. Click the Remove Wednesday button, and you'll see Wednesday disappear from the list. Add it back by clicking the Add Wednesday button. If you try to add a second Wednesday or remove a nonexistent Wednesday, you'll get a polite warning telling you that you can't do that.

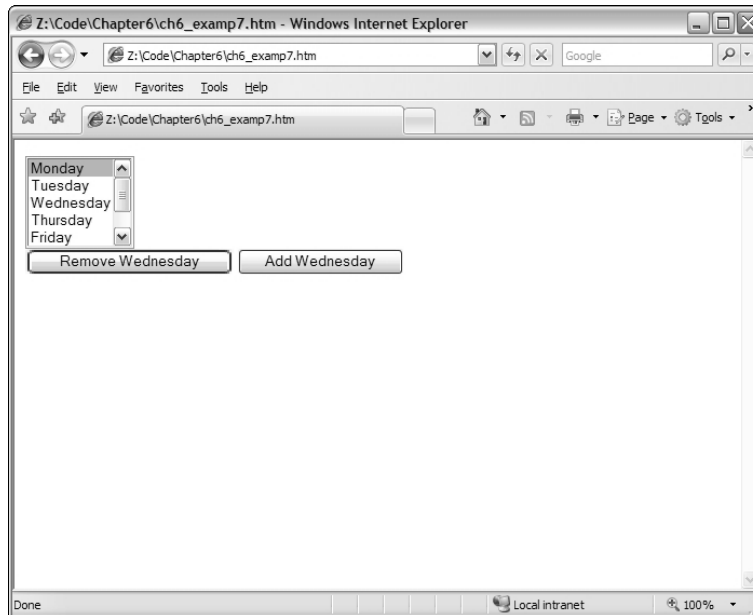


Figure 6-9

How It Works

Within the body of the page, you define a form with the name `form1`. This contains the select element, which includes day-of-the-week options that you have seen previously. The form also contains two buttons, as shown here:

```
<input type="button" value="Remove Wednesday" name=butRemoveWed
        onclick="butRemoveWed_onclick()" ">
<input type="button" value="Add Wednesday" name=butAddWed
        onclick="butAddWed_onclick()" ">
```

Each of these buttons has its `onclick` event handler connected to some code that calls one of two functions: `butRemoveWed_onclick()` and `butAddWed_onclick()`. These functions are defined in a script block in the head of the page. You'll take a look at each of them in turn.

At the top of the page you have your first function, `butRemoveWed_onclick()`, which removes the Wednesday option.

```
function butRemoveWed_onclick()
{
    if (document.form1.theDay.options[2].text == "Wednesday")
    {
        document.form1.theDay.options[2] = null;
    }
    else
    {
        alert('There is no Wednesday here!');
    }
}
```

The first thing you do in the function is a sanity check: You must try to remove the Wednesday option only if it's there in the first place! You make sure of this by seeing if the third option in the array (with index 2 because arrays start at index 0) has the text "Wednesday". If it does, you can remove the Wednesday option by setting that particular option to `null`. If the third option in the array is not Wednesday, you alert the user to the fact that there is no Wednesday to remove. Although this code uses the `text` property in the `if` statement's condition, you could just as easily have used the `value` property; it makes no difference.

Next you come to the `butAddWed_onclick()` function, which, as the name suggests, adds the Wednesday option. This is slightly more complex than the code required to remove an option. First you use an `if` statement to check that there is not already a Wednesday option.

```
function butAddWed_onclick()
{
    if (document.form1.theDay.options[2].text != "Wednesday")
    {
        var indexCounter;
        var days = document.form1.theDay;
        var lastoption = new Option();
        days.options[6] = lastoption;
        for (indexCounter = 6; indexCounter > 2; indexCounter--)
        {
            days.options[indexCounter].text = days.options[indexCounter - 1].text;
            days.options[indexCounter].value = days.options[indexCounter - 1].value;
        }
    }
}
```

If there is no Wednesday option, you then need to make space for the new Wednesday option to be inserted.

Before you do this, you define two variables, `indexCounter` and `days` (which refers to the `theDay` select element and is a shorthand reference for your convenience). Next you create a new option with the variable name `lastoption` and assign this new option to the element at index position 6 in your `options` array, which previously had no contents. You next assign the `text` and `value` properties of each of the `Option` objects from Thursday to Sunday to the `Option` at an index value higher by one in the `options` array, leaving a space in the `options` array at position 2 to put Wednesday in. This is the task for the `for` loop within the `if` statement.

Chapter 6: HTML Forms — Interacting with the User

Next, you create a new `Option` object by passing the text "Wednesday" and the value 2 to the `Option` constructor. The `Option` object is then inserted into the `options[]` array at position 2, and presto, it appears in your select box.

```
var option = new Option("Wednesday",2);
days.options[2] = option;
}
```

You end the function by alerting the user to the fact that there is already a Wednesday option in the list, if the condition in the `if` statement is `false`.

```
else
{
    alert('Do you want to have TWO Wednesdays?????');
}
}
```

Adding New Options with Internet Explorer

In IE, additional properties, methods, and events are associated with the select options. In particular, the `options[]` array you are interested in has the additional `add()` and `remove()` methods, which add and remove options. These make life a little simpler.

Before you add an option, you need to create it. You do this just as before, using the `new` operator.

The `add()` method enables you to insert an `Option` object that you have created and takes two parameters. You pass the option that you want to add as the first parameter. The optional second parameter enables you to specify which index position you want to add the option in. This parameter won't overwrite any `Option` object already at that position, but instead will simply move the `Option` objects up the array to make space. This is basically the same as what you had to code into the `butAddWed_onclick()` function using your `for` loop.

Using the `add()` method, you can rewrite the `butAddWed_onclick()` function in your `ch6_examp7.htm` example to look like this:

```
function butAddWed_onclick()
{
    if (document.form1.theDay.options[2].text != "Wednesday")
    {
        var option = new Option("Wednesday",2);
        document.form1.theDay.options.add(option,2);
    }
    else
    {
        alert('Do you want to have TWO Wednesdays?????');
    }
}
```

The `remove()` method takes just one parameter, namely the index of the option you want removed. When an option is removed, the options at higher index positions are moved down the array to fill the gap.

Using the `remove()` method, you can rewrite the `butRemoveWed_onclick()` function in your `ch6_examp7.htm` example to look like this:

```
function butRemoveWed_onclick()
{
    if (document.form1.theDay.options[2].text == "Wednesday")
    {
        document.form1.theDay.options.remove(2);
    }
    else
    {
        alert('There is no Wednesday here!');
    }
}
```

Modify the previous example and save it as `ch6_examp8_IE.htm` before loading it into IE. You'll see that it works just as the previous version did.

Select Element Events

Select elements have three event handlers, `onblur`, `onfocus`, and `onchange`. You've seen all these events before. You saw the `onchange` event with the text box element, where it fired when focus was moved away from the text box *and* the value in the text box had changed. Here it fires when the user changes which option in the list is selected.

Try It Out Using the Select Element for Date Difference Calculations

Let's take a look at an example that uses the `onchange` event and makes good use of the select element in its drop-down list form. Its purpose is to calculate the difference, in days, between two dates set by the user via drop-down list boxes.

```
<html>
<head>
<script language="JavaScript" type="text/javascript">
function writeOptions(startNumber, endNumber)
{
    var optionCounter;
    for (optionCounter = startNumber; optionCounter <= endNumber; optionCounter++)
    {
        document.write('<option value=' + optionCounter + '>' + optionCounter);
    }
}
function writeMonthOptions()
{
    var theMonth;
    var monthCounter;
    var theDate = new Date(1);
    for (monthCounter = 0; monthCounter < 12; monthCounter++)
    {
        theDate.setMonth(monthCounter);
        theMonth = theDate.toString();
        theMonth = theMonth.substr(4,3);
        document.write('<option value=' + theMonth + '>' + theMonth);
    }
}
```

```
    }  
}  
function recalcDateDiff()  
{  
    var myForm = document.form1;  
    var firstDay = myForm.firstDay.options[myForm.firstDay.selectedIndex].value;  
    var secondDay =  
        myForm.secondDay.options[myForm.secondDay.selectedIndex].value;  
    var firstMonth =  
        myForm.firstMonth.options[myForm.firstMonth.selectedIndex].value;  
    var secondMonth =  
        myForm.secondMonth.options[myForm.secondMonth.selectedIndex].value;  
    var firstYear =  
        myForm.firstYear.options[myForm.firstYear.selectedIndex].value;  
    var secondYear =  
        myForm.secondYear.options[myForm.secondYear.selectedIndex].value;  
    var firstDate = new Date(firstDay + " " + firstMonth + " " + firstYear);  
    var secondDate = new Date(secondDay + " " + secondMonth + " " + secondYear);  
    var daysDiff = (secondDate.valueOf() - firstDate.valueOf());  
    daysDiff = Math.floor(Math.abs((((daysDiff / 1000) / 60) / 60) / 24));  
    myForm.txtDays.value = daysDiff;  
    return true;  
}  
function window_onload()  
{  
    var theForm = document.form1;  
    var nowDate = new Date();  
    theForm.firstDay.options[nowDate.getDate() - 1].selected = true;  
    theForm.secondDay.options[nowDate.getDate() - 1].selected = true;  
    theForm.firstMonth.options[nowDate.getMonth()].selected = true;  
    theForm.secondMonth.options[nowDate.getMonth()].selected = true;  
    theForm.firstYear.options[nowDate.getFullYear() - 1970].selected = true;  
    theForm.secondYear.options[nowDate.getFullYear() - 1970].selected = true;  
}  
</script>  
</head>  
<body language=JavaScript onload="return window_onload()">  
<form name=form1>  
<p>  
First Date<br>  
<select name=firstDay size=1 onchange="return recalcDateDiff()">  
<script language=JavaScript>  
    writeOptions(1,31);  
</script>  
</select>  
<select name=firstMonth size=1 onchange="return recalcDateDiff()">  
<script language=JavaScript>  
    writeMonthOptions();  
</script>  
</select>  
<select name=firstYear size=1 onchange="return recalcDateDiff()">  
<script language=JavaScript>  
    writeOptions(1970,2020);  
</script>  
</select>
```

```
</p>
<p>
Second Date<br>
<select name=secondDay size=1 onchange="return recalcDateDiff()">
<script language=JavaScript>
    writeOptions(1,31);
</script>
</select>
<select name=secondMonth size=1 onchange="return recalcDateDiff()">
<script language=JavaScript>
    writeMonthOptions();
</script>
</select>
<select name=secondYear size=1 onchange="return recalcDateDiff()">
<script language=JavaScript>
    writeOptions(1970,2020);
</script>
</select>
</p>
Total difference in days
<input type="text" name=txtDays value=0 readonly>
<br>
</form>
</body>
</html>
```

Call the example `ch6_examp9.htm` and load it into your web browser. You should see the form shown in Figure 6-10, but with both date boxes set to the current date.

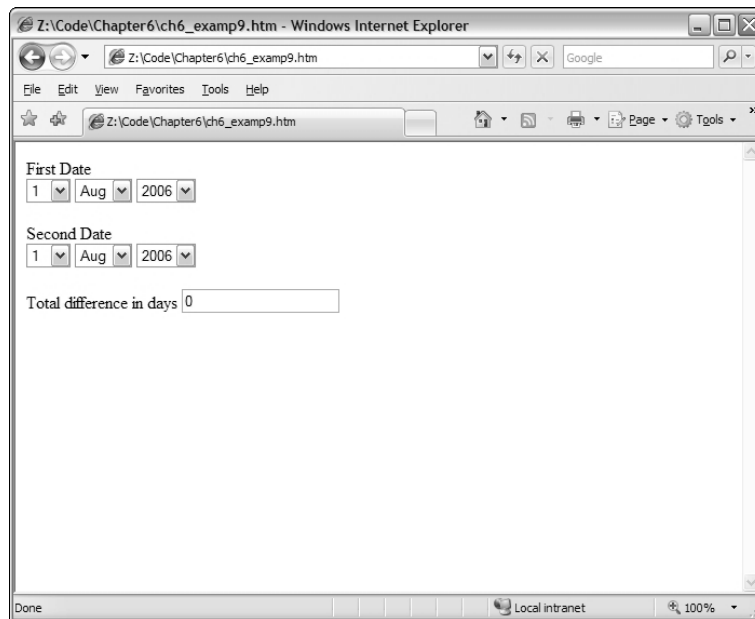


Figure 6-10

Chapter 6: HTML Forms — Interacting with the User

If you change any of the select boxes, the difference between the days will be recalculated and shown in the text box.

How It Works

In the body of the page, the form in the web page is built up with six drop-down list boxes and one text box. Let's look at an example of one of these select elements: take the first `<select>` tag, the one that allows the user to choose the day part of the first date.

```
<select name=firstDay size=1 onchange="return recalcDateDiff()">
<script language=JavaScript>
    writeOptions(1,31);
</script>
</select>
```

The `size` attribute has been set to 1 so that you have a drop-down list box rather than a list box, though strictly speaking 1 is the default, so you don't have to specify it. The `onchange` event handler has been connected to the `recalcDateDiff()` function that you'll be looking at shortly.

However, no `<option>` tags are defined within the `<select>` element. The drop-down list boxes need to be populated with too many options for you to enter them manually. Instead you populate the options using the functions, which make use of the `document.write()` method.

The date and year options are populated using the `writeOptions()` function declared in the head of the page. The function is passed two values: the start number and the end number of the options that you want the select element to be populated with. Let's look at the `writeOptions()` function.

```
function writeOptions(startNumber, endNumber)
{
    var optionCounter;
    for (optionCounter = startNumber; optionCounter <= endNumber; optionCounter++)
    {
        document.write('<option value=' + optionCounter + '>' + optionCounter);
    }
}
```

The function is actually quite simple, consisting of a `for` loop that loops from the first number (`startNumber`) through to the last (`endNumber`) using the variable `optionCounter`, and writes out the HTML necessary for each `<option>` tag. The text for the option and the value attribute of the `<option>` tag are specified to be the value of the variable `optionCounter`. It's certainly a lot quicker than typing out the 31 `<option>` tags necessary for the dates in a month.

For the year select box, the same function can be reused. You just pass 1970 and 2020 as parameters to the `writeOptions()` function to populate the year select box.

```
<select name=firstYear size=1 onchange="return recalcDateDiff()">
<script language=JavaScript>
    writeOptions(1970,2020);
</script>
</select>
```

To populate the month select box with the names of each month, you will need a different function. However, the principle behind populating the `<select>` element remains the same: You do it using `document.write()`. The function in this case is `writeMonthOptions()`, as you can see from the following month select element:

```
<select name=firstMonth size=1 onchange="return recalcdDateDiff()">
<script language=JavaScript>
    writeMonthOptions();
</script>
</select>
```

The new function, `writeMonthOptions()`, is defined in the head of the page. Let's take a look at it now. You start the function by defining three variables and initializing the variable, `theDate`, to the first day of the current month.

```
function writeMonthOptions()
{
    var theMonth;
    var monthCounter;
    var theDate = new Date(1);
```

You use the `Date` object you have stored to get the months as text (Jan, Feb...Dec). You get these months by setting the month in the `theDate` variable from 0 up to 11 using the `setMonth()` method in a `for` loop. Although the `Date` object does not provide a method for returning the date as anything other than a number, it does have the `toString()` method, which returns the value, as a string, of the date stored in the variable. It returns the date in the format of day of the week, month, day of the month, time, and finally year, for example, `Sat Feb 19 19:04:34 2000`. You just need the month part. Since you always know where it will be in the string and that its length is always 3, you can easily use the `String` object's `substr()` method to extract the month.

```
    for (monthCounter = 0; monthCounter < 12; monthCounter++)
    {
        theDate.setMonth(monthCounter);
        theMonth = theDate.toString();
        theMonth = theMonth.substr(4,3);
        document.write('<option value=' + theMonth + '>' + theMonth);
    }
}
```

Now that you have your month as a string of three characters, you can create the `<option>` tag and populate its text and value with the month.

For user convenience, it would be nice during the loading of the page to set both of the dates in the select elements to today's date. This is what you do in the `window_onload()` function, which is connected to the window's `onload` event by means of the `<body>` tag.

```
<body language="JavaScript" onload="return window_onload()">
```

The `window_onload()` function is defined in the head of the page. You start the function by setting the `theForm` variable to reference your `Form` object, because it shortens the reference needed in your code. Next you create a variable to hold a `Date` object to store today's date.

Chapter 6: HTML Forms — Interacting with the User

```
function window_onload()
{
    var theForm = document.form1;
    var nowDate = new Date();
```

Setting each of the `<select>` box's initial values is easy; the value returned by the `Date` object `nowDate` can be modified to provide the required index of the `options[]` array. For the day, the correct index is simply the day of the month minus one — remember that arrays start at 0, so day 1 is actually at index 0. The `selected` property is set to `true` to make that day the currently selected option in the list.

```
theForm.firstDay.options[nowDate.getDate() - 1].selected = true;
theForm.secondDay.options[nowDate.getDate() - 1].selected = true;
```

The month is even easier because the `getMonth()` function returns a value from 0 to 11 for the month, which exactly matches the necessary index value for our `options[]` array.

```
theForm.firstMonth.options[nowDate.getMonth()].selected = true;
theForm.secondMonth.options[nowDate.getMonth()].selected = true;
```

For the year, because you are starting with 1970 as your first year, you need to take 1970 from the current year to get the correct index value.

```
theForm.firstYear.options[nowDate.getFullYear() - 1970].selected = true;
theForm.secondYear.options[nowDate.getFullYear() - 1970].selected = true;
}
```

The final part of your code that you need to look at is the function connected to the `onchange` event of each select element, namely the `recalcDateDiff()` function. Your first task with this function is to build up the two dates the user has selected using the drop-down lists.

```
function recalcDateDiff()
{
    var myForm = document.form1;
    var firstDay = myForm.firstDay.options[myForm.firstDay.selectedIndex].value;
    var secondDay =
        myForm.secondDay.options[myForm.secondDay.selectedIndex].value;
    var firstMonth =
        myForm.firstMonth.options[myForm.firstMonth.selectedIndex].value;
    var secondMonth =
        myForm.secondMonth.options[myForm.secondMonth.selectedIndex].value;
    var firstYear =
        myForm.firstYear.options[myForm.firstYear.selectedIndex].value;
    var secondYear =
        myForm.secondYear.options[myForm.secondYear.selectedIndex].value;
```

You go through each select element and retrieve the value of the selected `Option` object. The `selectedIndex` property of the `Select` object provides the index you need to reference the selected `Option` object in the `options[]` array. For example, in the following line the index is provided by `myForm.firstDay.selectedIndex`:

```
var firstDay = myForm.firstDay.options[myForm.firstDay.selectedIndex].value;
```

You then use that value inside the square brackets as the index value for the `options[]` array of the `firstDay` select element. This provides the reference to the selected `Option` object, whose `value` property you store in the variable `firstDay`.

You use this technique for all the remaining select elements.

You can then create new `Date` objects based on the values obtained from the select elements and store them in the variables `firstDate` and `secondDate`.

```
var firstDate = new Date(firstDay + " " + firstMonth + " " + firstYear);
var secondDate = new Date(secondDay + " " + secondMonth + " " + secondYear);
```

Finally, you need to calculate the difference in days between the two dates.

```
var daysDiff = (secondDate.valueOf() - firstDate.valueOf());
daysDiff = Math.floor(Math.abs((((daysDiff / 1000) / 60) / 60) / 24));
```

The `Date` object has a method, `valueOf()`, which returns the number of milliseconds from the first of January, 1970, to the date stored in the `Date` object. You subtract the value of the `valueOf` property of `firstDate` from the value of the `valueOf` property of `secondDate` and store this in the variable `daysDiff`. At this point, it holds the difference between the two dates in milliseconds, so you convert this value to days in the following line. By dividing by 1,000 you make the value seconds, dividing the resulting number by 60 makes it minutes, by 60 again makes it hours, and finally you divide by 24 to convert to your final figure of difference in days. The `Math` object's `abs()` method makes negative numbers positive. The user may have set the first date to a later date than the second, and since you want to find only the difference between the two, not which is earlier, you make any negative results positive. The `Math.floor()` method removes the fractional part of any result and returns just the integer part rounded down to the nearest whole number.

Finally you write the difference in days to the `txtDays` text box in the page.

```
myForm.txtDays.value = daysDiff;
return true;
}
```

That completes our look at the more useful form elements available in web pages. The next section returns to the trivia quiz, where you can put your newfound knowledge to good use and actually create a working quiz page.

The Trivia Quiz

It's time to return to the trivia quiz as you left it in Chapter 3. So far you have defined the questions and answers in arrays, and defined a function to check whether the user's answer is correct. Now that you know how to create HTML forms and elements, you can start using them in the quiz to provide the user input. By the end of this section the question form will look like Figure 6-11.

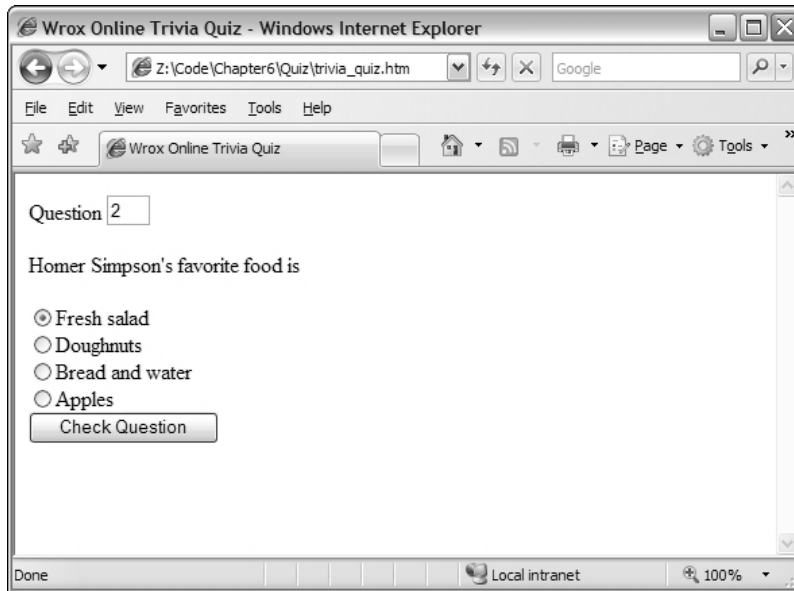


Figure 6-11

At present our questions are multiple-choice; you represent the multiple-choice options by a radio button group.

You create the form elements dynamically using our old friend `document.write()` and the information contained in the `questions` array. When the user has selected the radio button representing the answer, she then clicks the Check Question button, which calls your `checkAnswer()` function. This tells the user if she got the question right and lets her know. You then move on to the next question.

Let's start by creating the form elements.

Creating the Form

The first thing you need to do is add a form to your page in which the radio buttons will be written. Load `trivia_quiz.htm` and change the bottom of the page, below which the questions and answers arrays are defined, as follows:

```
// assign answer for question 3
answers[2] = "C";

</script>
</head>
<body>

<form name="QuestionForm">
  Question
  <input type="text" name=txtQNumber size=1>
  <script language="JavaScript" type="text/javascript">
```



```
document.write(getQuestion());
</script>
<input type="button" value="Check Question" name="buttonCheckQ"
  onclick="return buttonCheckQ_onclick()">
</form>
</body>
</html>
```

You're inserting the new form, named `QuestionForm`, inside the body of the page.

The elements on the form are a text box, defined by the following line:

```
<input type="text" name=txtQNumber size=1>
```

This will hold the current question number, and a button named `buttonCheckQ`.

```
<input type="button" value="Check Question" name="buttonCheckQ"
  onclick="return buttonCheckQ_onclick()">
```

When clicked, this will check the answer supplied by the user and let her know if she got it correct or not. The button has its `onclick` event connected to a function, `buttonCheckQ_onclick()`, which you'll create in a moment.

Where are the radio buttons you can see in Figure 6-11? Well, you'll be using the `document.write()` method again to dynamically insert the questions as the page is loaded. That way you can pick a random question each time from your question array. The following code inserts the question using the second function you need to add, `getQuestion()`:

```
<script language="JavaScript" type="text/javascript">
  document.write(getQuestion());
</script>
```

Creating the Answer Radio Buttons

You saw in the code that the radio buttons required will be inserted by the `getQuestion()` function, and that the `buttonCheckQ_onclick()` function is connected to the button's `onclick` event handler. You'll now add these functions to the top of the page in the same script block as the `answerCorrect()` function that you defined in Chapter 3.

Add the following lines to the top of the `trivia_quiz.htm` page:

```
<html>
<head>
<title>Wrox Online Trivia Quiz</title>
<script language="JavaScript" type="text/javascript">
var questionNumber;
function answerCorrect(questionNumber, answer)
{
  // declare a variable to hold return value
  var correct = false;
  // if answer provided is same as answer then correct answer is true
```

Chapter 6: HTML Forms — Interacting with the User

```
    if (answer == answers[questionNumber])
        correct = true;
    // return whether the answer was correct (true or false)
    return correct;
}

function getQuestion()
{
    questionNumber = Math.floor(Math.random() * (questions.length));
    var questionHTML = "<p>" + questions[questionNumber][0] + "</p>";
    var questionLength = questions[questionNumber].length;
    var questionChoice;
    for (questionChoice = 1; questionChoice < questionLength; questionChoice++)
    {
        questionHTML = questionHTML + "<input type=radio name=radQuestionChoice"
        if (questionChoice == 1)
        {
            questionHTML = questionHTML + " checked";
        }
        questionHTML = questionHTML + ">";
        questionHTML = questionHTML + questions[questionNumber][questionChoice];
        questionHTML = questionHTML + "<br>";
    }
    document.QuestionForm.txtQNumber.value = questionNumber + 1;
    return questionHTML;
}

function buttonCheckQ_onclick()
{
    var answer = 0;
    while (document.QuestionForm.radQuestionChoice[answer].checked != true)
    {
        answer++;
    }
    answer = String.fromCharCode(65 + answer);
    if (answerCorrect(questionNumber, answer) == true)
    {
        alert("You got it right");
    }
    else
    {
        alert("You got it wrong");
    }
    window.location.reload();
}
```

// questions and answers arrays will hold questions and answers

You will discuss the `getQuestion()` function first, which is used to build up the HTML needed to display the question to the user. You first want to select a random question from your `questions` array, so you need to generate a random number, which will provide the index for the question. You store this number in the global variable `questionNumber` that you declared at the top of the script block.

```
function getQuestion()
{
    questionNumber = Math.floor(Math.random() * (questions.length));
```

You generate a random number between 0 and 1 using the `Math.random()` method, and then multiply that by the number of questions in the `questions` array. This number is converted to an integer using the `Math` object's `floor()` method, which returns the lowest integer part of a floating-point number. This is exactly what you want here: a randomly selected number from 0 to `questions.length` minus one. Don't forget that arrays start at an index of 0.

Your next task is to create the radio buttons, which allow the user to answer the question. You do this by building up the HTML that needs to be written to the page inside the variable `questionHTML`. You can then display the question using just one `document.write()`, which writes the whole question out in one go.

You start this process by declaring the `questionHTML` variable and setting it to the HTML needed to write the actual question to the page. This information is stored in the first index position of the second dimension of your `questions` array—that is, `questions[questionNumber][0]`, where `questionNumber` is the random index you generated before.

```
var questionHTML = "<p>" + questions[questionNumber][0] + "</p>";
var questionLength = questions[questionNumber].length;
var questionChoice;
```

To create the possible answers for the user to select from, you need to know how many radio buttons are required, information that's stored in the `length` property of the second dimension of your `questions` array. Remember that the second dimension is really just an `Array` object stored in a particular position of your `questions` array and that `Array` objects have a `length` property. You use the variable `questionLength` to store the length of the array and also to declare another variable, `questionChoice`, which you will use to loop through your array.

Now you can start looping through the question options and build up the radio button group. You do this in the next `for` loop. If it's the first radio button that you are creating the HTML for, you add the `checked` word to the `<input>` tag. You do this to ensure that one of the radio buttons is checked, just in case the user tries to press the Check Answer button without actually providing one first.

```
for (questionChoice = 1; questionChoice < questionLength; questionChoice++)
{
    questionHTML = questionHTML + "<input type=radio name=radQuestionChoice"
    if (questionChoice == 1)
    {
        questionHTML = questionHTML + " checked";
    }
    questionHTML = questionHTML + ">";
    questionHTML = questionHTML + questions[questionNumber][questionChoice];
    questionHTML = questionHTML + "<br>";
}
```

For example, on one loop of the `for` loop, the HTML built up in `questionHTML` may be the following:

```
<input type=radio name=radQuestionChoice checked> A sixties rock group from
Liverpool<br>
```

With the looping finished and `questionHTML` containing the complete HTML needed to display one question, all that remains to do is to display the question number for the current question in the text box

Chapter 6: HTML Forms — Interacting with the User

in the form, and then return the `questionHTML` string to the calling code. You use `questionNumber + 1` as the question number purely for user friendliness. Even though it might be a question at index 0, most people think of starting at question 1 not question 0.

```
document.QuestionForm.txtQNumber.value = questionNumber + 1;
return questionHTML;
}
```

That completes the `getQuestion()` function. The final new code you need to look at is the `buttonCheckQ_onclick()` function that fires when the button is clicked. You saw this added to your code earlier.

You start the function by declaring the variable `answer` and initializing it to 0. You'll be using this as the index when looping through the radio button group and also to hold the actual answer.

```
function buttonCheckQ_onclick()
{
    var answer = 0;
```

You then use a `while` statement to loop through each of the radio buttons, incrementing the `answer` variable until it hits upon a radio button that is checked. At this point the loop ends and you now know which radio button the user chose as his answer, namely the one at the index stored in the `answer` variable.

```
while (document.QuestionForm.radQuestionChoice[answer].checked != true)

{
    answer++;
}
```

Since your `answers` array holds the answers as A, B, C, D, and so on, you need to convert the radio button index contained in `answer` into a character. You do this in the next line.

```
answer = String.fromCharCode(65 + answer);
```

This makes use of the fact that the character code for A is 65, so that if the user chooses the first radio button—the one with an index of 0—you just need to add 65 and the index number contained in `answer` to get the answer's character code. This code is converted to a character by means of the `String` object's `fromCharCode()` method. Remember that some methods of the `String` object, called *static methods*, can be used without having to actually create a `String` object; you can use the native `String` object, which is always present.

The `answerCorrect()` function you created in Chapter 3 is then used as part of an `if` statement. You pass the question number and the answer character to the function, and it returns `true` if the answer is correct. If it does return `true`, you show a message box telling the user that he got the question right; otherwise the `else` statement lets him know that he got it wrong.

```
if (answerCorrect(questionNumber, answer) == true)
{
    alert("You got it right");
}
```

```
else
{
    alert("You got it wrong");
}
```

Finally, you reload the page to select another random question.

```
window.location.reload();
}
```

In the next chapter you'll be making the Trivia Quiz a more sophisticated multi-frame-based application, also adding necessary features like one to make sure that the user doesn't get the same question twice.

Summary

In this chapter you looked at how to add a user interface onto your JavaScript so that you can interact with your users and acquire information from them. Let's look at some of the things we discussed in this chapter.

- ❑ The HTML form is where you place elements making up the interface in a page.
- ❑ Each HTML form groups together a set of HTML elements. When a form is submitted to a server for processing, all the data in that form are sent to the server. You can have multiple forms on a page, but only the information in one form can be sent to the server.
- ❑ A form is created with the opening tag `<form>` and ends with the close tag `</form>`. All the elements you want included in that form are placed in between the open and close `<form>` tags. The `<form>` tag has various attributes—for client-side scripting, the `name` attribute is the important one. You can access forms with either their `name` attribute or their `ID` attribute.
- ❑ Each `<form>` element creates a `Form` object, which is contained within the `document` object. To access a form named `myForm`, you write `document.myForm`. The `document` object also has a `forms[]` property, which is an array containing every form inside the document. The first form in the page is `document.forms[0]`, the second is `document.forms[1]`, and so on. Using the `length` property of an Array object, `document.forms.length` tells you how many forms are on the page.
- ❑ Having discussed forms, we then went on to look at the different types of HTML elements that can be placed inside forms, how to create them, and how they are used in JavaScript.
- ❑ The objects associated with the form elements have a number of properties, methods, and events that are common to them all. They all have the `name` property, which you can use to reference them in your JavaScript. They also all have the `form` property, which provides a reference to the `Form` object in which that element is contained. The `type` property returns a text string telling you what type of element this is; types include `text`, `button`, and `radio`.
- ❑ You also saw that the methods `focus()` and `blur()`, and the events `onfocus` and `onblur`, are available to every form element object. Such an element is said to receive the focus when it becomes the active element in the form, either because the user has selected that element or because you used the `focus()` method. However an element got the focus, its `onfocus` event will fire. When another element is set as the currently active element, the previous element is

said to lose its focus, or to blur. Again, loss of focus can be the result of the user selecting another element or the use of the `blur()` method; either way, when it happens the `onblur` event fires. You saw that the firing of `onfocus` and `onblur` can, if used carefully, be a good place to check things like the validity of data entered by a user into an element.

- ❑ All elements return a value, which is the string data assigned to that element. The meaning of the value depends on the element; for a text box, it is the value inside the text box, and for a button, it's the text displayed on its face.
- ❑ Having discussed the common features of elements, we then looked at each of the more commonly used elements in turn, starting with the button element.
- ❑ The button element's purpose in life is to be clicked by the user, where that clicking fires some script you have written. You can capture the clicking by connecting to the button's `onclick` event. A button is created by means of the `<input>` tag with the `type` attribute set to `button`. The `value` attribute determines what text appears on the button's face. Two variations on a button are the `submit` and `reset` buttons. In addition to acting as buttons, they also provide a special service not linked to code. The `submit` button will automatically submit the form to the server; the `reset` button clears the form back to its default state when loaded in the page.
- ❑ The text element allows the user to enter a single line of plain text. A text box is created by means of the `<input>` tag with the `type` attribute set to `text`. You can set how many characters the user can enter and how wide the text box is with the `maxlength` and `size` attributes, respectively, of the `<input>` tag. The text box has an associated object called `Text`, which has the additional events `onselect` and `onchange`. The `onselect` event fires when the user selects text in the box, and the more useful `onchange` event fires when the element loses focus and its contents have changed since the element gained the focus. The firing of the `onchange` event is a good place to do validation of what the user has just entered. If she entered illegal values, such as letters when you wanted numbers, you can let the user know and send her back to correct her mistake. A variation on the text box is the password box, which is almost identical to the text box except that the values typed into it are hidden and shown as an asterisk. Additionally, the text box also has the `onkeydown`, `onkeypress`, and `onkeyup` events.
- ❑ The next element you looked at was the text area, which is similar to the text box except that it allows multiple lines of text to be entered. This element is created with the open tag `<textarea>` and closed with the `</textarea>` tag, the width and height in characters of the text box being determined by the `cols` and `rows` attributes respectively. The `wrap` attribute determines whether the text area wraps text that reaches the end of a line and whether that wrapping is sent when the contents are posted to the server. If this attribute is left out, or set to `off`, no wrapping occurs; if set to `soft`, it causes wrapping client-side, but is not sent to the server when the form is sent; if set to `hard`, it causes wrapping client-side and is sent to the server. The associated `Textarea` object has virtually the same properties, methods, and events as a `Text` object.
- ❑ You then looked at the check box and radio button elements together. Essentially they are the same type of element, except that the radio button is a grouped element, meaning that only one in a group can be checked at once. Checking another one causes the previously checked button to be unchecked. Both elements are created with the `<input>` tag, the `type` attribute being `checkbox` or `radio`. If `checked` is put inside the `<input>` tag, that element will be checked when the page is loaded. Creating radio buttons with the same name creates a radio button group. The name of a radio button actually refers to an array, and each element within that array is a radio button defined on the form to be within that group. These elements have associ-

ated objects called `Checkbox` and `Radio`. Using the `checked` property of these objects, you can find out whether a check box or radio button is currently checked. Both objects also have the `onclick` event in addition to the common events `onfocus` and `onblur`.

- ❑ Next in your look at elements were the drop-down list and list boxes. Both in fact are actually the same `select` element, with the `size` attribute determining whether it's a drop-down or list box. The `<select>` tag creates these elements, the `size` attribute determining how many list items are visible at once. If a `size` of 1 is given, a drop-down box rather than a list box is created. Each item in a `select` element is defined by the `<option>` tag, or added to later by means of the `Select` object's `options[]` array property, which is an array containing each `Option` object for that element. However, adding options after the page is loaded is different for Firefox and Microsoft browsers. The `Select` object's `selectedIndex` property tells you which option is selected; you can then use that value to access the appropriate option in the `options[]` array and use the `Option` object's `value` property. The `Option` object also has the `text` and `index` properties, `text` being the displayed text in the list and `index` being its position in the `Select` object's `options[]` array property. You can loop through the `options[]` array, finding out its length from the `Select` object's `length` property. The `Select` object has the `onchange` event, which fires when the user selects another item from the list.
- ❑ Finally, you added a basic user interface to the trivia quiz. Now questions are created dynamically with the `document.write()` method and the user can select his answer from a group of radio buttons.

In the next chapter you'll look at how, once you have created a frameset in a page, you can access code and variables between frames. You'll also look at how to open new windows using JavaScript, and methods of manipulating them when they are open. You'll see the trivia quiz become a frame-based application.

Exercises

Suggested solutions to these questions can be found in Appendix A.

Question 1

Using the code from the temperature converter example you saw in Chapter 2, create a user interface for it and connect it to the existing code so that the user can enter a value in degrees Fahrenheit and convert it to centigrade.

Question2

Create a user interface that allows the user to pick the computer system of her dreams, similar in principle to the e-commerce sites selling computers over the Internet. For example, she could be given a choice of processor type, speed, memory, and hard drive size, and the option to add additional components like a DVD-ROM drive, a sound card, and so on. As the user changes her selections, the price of the system should update automatically and notify her of the cost of the system as she has specified it, either by using an alert box or by updating the contents of a text box.

7

Windows and Frames

Until now, the pages you have been looking at have just been single pages. However, many web applications make use of frames to split up the browser's window, much as panes of glass split up a real window. It's quite possible that you'll want to build web sites that make use of such frames. The good news is that JavaScript enables the manipulation of frames and allows functions and variables you create in one frame to be used from another frame. One advantage of this is that you can keep common variables and functions in one place but use them from many places. You start this chapter by looking at how you can script across such frames.

But a number of other good reasons exist for wanting to access variables and functions in another frame. Two important reasons are to make your code *modular* and to gain the ability to maintain information between pages.

What do we mean by *modular*? In other programming languages, like Visual Basic, you can create a module—an area to hold general functions and variables—and reuse it from different places in your program. Well, when using frames you can put all of your general functions and variables into one area, such as the top frame, which you can think of as your code module. Then you can call the functions repeatedly from different pages and different frames.

If you put the general functions and variables in a page that defines the frames that it contains (that is, a frameset-defining page), then if you need to make changes to the pages inside the frames, any variables defined in the frameset page will retain their value. This provides a very useful means of holding information even when the user is navigating your web site. A further advantage is that any functions defined in the frameset-defining page can be called by subsequent pages and have to be loaded into the browser only once, making your page's loading faster.

The second subject of this chapter is how you can open up and manipulate new browser windows. There are plenty of good uses for new windows. For example, you may wish to open up an "external" web site in a new window from your web site, but still leave your web site open for the user. *External* here means a web site created and maintained by another person or company. Let's say you have a web site about cars—well, you may wish to have a link to external sites, such as manufacturing web sites (for example, that of Ford or General Motors). Perhaps even more useful is using small windows as dialog boxes, which you can use to obtain information from the user. Just as you can script between frames, you can do similar things between certain windows. You find out how later in the chapter, but let's start by looking at scripting between frames.

Frames and the window Object

Frames are a means of splitting up the browser window into various panes, into which you can then load different HTML documents. The frames are defined in a frameset-defining page by the `<frameset>` and `<frame>` tags. The `<frameset>` tag is used to contain the `<frame>` tags and specifies how the frames should look on the page. The `<frame>` tags are then used to specify each frame and to include the required documents in the page.

You saw in Chapter 5 that the `window` object represents the browser's frame on your page or document. If you have a page with no frames, there will be just one `window` object. However, if you have more than one frame, there will be one `window` object for each frame. Except for the very top-level window of a frameset, each `window` object is contained inside another.

The easiest way to demonstrate this is through an example in which you create three frames, a top frame with two frames inside it.

Try It Out Multiple Frames

For this multi-frame example, you'll need to create three HTML files. The first is the frameset-defining page.

```
<html>
<frameset rows="50%,*" ID="TopWindow">
<frame name="UpperWindow" src="UpperWindow.htm">
<frame name="LowerWindow" src="LowerWindow.htm">
</frameset>
</html>
```

Save this as `TopWindow.htm`. Note that the `src` attributes for the two `<frame>` tags in this page are `UpperWindow.htm` and `LowerWindow.htm`. You will create these next.

```
<html>
<head>
<script language="JavaScript" type="text/javascript">
function window_onload()
{
alert("The name of the upper frame's window object is " + window.name);
alert("The location of UpperWindow's parent is " +
window.parent.location.href);
}
</script>
</head>
<body onload="return window_onload()">
<p>Upper Frame</p>
</body>
</html>
```

The preceding page is the source page for the top frame with the name `UpperWindow` and needs to be saved as `UpperWindow.htm`. The final page is very similar to it:

```

<html>
<head>
<script language="JavaScript" type="text/javascript">
function window_onload()
{
alert("The name of the lower frame's window object is " + window.name);
alert("The location of LowerWindow's parent is " +
window.parent.location.href);
}
</script>
</head>
<body onload="return window_onload()">
<p>Lower Frame</p>
</body>
</html>

```

This is the source page for the lower frame; save it as `LowerWindow.htm`.

These three pages fit together so that `UpperWindow.htm` and `LowerWindow.htm` are contained within the `TopWindow.htm` page.

When you load them into the browser, you have three window objects. One is the *parent* window object and contains the file `TopWindow.htm`, and two are *child* window objects, containing the files `UpperWindow.htm` and `LowerWindow.htm`. The two child window objects are contained within the parent window, as shown in Figure 7-1.

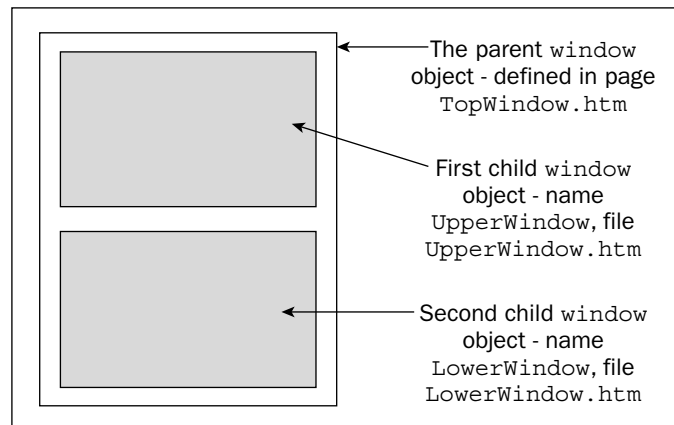


Figure 7-1

If any of the frames had frames contained inside them, these would have `window` objects that were children of the `window` object of that frame.

When you load `TopWindow.htm` into your browser, you'll see a series of four message boxes, as shown in Figures 7-2 through 7-5. These are making use of the `window` object's properties to gain information and demonstrate the `window` object's place in the hierarchy.

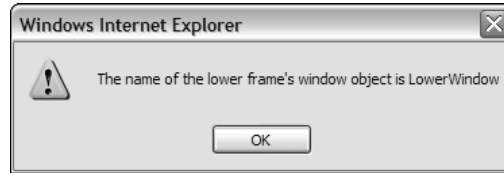


Figure 7-2

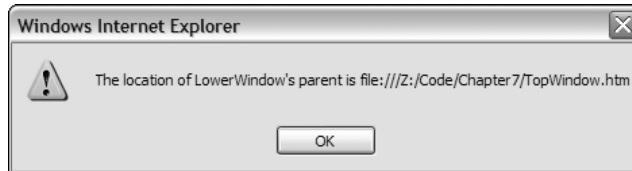


Figure 7-3

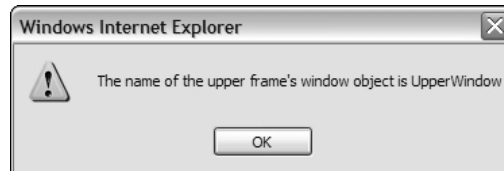


Figure 7-4

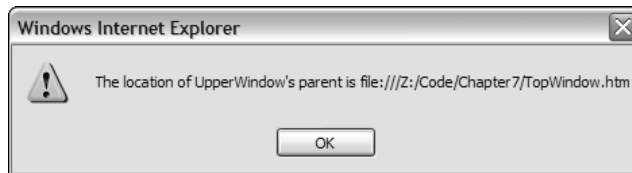


Figure 7-5

How It Works

Look at the frameset-defining page, starting with `TopWindow.htm`, as shown in the following snippet:

```
<html>
<frameset rows="50%,*" ID=TopWindow>
<frame name="UpperWindow" src="UpperWindow.htm">
<frame name="LowerWindow" src="LowerWindow.htm">
</frameset>
</html>
```

The frameset is defined with the `<frameset>` tag. You use two attributes: `rows` and `ID`. The `rows` attribute takes the value `"50%,*"` meaning that the first frame should take up half of the length of the window, and the second frame should take up the rest of the room. The `ID` attribute is used to give a name that you can use to reference the page.

The two child windows are created using `<frame>` tags. In each of the `<frame>` tags, you specify a name by which the window objects will be known and the `src` attribute of the page that will be loaded into the newly created windows and will form the basis of the `document` object that each window object contains.

Let's take a look at the `UpperWindow.htm` file next. In the `<body>` tag of the page, you attach a function, `window_onload()`, to the window object's `onload` event handler. This event handler is called when the browser has finished loading the window, the document inside the window, and all the objects within the document. It's a very useful place to put initialization code or code that needs to change things after the page has loaded but before control passes back to the user.

```
<body onload="return window_onload()">
```

This function is defined in a script block in the head of the page as follows:

```
function window_onload()
{
    alert("The name of the upper frame's window object is " + window.name);
    alert("The location of UpperWindow's parent is " +
        window.parent.location.href);
}
```

The `window_onload()` function makes use of two properties of the `window` object for the frame that the page is loaded in: its `name` and `parent` properties. The `name` property is self-explanatory—it's the name you defined in the frameset page. In this case, the name is `UpperWindow`.

The second property, the `parent` property, is very useful. It gives you access to the `window` object of the frame's parent. This means you can access all of the parent window object's properties and methods. Through these, you can access the document within the parent window as well as any other frames defined by the parent. Here, you display a message box giving details of the parent frame's file name or URL by using the `href` property of the `location` object (which itself is a property of the `window` object).

The code for `LowerWindow.htm` is identical to the code for `UpperWindow.htm`, but with different results because you are accessing a different window object. The name of the window object this time is `LowerWindow`. However, it shares the same parent window as `UpperWindow`, and so when you access the `parent` property of the window object, you get a reference to the same window object as in `UpperWindow`. The message box demonstrates this by displaying the file name/URL or `href` property, and this matches the file name of the page displayed in the `UpperWindow` frame.

Please note that the order of display of messages may vary among different types of browsers and even different operating systems. This may not be important here, but there will be times when the order in which events fire is important and affects the working of your code. It's an incompatibility that's worth noting and watching out for in your own programs.

Coding Between Frames

You've seen that each frame exists as a different window and gets its own `window` object. In addition, you saw that you can access the `window` object of a frameset-defining page from any of the frame pages it specifies, by using the `window` object's `parent` property. When you have a reference to the parent window's `window` object, you can access its properties and methods in the same way that you access the `window` object of the current page. In addition, you have access to all the JavaScript variables and functions defined in that page.

Try It Out Using the Frameset Page as a Module

Let's look at a more complex example, wherein you use the top frame to keep track of pages as the user navigates the web site. You're creating five pages in this example, but don't panic; four of them are almost identical. The first page that needs to be created is the frameset-defining page.

```
<html>
<head>
<title>The Unchanging frameset page</title>
<script Language="JavaScript" type="text/javascript">
var pagesVisited = new Array();
function returnPagesVisited()
{
var returnValue = "So far you have visited the following pages\n";
var pageVisitedIndex;
var numberOfPagesVisited = pagesVisited.length;
for (pageVisitedIndex = 0; pageVisitedIndex < numberOfPagesVisited;
pageVisitedIndex++)
{
returnValue = returnValue + pagesVisited[pageVisitedIndex] + "\n";
}
return returnValue;
}
function addPage(fileName)
{
var fileNameStart = fileName.lastIndexOf("/") + 1;
fileName = fileName.substr (fileNameStart);
pagesVisited[pagesVisited.length] = fileName;
return true;
}
</script>
</head>
<frameset cols="50%,*">
<frame name=fraLeft src="page_a.htm">
<frame name=fraRight src="page_b.htm">
</frameset>
</html>
```

Save this page as `frameset_page.htm`.

Notice that the two frames have the `src` attributes initialized as `page_a.htm` and `page_b.htm`. However, you also need to create `page_c.htm` and `page_d.htm` because you will be allowing the user to choose the page loaded into each frame from these four pages. You'll create the `page_a.htm` page first, as shown in the following:

```
<html>
<head>
<script language="JavaScript" type="text/javascript">
function butShowVisited_onclick()
{
document.form1.txtaPagesVisited.value = window.parent.returnPagesVisited();
}
</script>
</head>
```

```

<body onload="window.parent.addPage(window.location.href);">
<center>
<font size=6 color=MidnightBlue face=verdana>
This is Page A
</font>
</center>
<p>
<A href="page_a.htm">Page A</A>
<A href="page_b.htm">Page B</A>
<A href="page_c.htm">Page C</A>
<A href="page_d.htm">Page D</A>
</p>
<form name=form1>
<textarea rows=10 cols=35 name=txtaPagesVisited wrap=hard>
</textarea>
<br>
<input type="button" value="List Pages Visited" name=butShowVisited
onclick="butShowVisited_onclick()">
</form>
</body>
</html>

```

Save this page as `page_a.htm`.

The other three pages are identical to `page_a.htm`, except for one line, so you can just cut and paste the text from `page_a.htm`. Change the HTML that displays the name of the page loaded to the following:

```

<center>
<font size=6 color=MidnightBlue face=verdana>
This is Page B
</font>
</center>

```

Then save this as `page_b.htm`.

Do the same again, to create the third page (page C):

```

<center>
<font size=6 color=MidnightBlue face=verdana>
This is Page C
</font>
</center>

```

Save this as `page_c.htm`.

The final page is again a copy of `page_a.htm` except for the following lines:

```

<center>
<font size=6 color=MidnightBlue face=verdana>
This is Page D
</font>
</center>

```

Save this as `page_d.htm`.

Chapter 7: Windows and Frames

Load `FramesetPage.htm` into your browser and navigate to various pages by clicking the links. Then click the List Pages Visited button in the left-hand frame, and you should see a screen similar to the one shown in Figure 7-6.

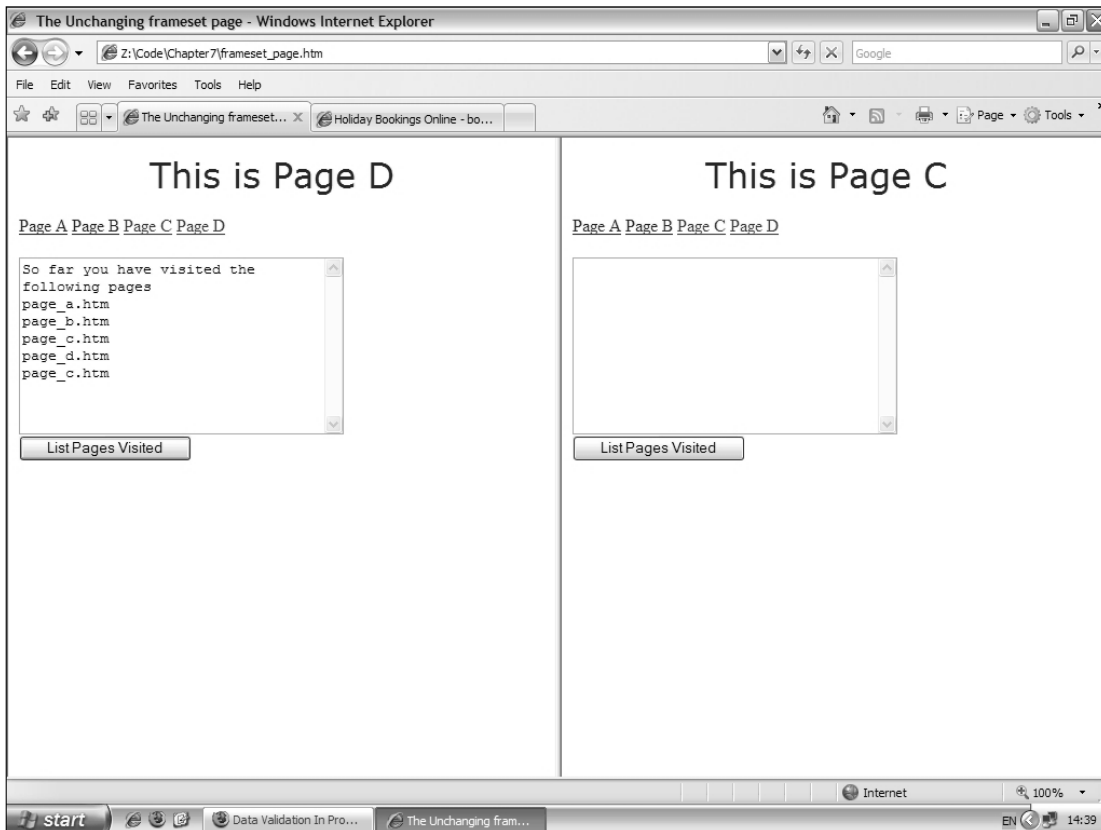


Figure 7-6

Click the links in either frame to navigate to a new location. For example, click the Page C link in the right frame, then the Page D link in the left frame. Click the left frame's List Pages Visited button and you'll see that `page_c.htm` and `page_d.htm` have been added to the list.

Normally when a new page is loaded, any variables and their values in the previous page are lost, but with framesets it does not matter which page is loaded into each frame — the top frame remains loaded and its variables keep their values. What you are seeing in this example is that, regardless of which page is loaded in each frame, some global variable in the top frame is keeping track of the pages that have been viewed and the top frame's variables and functions can be accessed by any page loaded into either frame.

You'll see later that there are restrictions when the pages you load into the frames are from external sources — more on this later in the chapter.

How It Works

Let's first look at the JavaScript in `frameset_page.htm`, which is the frameset-defining page. The head of the page contains a script block. The first thing you do in this script block is declare the variable `pagesVisited` and set it to reference a new `Array` object. In the array, you'll be storing the file name of each page visited as the user navigates the site.

```
var pagesVisited = new Array();
```

You then have two functions. The first of the two functions, `returnPagesVisited()`, does what its name suggests—it returns a string containing a message and a list of each of the pages visited. It does this by looping through the `pagesVisited` array, building up the message string inside the variable `returnValue`, which is then returned to the calling function.

```
function returnPagesVisited()
{
    var returnValue = "So far you have visited the following pages\n";
    var pageVisitedIndex;
    var numberOfPagesVisited = pagesVisited.length;
    for (pageVisitedIndex = 0; pageVisitedIndex < numberOfPagesVisited;
        pageVisitedIndex++)
    {
        returnValue = returnValue + pagesVisited[pageVisitedIndex] + "\n";
    }
    return returnValue;
}
```

The second function, `addPage()`, adds the name of a page to the `pagesVisited` array.

```
function addPage(fileName)
{
    var fileNameStart = fileName.lastIndexOf("/") + 1;
    fileName = fileName.substr(fileNameStart);
    pagesVisited[pagesVisited.length] = fileName;
    return true;
}
```

The `fileName` parameter passed to this function is the full file name and path of the visited page, so you need to strip out the path to get just the file name. The format of the string will be something like `file:///D:/myDirectory/page_b.htm`, and you need just the bit after the last `/` character. So in the first line of code, you find the position of that character and add one to it because you want to start at the next character.

Then, using the `substr()` method of the `String` object in the following line, you extract everything from character position `fileNameStart` right up to the end of the string. Remember that the `substr()` method takes two parameters, namely the starting character you want and the length of the string you want to extract, but if the second parameter is missing, all characters from the start position to the end are extracted.

You then add the file name into the array, the `length` property of the array providing the next free index position.

Chapter 7: Windows and Frames

You'll now turn to look collectively at the frame pages, namely `page_a.htm`, `page_b.htm`, `page_c.htm`, and `page_d.htm`. In each of these pages, you create a form called `form1`.

```
<form name=form1>
<textarea rows=10 cols=35 name=txtaPagesVisited wrap=hard>
</textarea>
<br>
<input type="button" value="List Pages Visited" name=butShowVisited
onclick="butShowVisited_onclick()" ">
</form>
```

This contains the `textarea` control that will display the list of pages visited, and a button the user can click to populate the `textarea`.

When one of these pages is loaded, its name is put into the `pagesVisited` array defined in `frameset_page.htm` by the window object's `onload` event handler's being connected to the `addPage()` function that you also created in `frameset_page.htm`. You connect the code to the event handler in the `<body>` tag of the page as follows:

```
<body onload="window.parent.addPage(window.location.href);">
```

Recall that all the functions you declare in a page are contained, like everything else in a page, inside the `window` object for that page, but that because the `window` object is the global object, you don't need to prefix the name of your variables or functions with `window`.

However, this time the function is not in the current page, but in the `frameset_page.htm` page. The `window` containing this page is the parent window to the window containing the current page. You need, therefore, to refer to the parent frame's `window` object using the `window` object's `parent` property. The code `window.parent` gives you a reference to the `window` object of `frameset_page.htm`. With this reference, you can now access the variables and functions contained in `frameset_page.htm`. Having stated which `window` object you are referencing, you just add the name of the function you are calling, in this instance the `addPage()` function. You pass this function the `location.href` string, which contains the full path and file name of the page, as the value for its one parameter.

As you saw earlier, the button on the page has its `onclick` event handler connected to a function called `butShowVisited_onclick()`. This is defined in the head of the page.

```
function butShowVisited_onclick()
{
document.form1.txtaPagesVisited.value = window.parent.returnPagesVisited();
}
```

In this function you call the parent `window` object's `returnPagesVisited()` function, which, as you saw earlier, returns a string containing a list of pages visited. The `value` property of the `textarea` object is set to this text.

That completes your look at the code in the frame pages, and as you can see, there's not much of it because you have placed all the general functions in the `frameset` page. Not only does this code reuse make for less typing, but it also means that all your functions are in one place. If there is a bug in a function, fixing the

bug for one page also fixes it for all other pages that use the function. Of course, it only makes sense to put general functions in one place; functions that are specific to a page and are never used again outside it are best kept in that page.

Code Access Between Frames

You've just seen how a child window can access its parent window's variables and functions, but how can frames inside a frameset access each other?

You saw a simple example earlier in this chapter, so this time let's look at a much more complex example. When created, your page will look like the one shown in Figure 7-7.

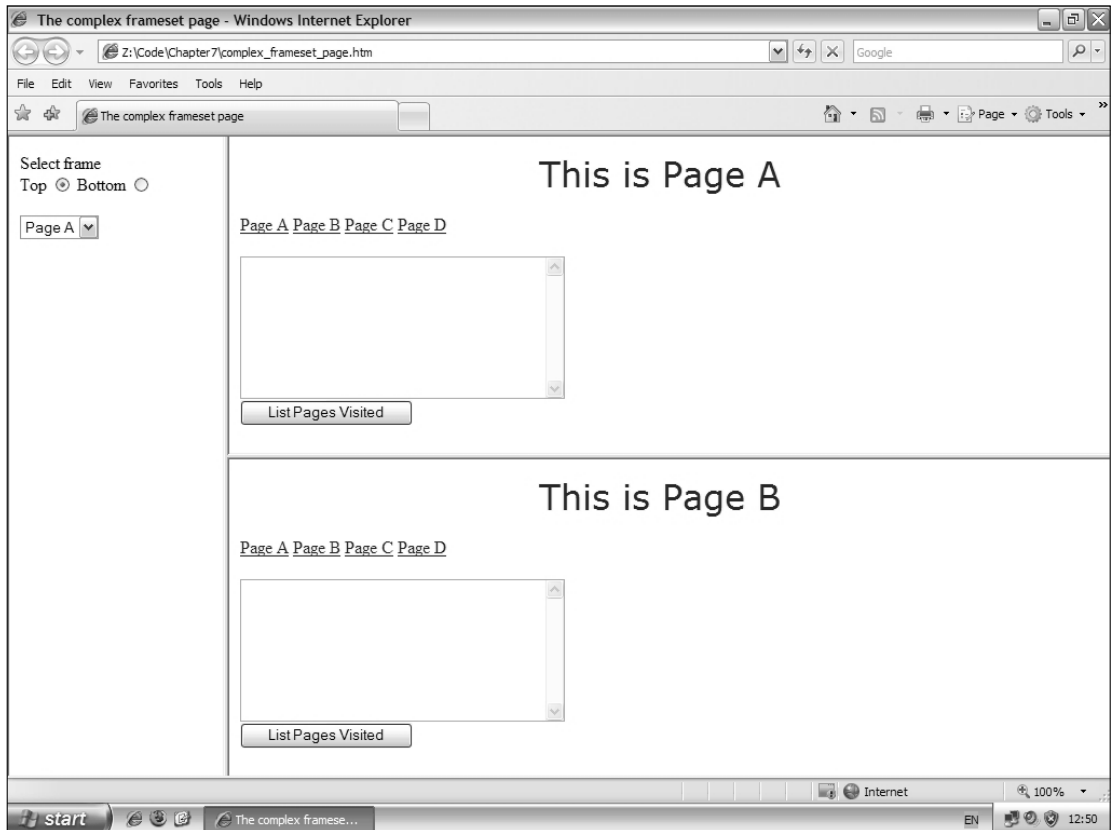


Figure 7-7

A diagram of the frame layout is shown in Figure 7-8. The text labels indicate the names that each frame has been given in the `<frameset>` and `<frame>` tags, with the exception of the top frame, which is simply the window at the top of the frameset hierarchy.

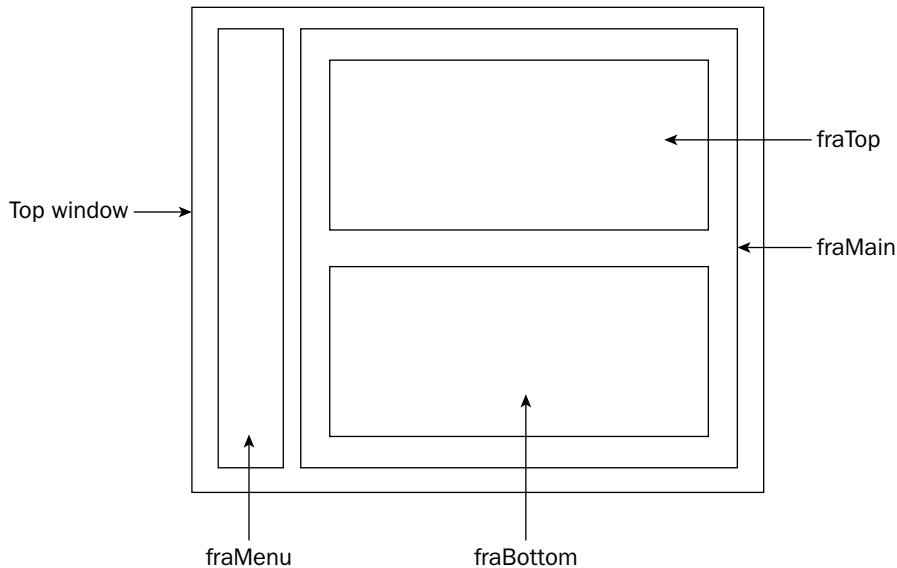


Figure 7-8

The easiest way to think of the hierarchy of such a frames-based web page is in terms of familial relationships, which can be shown in a family tree. If you represent your frameset like that, it looks something like the diagram in Figure 7-9.

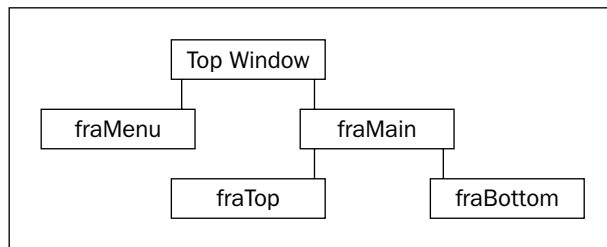


Figure 7-9

From the diagram you can see that `fraBottom`, the right-hand frame's bottom frame, has a parent frame called `fraMain`, which itself has a parent, the top window. Therefore, if you wanted to access a function in the top window from the `fraBottom` window, you would need to access `fraBottom`'s parent's parent's window object. You know that the window object has the `parent` property, which is a reference to the parent window of that window object. So let's use that and create the code to access a function, for example, called `myFunction()`, in the top window.

```
window.parent.parent.myFunction();
```

Let's break this down. The following code gets you a reference to the parent window object of the window in which the code is running.

```
window.parent
```

The code is in `fraBottom`, so `window.parent` will be `fraMain`. However, you want the top window, which is `fraMain`'s parent, so you add to the preceding code to make this:

```
window.parent.parent
```

Now you have a reference to the top window. Finally, you call `myFunction()` by adding that to the end of the expression.

```
window.parent.parent.myFunction();
```

What if you want to access the window object of `fraMenu` from code in `fraBottom`? Well, you have most of the code you need already. You saw that `window.parent.parent` gives you the top window, so now you want that window's child window object called `fraMenu`. You can get it in three ways, all with identical results.

You can use its index in the `frames[]` array property of the window object as follows:

```
window.parent.parent.frames[0]
```

Alternatively, you can use its name in the `frames[]` array like this:

```
window.parent.parent.frames["fraMenu"]
```

Finally, you can reference it directly by using its name as you can with any window object:

```
window.parent.parent.fraMenu
```

The third method is the easiest unless you don't know the name of a frame and need to access it by its index value in the `frames[]` array, or are looping through each child frame in turn.

Since `window.parent.parent.fraMenu` gets you a reference to the window object associated with `fraMenu`, to access a function `myFunction()` or variable `myVariable`, you would just type one of these lines:

```
window.parent.parent.fraMenu.myFunction
```

or

```
window.parent.parent.fraMenu.myVariable
```

What if you want to access not a function or variable in a page within a frame, but a control on a form or even the links on that page? Well, let's imagine you want to access, from the `fraBottom` page, a control named `myControl`, on a form called `myForm` in the `fraMenu` page.

You found that `window.parent.parent.fraMenu` gives you the reference to `fraMenu`'s window object from `fraBottom`, but how do you reference a form there?

Basically, it's the same as how you access a form from the inside of the same page as the script, except that you need to reference not the window object of that page but the window object of `fraMenu`, the page you're interested in.

Chapter 7: Windows and Frames

Normally you write `document.myForm.myControl.value`, with `window` being assumed since it is the global object. Strictly speaking, it's `window.document.myForm.myControl.value`.

Now that you're accessing another window, you just reference the window you want and then use the same code. So you need this code if you want to access the `value` property of `myControl` from `fraBottom`:

```
window.parent.parent.fraMenu.document.myForm.myControl.value
```

As you can see, references to other frames can get pretty long, and in this situation it's a very good idea to store the reference in a variable. For example, if you are accessing `myForm` a number of times, you could write this:

```
var myFormRef = window.parent.parent.fraMenu.document.myForm;
```

Having done that, you can now write

```
myFormRef.myControl.value;
```

rather than

```
window.parent.parent.fraMenu.document.myForm.myControl.value;
```

The top Property

Using the `parent` property can get a little tedious when you want to access the very top window from a frame quite low down in the hierarchy of frames and `window` objects. An alternative is the `window` object's `top` property. This returns a reference to the `window` object of the very top window in a frame hierarchy. In the current example, this is `top.window`.

For instance, in the example you just saw, this code:

```
window.parent.parent.fraMenu.document.myForm.myControl.value;
```

could be written like this:

```
window.top.fraMenu.document.myForm.myControl.value;
```

Although, because the `window` is a global object, you could shorten that to just this:

```
top.fraMenu.document.myForm.myControl.value;
```

So when should you use `top` rather than `parent`, or vice versa?

Both properties have advantages and disadvantages. The `parent` property enables you to specify `window` objects relative to the current window. The window above this window is `window.parent`, its parent is `window.parent.parent`, and so on. The `top` property is much more generic; `top` is always the very top window regardless of the frameset layout being used. There will always be a `top`, but there's not necessarily going to always be a `parent.parent`. If you put all your global functions and variables that you want accessible from any page in the frameset in the very top window, `window.top`

will always be valid regardless of changes to framesets beneath it, whereas the `parent` property is dependent on the frameset structure above it. However, if someone else loads your web site inside a frameset page of his own, then suddenly the `top` window is not yours but his, and `window.top` is no longer valid. You can't win, or can you?

One trick is to check to see whether the `top` window contains your page; if it doesn't, reload the `top` page again and specify that your `top` page is the one to be loaded. For example, check to see that the file name of the `top` page actually matches the name you expect. The `window.top.location.href` will give you the name and path—if they don't match what you want, use `window.top.location.replace("myPagename.htm")` to load the correct `top` page. However, as you'll see later, this will cause problems if someone else is loading your page into a frameset she has created—this is where something called the *same-origin policy* applies. More on this later in the chapter.

Try It Out Scripting Frames

Let's put all you've learned about frames and scripting into an example based on the frameset you've been looking at. You're going to be reusing a lot of the pages and code from the previous example in this chapter.

The first page you're creating is the `top` window page.

```
<html>
<head>
<title>The complex frameset page</title>
<script language="JavaScript" type="text/javascript">
var pagesVisited = new Array();
function returnPagesVisited()
{
var returnValue = "So far you have visited the following pages\n";
var pageVisitedIndex;
var numberOfPagesVisited = pagesVisited.length;
for (pageVisitedIndex = 0; pageVisitedIndex < numberOfPagesVisited;
pageVisitedIndex++)
{
returnValue = returnValue + pagesVisited[pageVisitedIndex] + "\n";
}
return returnValue;
}
function addPage(fileName)
{
var fileNameStart = fileName.lastIndexOf("/") + 1;
fileName = fileName.substr(fileNameStart);
pagesVisited[pagesVisited.length] = fileName;
return true;
}
</script>
</head>
<frameset cols="200,*">
<frame name=fraMenu src="menu_page.htm">
<frame name=fraMain src="main_page.htm">
</frameset>
</html>
```

Chapter 7: Windows and Frames

As you can see, you've reused a lot of the code from `frameset_page.htm`, so you can cut and paste the script block from there. Only the different code lines are highlighted. Save this page as `complex_frameset_page.htm`.

Next, create the page that will be loaded into `fraMenu`, namely `menu_page.htm`.

```
<html>
<head>
<script language="JavaScript" type="text/javascript">
function choosePage_onchange()
{
var choosePage = document.form1.choosePage;
var windowobject;
if (document.form1.radFrame[0].checked == true)
{
windowobject = window.parent.fraMain.fraTop;
}
else
{
windowobject = window.parent.fraMain.fraBottom;
}
windowobject.location.href =
choosePage.options[choosePage.selectedIndex].value;
return true;
}
</script>
</head>
<body>
<form name=form1>
Select frame<br>
Top <input name="radFrame" checked type=radio>
Bottom <input name="radFrame" type=radio>
<br><br>
<select name=choosePage language="JavaScript" type="text/javascript"
onchange="choosePage_onchange()">
<option value=page_a.htm>Page A
<option value=page_b.htm>Page B
<option value=page_c.htm>Page C
<option value=page_d.htm>Page D
</select>
</form>
</body>
</html>
```

Save this as `menu_page.htm`.

The `fraMain` frame contains a page that is simply a frameset for the `fraTop` and `fraBottom` pages.

```
<html>
<frameset rows="50%,*">
<frame name=fraTop src="page_a.htm">
<frame name=fraBottom src="page_b.htm">
</frameset>
</html>
```


Save this as `main_page.htm`.

For the next four pages you reuse the four pages—`page_a.htm`, `page_b.htm`, `page_c.htm`, and `page_d.htm`—from the first example. You'll need to make a few changes, as shown in the following code. (Again, all the pages are identical except for the text shown in the page, so only `page_a.htm` is shown. Amend the rest in a similar way.)

```
<html>
<head>
<script language="JavaScript" type="text/javascript">
function butShowVisited_onclick()
{
document.form1.txtaPagesVisited.value = window.top.returnPagesVisited();
}
function setFrameAndPageControls(linkIndex)
{
var formobject = window.parent.parent.fraMenu.document.form1;
formobject.choosePage.selectedIndex = linkIndex;
if (window.parent.fraTop == window.self)
{
formobject.radFrame[0].checked = true;
}
else
{
formobject.radFrame[1].checked = true;
}
}
return true;
}
</script>
</head>
<body
onload="return window.top.addPage(window.location.href);">
<center>
<font size=6 color=MidnightBlue face=verdana>
This is Page A
</font>
</center>
<p>
<A href="page_a.htm" name="pageALink"
onclick="return setFrameAndPageControls(0)">Page A</A>
<A href="page_b.htm" name="pageBLink"
onclick="return setFrameAndPageControls(1)">Page B</A>
<A href="page_c.htm" name="pageCLink"
onclick="return setFrameAndPageControls(2)">Page C</A>
<A href="page_d.htm" name="pageDLink"
onclick="return setFrameAndPageControls(3)">Page D</A>
</p>
<form name=form1>
<textarea rows=8 cols=35 name=txtaPagesVisited>
</textarea>
<br>
<input type="button" value="List Pages Visited" name=butShowVisited
onclick="butShowVisited_onclick()">
</form>
</body>
</html>
```

Chapter 7: Windows and Frames

Resave the pages under their old names.

Load `complex_frameset_page.htm` into your browser, and you'll see a screen similar to the one shown in Figure 7-7.

The radio buttons allow the user to determine which frame he wants to navigate to a new page. When he changes the currently selected page in the drop-down list, that page is loaded into the frame selected by the radio buttons.

If you navigate using the links in the pages inside the `fraTop` and `fraBottom` frames, you'll notice that the selected frame radio buttons and the drop-down list in `fraMenu` on the left will be automatically updated to the page and frame just navigated to. Note that as the example stands, if the user loads `page_a.htm` into a frame the select list doesn't allow it to load the same page in the other frame. You could improve on this example by adding a button that loads the currently selected page into the chosen frame.

The List Pages Visited buttons display a list of visited pages, as they did in the previous example.

How It Works

You've already seen how the code defining the top window in `complex_frameset_page.htm` works, as it is very similar to the previous example. However, you'll just look quickly at the `<frameset>` tags where, as you can see, the names of the windows are defined in the names of the `<frame>` tags.

```
<frameset cols="200,*">
<frame name=fraMenu src="menu_page.htm">
<frame name=fraMain src="main_page.htm">
</frameset>
```

Notice also that the `cols` attribute of the `<frameset>` tag is set to `"200,*"`. This means that the first frame will occupy a column 200 pixels wide, and the other frame will occupy a column taking up the remaining space.

Let's look in more detail at the `fraMenu` frame containing `menu_page.htm`. At the top of the page, you have your main script block. This contains the function `choosePage_onchange()`, which is connected to the `onchange` event handler of the select box lower down on the page. The select box has options containing the various page URLs.

The function starts by defining two variables. One of these, `choosePage`, is a shortcut reference to the `choosePage Select` object further down the page.

```
function choosePage_onchange()
{
  var choosePage = document.form1.choosePage;
  var windowobject;
```

The `if...else` statement then sets your variable `windowobject` to reference the window object of whichever frame the user has chosen in the `radFrame` radio button group.

```
if (document.form1.radFrame[0].checked == true)
{
  windowobject = window.parent.fraMain.fraTop;
```

```

    }
    else
    {
        windowobject = window.parent.fraMain.fraBottom;
    }

```

As you saw earlier, it's just a matter of following through the references, so `window.parent` gets you a reference to the parent window object. In this case, `window.top` would have done the same thing. Then `window.parent.fraMain` gets you a reference to the window object of the `fraMain` frame. Finally, depending on which frame you want to navigate in, you reference the `fraTop` or `fraBottom` window objects contained within `fraMain`, using `window.parent.fraMain.fraTop` or `window.parent.fraMain.fraBottom`.

Now that you have a reference to the window object of the frame in which you want to navigate, you can go ahead and change its `location.href` property to the value of the selected drop-down list item, causing the frame to load that page.

```

windowobject.location.href =
choosePage.options[choosePage.selectedIndex].value;
return true;
}

```

As you saw before, `main_page.htm` is simply a frameset-defining page for `fraTop` and `fraBottom`. Let's now look at the pages you're actually loading into `fraTop` and `fraBottom`. Because they are all the same, you'll look only at `page_a.htm`.

Let's start by looking at the top script block. This contains two functions, `butShowVisited_onclick()` and `setFrameAndPageControls()`. You saw the function `butShowVisited_onclick()` in the previous example.

```

function butShowVisited_onclick()
{
    document.form1.txtaPagesVisited.value = window.top.returnPagesVisited();
}

```

However, because the frameset layout has changed, you do need to change the code. Whereas previously the `returnPagesVisited()` function was in the parent window, it's now moved to the top window. As you can see, all you need to do is change the reference from `window.parent.returnPagesVisited()` to `window.top.returnPagesVisited()`.

As it happens, in the previous example the parent window was also the top window, so if you had written your code in this way in the first place, there would have been no need for changes here. It's often quite a good idea to keep all your general functions in the top frameset page. That way all your references can be `window.top`, even if the frameset layout is later changed.

The new function in this page is `setFrameAndPageControls()`, which is connected to the `onclick` event handler of the links defined lower down on the page. This function's purpose is to make sure that if the user navigates to a different page using the links rather than the controls in the `fraMenu` window, those controls will be updated to reflect what the user has done.

Chapter 7: Windows and Frames

The first thing you do is set the `formobject` variable to reference the `form1` in the `fraMenu` page, as follows:

```
function setFrameAndPageControls(linkIndex)
{
  var formobject = window.parent.parent.fraMenu.document.form1;
```

Let's break this down.

```
window.parent
```

gets you a reference to the `fraMain` window object. Moving up the hierarchy, you use the following code to get a reference to the window object of the top window:

```
window.parent.parent
```

Yes, you're right. You could have used `window.top` instead, and this would have been a better way to do it. We're doing it the long way here just to demonstrate how the hierarchy works.

Now you move down the hierarchy, but on the other side of your tree diagram, to reference the `fraMenu`'s window object.

```
window.parent.parent.fraMenu
```

Finally, you are interested only in the form and its controls, so you reference that object like this:

```
window.parent.parent.fraMenu.document.form1
```

Now that you have a reference to the form, you can use it just as you would if this were code in `fraMenu` itself.

The function's parameter `linkIndex` tells you which of the four links was clicked, and you use this value in the next line of the function's code to set which of the options is selected in the drop-down list box on `fraMenu`'s form.

```
formobject.choosePage.selectedIndex = linkIndex;
```

The `if...else` statement is where you set the `fraMenu`'s radio button group `radFrame` to the frame the user just clicked on, but how can you tell which frame this is?

```
if (window.parent.fraTop == window.self)
{
  formobject.radFrame[0].checked = true
}
else
{
  formobject.radFrame[1].checked = true
}
```

You check to see whether the current window object is the same as the window object for `fraTop`. You do this using the `self` property of the window object, which returns a reference to the current window

object, and `window.parent.fraTop`, which returns a reference to `fraTop`'s window object. If one is equal to the other, you know that they are the same thing and that the current window is `fraTop`. If that's the case, the `radFrame` radio group in the `fraMenu` frame has its first radio button checked. Otherwise, you check the other radio button for `fraBottom`.

The last thing you do in the function is return `true`. Remember that this function is connected to an `A` object, so returning `false` cancels the link's action, and `true` allows it to continue, which is what you want.

```
return true;
}
```

Opening New Windows

So far in this chapter, you have been looking at frames and scripting between them. In this section, you'll change direction slightly and look at how you can open up additional browser windows.

Why would you want to bother opening up new windows? Well, they can be useful in all sorts of different situations, such as the following:

- ❑ You might want a page of links to web sites, in which clicking a link opens up a new window with that web site in it.
- ❑ Additional windows can be useful for displaying information. For example, if you had a page with products on it, the user could click a product image to bring up a new small window listing the details of that product. This can be less intrusive than navigating the existing window to a new page with product details, and then requiring the user to click Back to return to the list of products. You'll be creating an example demonstrating this later in this chapter.
- ❑ Dialog windows can be very useful for obtaining information from users, although overuse may annoy them.

Something you do need to be aware of is that some users have programs installed to block pop-up windows, and the latest versions of Firefox and Internet Explorer also enable the user to switch off the ability to open new windows. By default, new windows created automatically when a page loads are usually blocked. However, windows that open only when the user must perform an action, for example clicking a link or button, are not normally blocked by default, but the user may change the browser settings to block them.

Opening a New Browser Window

The window object has an `open()` method, which opens up a new window. It takes three parameters, although the third is optional, and it returns a reference to the window object of the new browser window.

The first parameter of the `open()` method is the URL of the page that you want to have opened in the new window. However, if you want, you can pass an empty string for this parameter and get a blank page, and then use the `document.write()` method to insert HTML into the new window dynamically. You'll see an example of this later in the chapter.

Chapter 7: Windows and Frames

The second parameter is the name you want to allocate to the new window. This is not the name you use for scripting, but instead is used for the `target` attribute of things such as hyperlinks and forms. For example, if you set this parameter to `myWindow` and set a hyperlink on the original page to

```
<A href="test3.htm" TARGET=myWindow>Test3.htm</A>
```

then clicking that hyperlink will cause the hyperlink to act on the new window opened. This means that `test3.htm` will be loaded into the new window and not the current window. The same applies to the `<form>` tag's `target` attribute, which you'll be looking at in the chapters on server-side JavaScript. In this case, if a form is submitted from the original window, the response from the server can be made to appear in the new window.

When a new window is opened, it is opened (by default) with a certain set of properties, such as `width` and `height`, and with the normal browser-window features. Browser-window features include things such as a location entry field and a menu bar with navigation buttons.

The third parameter of the `open()` method can be used to specify values for the `height` and `width` properties. Also, because by default most of the browser window's features are switched off, you can switch them back on using the third parameter of the `open()` method. You'll look at browser features in more detail shortly.

Let's first look at an example of the code you need to open a basic window. You'll name this window `myWindow` and give it a `width` and `height` of 250 pixels. You want the new window to open with the `test2.htm` page inside.

```
var newWindow;  
newWindow = window.open("test2.htm", "myWindow", "width=250,height=250");
```

You can see that `test2.htm` has been passed as the first parameter; that is the URL of the page you want to open. You've named the window `myWindow` in the second parameter. In the third parameter, you've set the `width` and `height` properties to 250.

You'll also notice that you've set the variable `newWindow` to the return value returned by the `open()` method, which is a reference to the `window` object of the new window just opened. You can now use `newWindow` to manipulate the new window and gain access to the `document` contained inside it using the `window.document` property. You can do everything with this reference that you did when dealing with frames and their `window` objects. For example, if you wanted to change the background color of the `document` contained inside the new window, you would type this:

```
newWindow.document.bgColor = "RED";
```

How would you close the window you just opened? Easy, you just use the `window` object's `close()` method like this:

```
newWindow.close();
```

Try It Out Opening Up New Windows

Let's look at the example mentioned earlier of a products page in which clicking a product brings up a window listing the details of that product. In a shameless plug, you'll be using a couple of Wrox books

as examples — though with just two products on your page, it's not exactly the world's most extensive online catalog.

```
<html>
<head>
<title>Online Books</title>
<script language="JavaScript" type="text/javascript">
var detailsWindow;
function showDetails(bookURL)
{
detailsWindow = window.open(bookURL,"bookDetails","width=400,height=350");
detailsWindow.focus();
return false;
}
</script>
</head>
<body>
<h2 align=center>Online Book Buyer</h2>

<p>
Click any of the images below for more details
</p>
<strong>Professional Active Server Pages .Net</strong>
<br>

<br><br>
<strong>Beginning Dreamweaver MX 2004</strong>
<br>

</body>
</html>
```

Save this page as `online_books.htm`. You'll also need to create two images and name them `pro_asp.jpg` and `beg_dreamweaver.jpg`. Alternatively, you can find these files in the code download. Note that if the user has disabled JavaScript, the window won't be opened. You can get around this by adding the page details to the `href` attribute as follows:

```
<a href="pro_asp_details.htm"
onclick="return showDetails(this.href);"><img ...></a>
```

You now need to create the two details pages, both plain HTML.

```
<html>
<head>
<title>Professional ASP.NET 2.0</title>
</head>
<body>
<strong>Professional ASP.NET 2.0</strong>
<br>
Subjects
<br>
ASP
```

Chapter 7: Windows and Frames

```
<br>
Internet
<br>
<HR color=#cc3333>
<p><strong>Book overview</strong> </p>
<p>This comprehensive compendium provides a broad and thorough
investigation of all aspects of programming with ASP.NET. Entirely revised
and updated for the 2.0 Release of .NET, this book will give you the information
you need to master ASP.NET and build dynamic, successful, enterprise Web
applications.
</p>
</body>
</html>
```

Save this as `pro_asp_details.htm`.

```
<html>
<head>
<title>Beginning Dreamweaver MX 2004</title>
</head>
<body>
<strong>Beginning Dreamweaver MX 2004</strong>
<br>
<strong>Subjects</strong>
Dreamweaver<br>
Internet<br>
Web Design<br>
XML and Scripting<br>
<HR color=#cc3333>
<p><strong>Book overview</strong></p>
<p>With this book you'll quickly be creating powerful,
dynamic web sites with Dreamweaver MX 2004 - Macromedia's powerful, integrated web
development and editing tool.
You'll learn how to use the tools and features of
Dreamweaver to construct three complete sites using
HTML, JavaScript, Active Server Pages (ASP), and databases.
</p>
</body>
</html>
```

Save the final page as `beg_dreamweaver_details.htm`.

Load `online_books.htm` into your browser and click either of the two images. A new window containing the book's details should appear above the existing browser window. Click the other book image, and the window will be replaced by one containing the details of that book.

How It Works

The files `pro_asp_details.htm` and `beg_dreamweaver_details.htm` are both plain HTML files, so you won't look at them. However, in `online_books.htm` you find some scripting action, which you *will* look at here.

In the script block at the top of the page, you first define the variable `detailsWindow`.

```
var detailsWindow;
```

You then have the function that actually opens the new windows.

```
function showDetails(bookURL)
{
    detailsWindow = window.open(bookURL, "bookDetails", "width=400,height=350");
    detailsWindow.focus();
    return false;
}
```

This function is connected to the `onclick` event handlers of book images that appear later in the page. The parameter `bookURL` is passed by the code in the `onclick` event handler and will be either `beg_asp3_details.htm` or `prof_js_details.htm`.

You create the new window with the `window.open()` method. You pass the `bookURL` parameter as the URL to be opened. You pass `bookDetails` as the name you want applied to the new window. If the window already exists, another new window won't be opened, and the existing one will be navigated to the URL that you pass. This only occurs because you are using the same name (`bookDetails`) when opening the window for each book. If you had used a different name, a new window would be opened.

By storing the reference to the window object just created in the variable `detailsWindow`, you can access its methods and properties. On the next line, you'll see that you use the `window` object, referenced by `detailsWindow`, to set the focus to the new window—otherwise it will appear behind the existing window if you click the same image in the main window more than once.

Although you are using the same function for each of the image's `onclick` event handlers, you pass a different parameter for each, namely the URL of the details page for the book in question.

```
<strong>Professional Active Server Pages .Net</strong>
<br>

<br><br>
<strong>Beginning Dreamweaver MX 2004</strong>
<br>

</A>
```

Adding HTML to a New Window

You learned earlier that you can pass an empty string as the first parameter of the `window` object's `open()` method and then write to the page using HTML. Let's see how you would do that.

First, you need to open a blank window by passing an empty value to the first parameter that specifies the file name to load.

```
var newWindow = window.open("", "myNewWindow", "width=150,height=150");
```

Chapter 7: Windows and Frames

Now you can open the window’s document to receive your HTML.

```
newWindow.document.open();
```

This is not essential when a new window is opened, because the page is blank; but with a document that already contains HTML, it has the effect of clearing out all existing HTML and blanking the page, making it ready for writing.

Now you can write out any valid HTML using the `document.write()` method.

```
newWindow.document.write("<h4>Hello</h4>");
newWindow.document.write("<p>Welcome to my new little window</p>");
```

Each time you use the `write()` method, the text is added to what’s already there until you use the `document.close()` method.

```
newWindow.document.close();
```

If you then use the `document.write()` method again, the text passed will replace existing HTML rather than adding to it.

Adding Features to Your Windows

As you have seen, the `window.open()` method takes three parameters, and it’s the third of these parameters that you’ll be looking at in this section. Using this third parameter, you can control things such as the size of the new window created, its start position on the screen, whether the user can resize it, whether it has a toolbar, and so on.

Features such as menu bar, status bar, and toolbar can be switched on or off with `yes` or `1` for on and `no` or `0` for off. You can also switch these features on by including their names without specifying a value.

The list of possible options shown in the following table is not complete, and not all of them work with both IE and Firefox browsers.

Window Feature	Possible Values	Description
copyHistory	yes, no	Copy the history of the window doing the opening to the new window
directories	yes, no	Show directory buttons
height	integer	Height of new window in pixels
left	integer	Window’s left starting position in pixels
location	yes, no	Show location text field
menubar	yes, no	Show menu bar
resizable	yes, no	Enable the user to resize the window after it has been opened

Window Feature	Possible Values	Description
scrollbars	yes, no	Show scrollbars if the page is too large to fit in the window
status	yes, no	Show status bar
toolbar	yes, no	Show toolbar
top	integer	Window's top starting position in pixels
width	integer	Width of new window in pixels

As mentioned earlier, this third parameter is optional. If you don't include it, all the window features default to *yes*, except the window's size and position properties, which default to preset values. For example, if you try the following code, you'll see a window something like the one shown in Figure 7-10:

```
<html>
<head>
<script language="JavaScript" type="text/javascript">
var newWindow;
newWindow = window.open("", "myWindow");
</script>
</head>
</html>
```

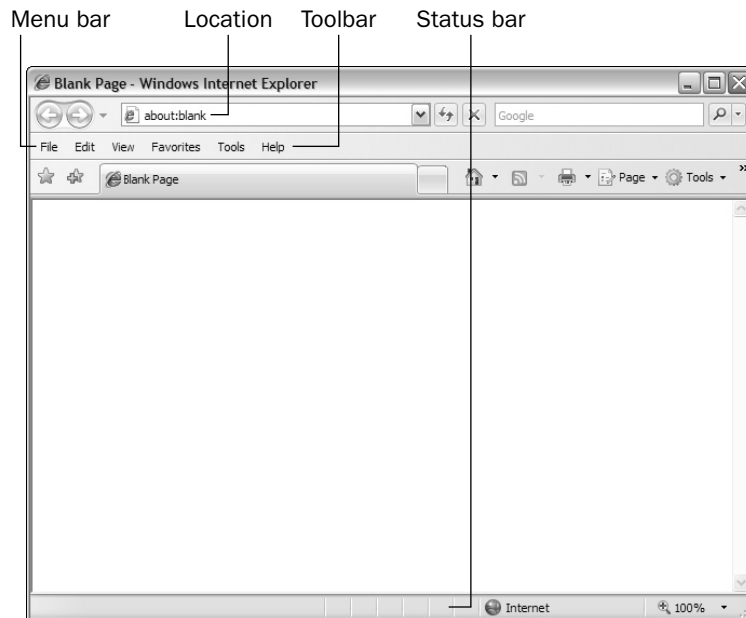


Figure 7-10

Chapter 7: Windows and Frames

However, if you specify even one of the features, all the others (except size and position properties) are set to `no` by default. For example, although you have defined its size, the following code produces a window with no features, as shown in Figure 7-11:

```
var newWindow;  
newWindow = window.open("", "myWindow", "width=200,height=120")
```

The larger window is the original page, and the smaller one on top (shown in Figure 7-11) is the pop-up window.

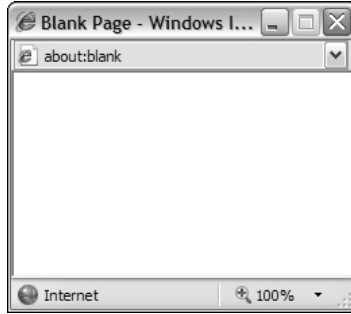


Figure 7-11

Let's see another example. The following creates a resizable 250-by-250-pixel window, with a location field and menu bar:

```
var newWindow;  
newWindow =  
window.open("", "myWindow", "width=250,height=250,location,menubar,resizable")
```

A word of warning, however: Never include spaces inside the features string; otherwise some browsers will consider the string invalid and ignore your settings.

Scripting Between Windows

You've taken a brief look at how you can manipulate the new window's properties and methods, and access its `document` object using the return value from the `window.open()` method. Now you're going to look at how the newly opened window can access the window that opened it and, just as with frames, how it can use functions there.

The key to accessing the window object of the window that opened the current window is the `window` object's `opener` property. This returns a reference to the `window` object of the window that opened the current window. So the following code will change the background color of the `opener` window to red:

```
window.opener.document.bgColor = "RED"
```

You can use the reference pretty much as you used the `window.parent` and `window.top` properties when using frames.

Try It Out Inter-Window Scripting

Let's look at an example wherein you open a new window and access a form on the opener window from the new window.

```
<html>
<head>
<script language="JavaScript" type="text/javascript">
var newWindow;
function butOpenWin_onclick()
{
var winTop = (screen.height / 2) - 125;
var winLeft = (screen.width / 2) - 125;
var windowFeatures = "width=250,height=250,";
windowFeatures = windowFeatures + "left=" + winLeft + ",";
windowFeatures = windowFeatures + "top=" + winTop;
newWindow = window.open("newWindow.htm", "myWindow", windowFeatures);
}
function butGetText_onclick()
{
if (typeof(newWindow) == "undefined" || newWindow.closed == true)
{
alert("No window is open");
}
else
{
document.form1.text1.value = newWindow.document.form1.text1.value;
}
}

function window_onunload()
{
if (typeof(newWindow) != "undefined" && newWindow.closed == false)
{
newWindow.close();
}
}

</script>
</head>
<body onunload="window_onunload()">
<form name=form1> <input type="button" value="Open newWindow" name=butOpenWin
onclick="butOpenWin_onclick()"> <br><br> NewWindow's Text <br>
<input type="text" name=text1>
<br> <input type="button" value="Get Text" name=butGetText
onclick="return butGetText_onclick()">
</form>
</body>
</html>
```

This is the code for your original window. Save it as `openerwindow.htm`. Now you'll look at the page that will be loaded by the opener window.

Chapter 7: Windows and Frames

```
<html>
<head>
<script language="JavaScript" type="text/javascript">
function butGetText_onclick()
{
document.form1.text1.value = window.opener.document.form1.text1.value;
}
</script>
</head>
<body>
<form name=form1>
Opener window's text<BR>
<input type="text" name=text1>
<br>
<input type="button" value="Get Text" name=butGetText language="JavaScript"
type="text/javascript"
onclick="butGetText_onclick()" ">
</form>
</body>
</html>
```

Save this as newWindow.htm.

Open openerwindow.htm in your browser, and you'll see a page with the simple form shown in Figure 7-12.

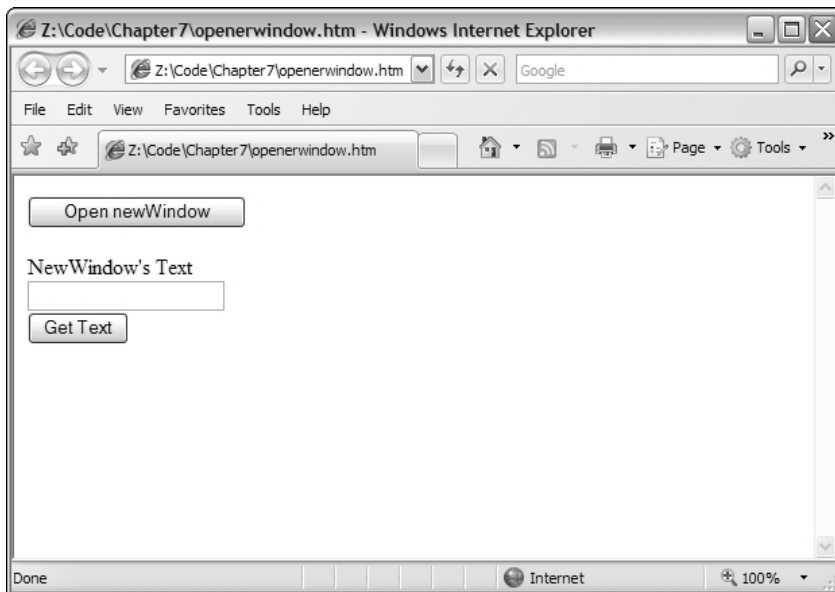


Figure 7-12

Click the `Open newWindow` button, and you'll see the window shown in Figure 7-13 open above the original page.

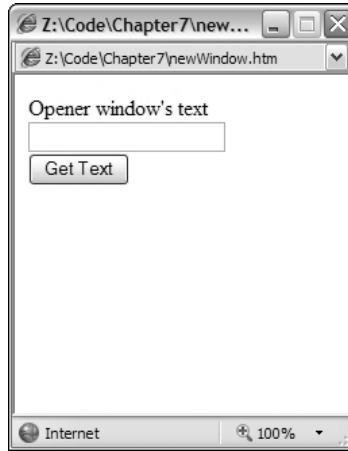


Figure 7-13

Type something into the text box of the new window. Then return to the original opener window, click the `Get Text` button, and you'll see what you just typed into `newWindow` appear in the text box on the opener window's form.

Change the text in the opener window's text box and then return to the `newWindow` and click the `Get Text` button. The text you typed into the opener window's text box will appear in `newWindow`'s text box.

How It Works

Let's look at the opener window first. In the head of the page is a script block in which a variable and three functions are defined. At the top you have declared a new variable, `newWindow`, which will hold the `window` object reference returned by the `window.open()` method you'll use later. Being outside any function gives this variable a global scope, so you can access it from any function on the page.

```
var newWindow;
```

Then you have the first of the three functions in this page, `butOpenWin_onclick()`, which is connected further down the page to the `Open newWindow` button's `onclick` event handler. Its purpose is simply to open the new window.

Rather than have the new window open up anywhere on the page, you use the built-in `screen` object, which is a property of the `window` object, to find out the resolution of the user's display and place the window in the middle of the screen. The `screen` object has a number of read-only properties, but you're interested here in the `width` and `height` properties. You initialize the `winTop` variable to the vertical position onscreen at which you want the top edge of the popup window to appear. The `winLeft` variable is set to the horizontal position onscreen at which you want the left edge of the pop-up window to appear. In this case, you want the position to be in the middle of the screen both horizontally and vertically.

Chapter 7: Windows and Frames

```
function butOpenWin_onclick()
{
    var winTop = (screen.height / 2) - 125;
    var winLeft = (screen.width / 2) - 125;
```

You build up a string for the window features and store it in the `windowFeatures` variable. You set the width and height to 250 and then use the `winLeft` and `winTop` variables you just populated to create the initial start positions of the window.

```
var windowFeatures = "width=250,height=250,";
windowFeatures = windowFeatures + "left=" + winLeft + ",";
windowFeatures = windowFeatures + "top=" + winTop;
```

Finally, you open the new window, making sure you put the return value from `window.open()` into global variable `newWindow` so you can manipulate it later.

```
newWindow = window.open("newWindow.htm", "myWindow", windowFeatures);
}
```

The next function is used to obtain the text from the text box on the form in `newWindow`.

In this function you use an `if` statement to check two things. First, you check that `newWindow` is defined and second, that the window is actually open. You check because you don't want to try to access a nonexistent window, for example if no window has been opened or a window has been closed by the user. The `typeof` operator returns the type of information held in a variable, for example number, string, Boolean, object, and undefined. It returns `undefined` if the variable has never been given a value, as `newWindow` won't have been if no new window has been opened.

Having confirmed that a window has been opened at some point, you now need to check whether it's still open, and the window object's `closed` property does just that. If it returns `true`, the window is closed, and if it returns `false`, it's still open. (Do not confuse this `closed` property with the `close()` method you saw previously.)

In the `if` statement you'll see that checking if `newWindow` is defined comes first, and this is no accident. If `newWindow` really were undefined, `newWindow.closed` would cause an error, because there are no data inside `newWindow`. However, you are taking advantage of the fact that if an `if` statement's condition will be `true` or `false` at a certain point regardless of the remainder of the condition, the remainder of the condition is not checked.

```
function butGetText_onclick()
{
    if (typeof(newWindow) == "undefined" || newWindow.closed == true)
    {
        alert("No window is open");
    }
```

If `newWindow` exists and is open, the `else` statement's code will execute. Remember that `newWindow` will contain a reference to the window object of the window opened. This means you can access the form in `newWindow`, just as you'd access a form on the page the script's running in, by using the `document` object inside the `newWindow` window object.


```

else
{
    document.form1.text1.value = newWindow.document.form1.text1.value;
}
}

```

The last of the three functions is `window_onunload()`, which is connected to the `onunload` event of this page and fires when either the browser window is closed or the user navigates to another page. In the `window_onunload()` function, you check to see if `newWindow` is valid and open in much the same way that you just did. You must check to see if the `newWindow` variable is defined first. With the `&&` operator, JavaScript checks the second part of the operation only if the first part evaluates to `true`. If `newWindow` is defined, and does therefore hold a window object (even though it's possibly a closed window), you can check the `closed` property of the window. However, if `newWindow` is undefined, the check for its `closed` property won't happen, and no errors will occur. If you check the `closed` property first and `newWindow` is undefined, an error will occur, because an undefined variable has no `closed` property.

```

function window_onunload()
{
    if (typeof(newWindow) != "undefined" && newWindow.closed == false)
    {
        newWindow.close();
    }
}

```

If `newWindow` is defined and open, you close it. This prevents the `newWindow`'s Get Text button from being clicked when there is no opener window in existence to get text from (since this function fires when the opener window is closed).

Let's now look at the code for the page that will be loaded in your `newWindow`, namely `newWindow.htm`. This page contains one function, `butGetText_onclick()`. This is connected to the `onclick` event handler of the Get Text button in the page and is used to retrieve the text from the opener window's text box.

```

function butGetText_onclick()
{
    document.form1.text1.value = window.opener.document.form1.text1.value;
}

```

In this function, you use the `window.opener` property to get a reference to the window object of the window that opened this one, and then use that reference to get the value out of the text box in the form in that window. This value is placed inside the text box in the current page.

Moving and Resizing Windows

Before leaving the subject of windows, let's look at the methods available to you for resizing and moving existing windows.

After you have opened a window, you can change its onscreen position and its size using the window object's `resizeTo()` and `moveTo()` methods, both of which take two arguments in pixels.

Chapter 7: Windows and Frames

Imagine that, having just opened a new window, like this:

```
var newWindow = window.open(myURL, "myWindow", "width=125,height=150,resizable");
```

you want to make it 350 pixels wide by 200 pixels high and move it to a position 100 pixels from the left of the screen and 400 pixels from the top. What code would you need?

```
newWindow.resizeTo(350,200);  
newWindow.moveTo(100,400);
```

You can see that you can resize your window to 350 pixels wide by 200 pixels high using `resizeTo()`. Then you move it so it's 100 pixels from the left of the screen and 400 pixels from the top of the screen using `moveTo()`.

The `window` object also has `resizeBy()` and `moveBy()` methods. These each take two parameters, in pixels. For example:

```
newWindow.resizeBy(100,200);
```

This code will increase the size of `newWindow` by 100 pixels horizontally and 200 pixels vertically. Similarly,

```
newWindow.moveBy(20,50);
```

will move the `newWindow` by 20 pixels horizontally and 50 pixels vertically.

When using these methods, you must bear in mind that users can manually resize these windows if they so wish. In addition, the size of the client's screen in pixels will vary between users.

Security

Browsers, such as Firefox and Internet Explorer, put certain restrictions on what information scripts can access between frames and windows.

If all the pages in these frames and windows are based on the same server, or on the same computer when you're loading them into the browser locally, as you are in these examples, you have a reasonably free rein over what your scripts can access and do. However, some restrictions do exist. For example, if you try to use the `window.close()` method in a script page loaded into a browser window that the user opened, as opposed to a window opened by your script, a message box will appear giving the user the option of canceling your `close()` method and keeping the window open.

When a page in one window or frame hosted on one server tries to access the properties of a window or frame that contains a page from a different server, the same-origin policy comes into play, and you'll find yourself very restricted as to what your scripts can do.

Imagine you have a page hosted on a web server whose URL is `http://www.myserver.com`. Inside the page is the following script:

```
var myWindow =  
window.open("http://www.anotherserver.com/anotherpage.htm", "myWindow");
```

Now you have two windows, one that is hosted at `www.myserver.com` and another that is hosted on a different server, `www.anotherserver.com`. Although this code does work, the same-origin policy prevents any access to the `document` object of one page from another. For example, the following code in the opener page will cause a security problem and will be prevented by the browser:

```
var myVariable = myWindow.document.form1.text1.value;
```

Although you do have access to the `window` object of the page on the other server, you have access to a limited subset of its properties and methods.

The same-origin restriction applies to frames and windows equally. The idea behind it is very sound: It is there to prevent hackers from putting your pages inside their own and extracting information by using code inside their pages. However, the restrictions are fairly severe, perhaps too severe, and mean that you should avoid scripting across frames or windows if the pages are hosted on different servers.

Trivia Quiz

As you left it in the previous chapter, the trivia quiz was simply a single page that asked a single randomly selected question. Your task for the trivia quiz in this chapter is to convert it from a single-page application to a multi-frame-based application containing six pages. This is not a small change and will require a lot of work. The enhancements in this chapter will transform the quiz into something resembling a proper application. When the application is first loaded, the user will be presented with the screen shown in Figure 7-14.

As you can see, this is quite a change from the way the quiz looked in Chapter 6! Next you'll look at the strategy for creating this application.

On some browsers, for example some versions of Internet Explorer 6, you may see a warning bar when you load the quiz. This only occurs when you load the page from your local computer; it won't occur if the page is loaded from a web site. When it asks if you want to run the active content, click Yes.

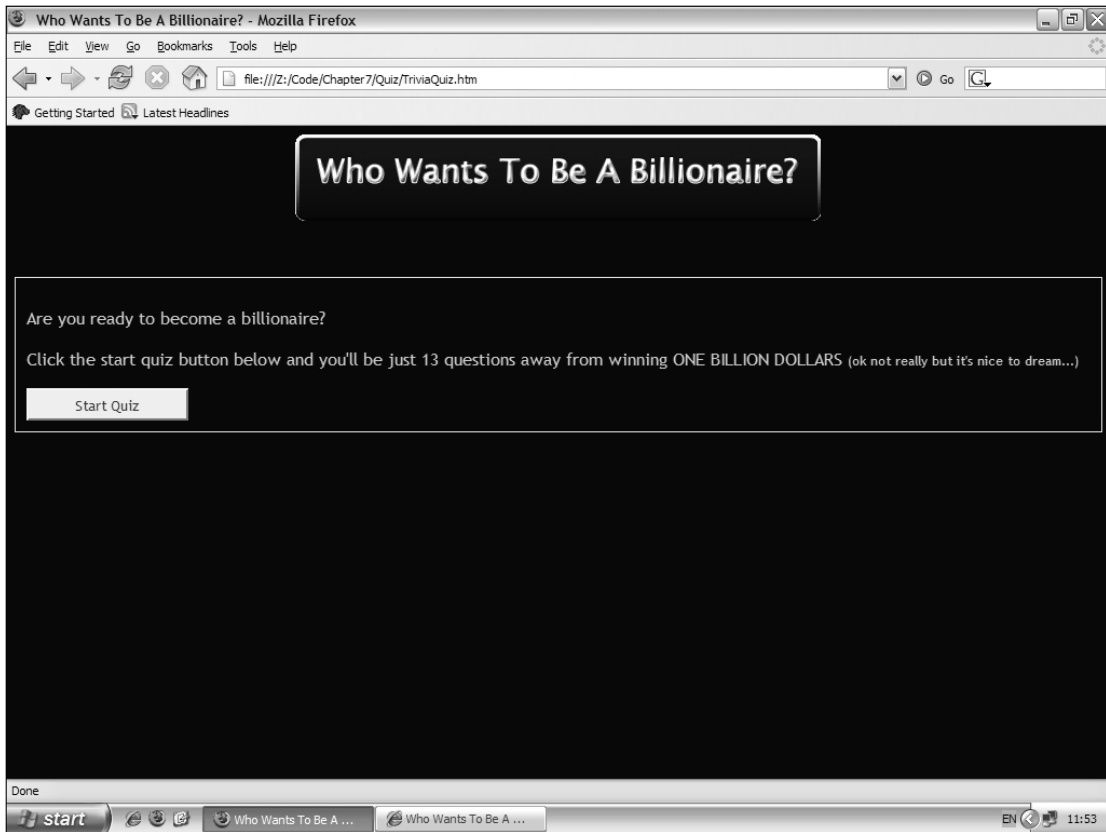


Figure 7-14

Creating the New Trivia Quiz

The idea behind using frames is that there will be a page called `globalfunctions.htm` to hold all the global functions that you use again and again. This page will be loaded into a frame called `fraGlobalFunctions`.

There will also be a page that simply displays the banner, Who Wants To Be A Billionaire, shown in Figure 7-14. This page is called `menubar.htm` and will be loaded into a frame called `fraMenubar`.

The third and fourth pages are where all the action takes place as far as the user is concerned. The `QuizPage.htm` is where the quiz is started—it displays the welcome message and Start Quiz button that you can see in Figure 7-14. The `AskQuestion.htm` page is where the questions are displayed and answered, and finally, when the quiz is finished, where the results are listed. These will be loaded into the frame called `fraQuizPage`.

The two other pages are called `TriviaQuiz.htm` and `TopFrame.htm`, whose job is solely to define the framesets for the frames containing the other pages. These frameset pages will be contained in the frames called `Top window` and `fraTopFrame`.

The frame structure of the application is shown in Figure 7-15, along with the name of each frame. Note that although the `fraGlobalFunctions` frame is shown, it's actually invisible to the user, so they won't be able to easily see the code and cheat by reading the answers.

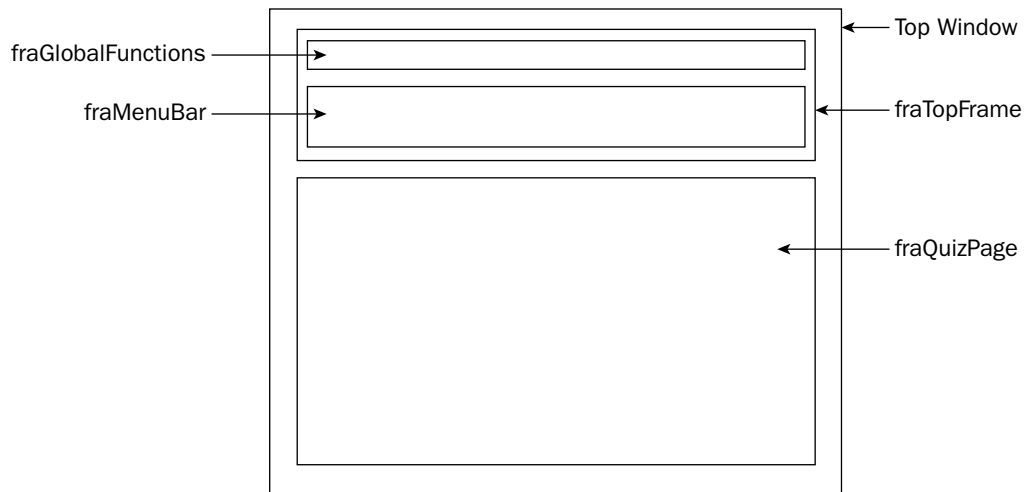


Figure 7-15

In terms of a tree diagram, the frames look like those shown in Figure 7-16.

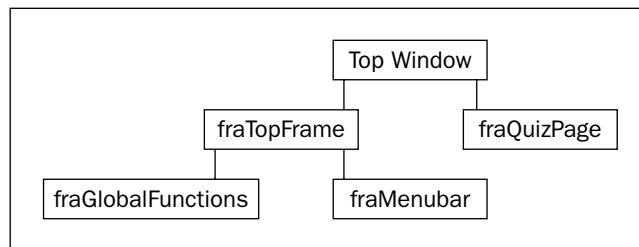


Figure 7-16

You'll now look at each frame in turn and give the code for the page or pages that will be loaded into it.

Top Window

Figure 7-17 shows the frame structure, with the top window frame highlighted.

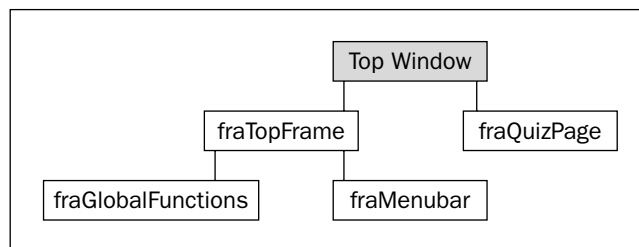


Figure 7-17

Chapter 7: Windows and Frames

Create the frameset page that defines the top and bottom frames that you can see.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
"http://www.w3.org/TR/html4/frameset.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<title>Wrox Online Trivia Quiz</title>
</head>
<frameset rows="120,*" border="0">
<frame src="TopFrame.htm" name="fraTopFrame">
<frame src="QuizPage.htm" name="fraQuizPage">
</frameset>
</html>
```

Save this as `TriviaQuiz.htm`.

This is the page the user loads into his browser. It defines the frames `fraTopFrame` and `fraQuizPage` and specifies the pages that will be loaded into them.

fraQuizPage

The next frame you're looking at is `fraQuizPage`, whose position in the frames hierarchy is shown in Figure 7-18. This frame will have two pages loaded into it in turn: `QuizPage.htm` and `AskQuestion.htm`.

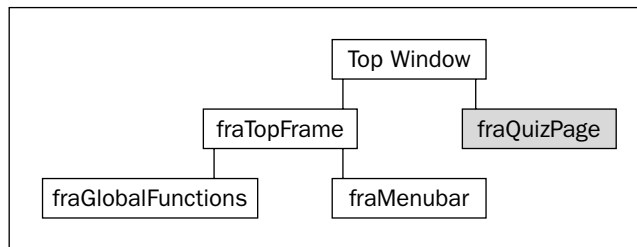


Figure 7-18

QuizPage.htm

In the previous screenshot you saw what the trivia quiz looks like before the quiz has started. You simply have a start page with a bit of text and a button to click that starts the quiz. When the quiz is finished, this page is loaded again if the user asks to restart the quiz.

Let's create that start page. When you've finished typing the code, save the page as `QuizPage.htm`.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<title>Quiz Page</title>
<script language="JavaScript" type="text/javascript">
function cmdStartQuiz_onclick()
```

```

{
window.top.fraTopFrame.fraGlobalFunctions.resetQuiz();
window.location.href = "AskQuestion.htm";
}
</script>
<style type="text/css">
<!--
body {
background-color: #000033;
}
body,td,th {
font-family: Trebuchet MS, helvetica, sans-serif;
color: #CCCCCC;
}

input, select {

font-family: "Trebuchet MS", "Lucida Sans", Georgia, "Times New Roman", Times,
serif;
font-size: 13px;
color: #333333;
padding: 1px;
background: #EEE;

}
.SmallText {font-size: 12px}

div#MainBodyDIV
{
border:1px;
border-color:#CCCCCC;
border-style:groove;
padding:10px;
margin-top:20px;
}
-->
</style>
</head>
<body>

<div id="MainBodyDIV">
<p>Are you ready to become a billionaire?</p>
<p>Click the Start Quiz button below and you'll be just 13 questions away from
winning ONE BILLION DOLLARS.
<span class="SmallText">(ok not really but it's nice to dream...)</span> </p>
<form name="frmQuiz">
<input name="cmdStartQuiz" type="button" value="Start Quiz"
onclick="return cmdStartQuiz_onclick()" style="width:150px; height:30px;">
</p>
</form>
</div>

</p>
</body>
</html>

```

AskQuestion.htm

After the Start Quiz button has been clicked, the next page loaded into the `fraQuizPage` frame is `AskQuestion.htm`, as shown in Figure 7-19 (the questions are randomly selected, so you may have a different start question).

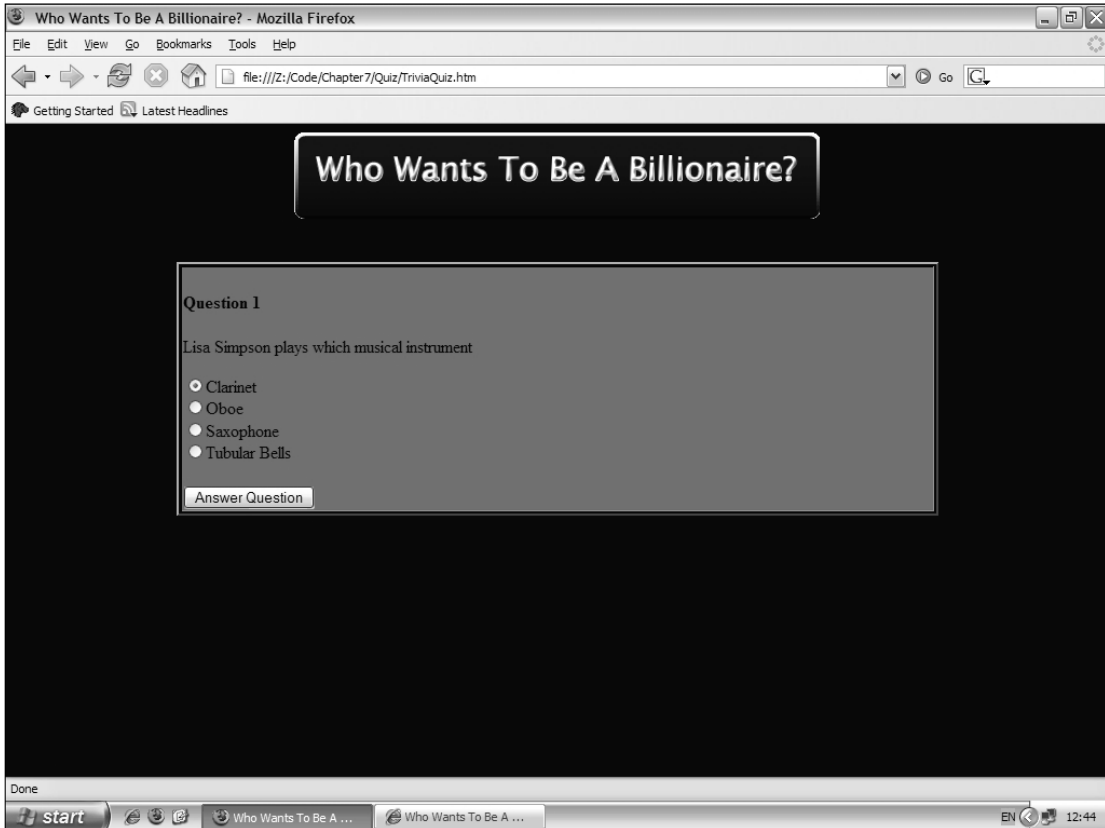


Figure 7-19

When the Answer Question button is clicked, the user's answer is checked, and this page is reloaded. If there are more questions to ask, then another, different, randomly selected question is shown. If you've come to the end of the quiz, a results page like the one shown in Figure 7-20 is displayed. Actually, the results page and question-asking pages are the same HTML page, but they're created dynamically depending on whether the quiz has ended.

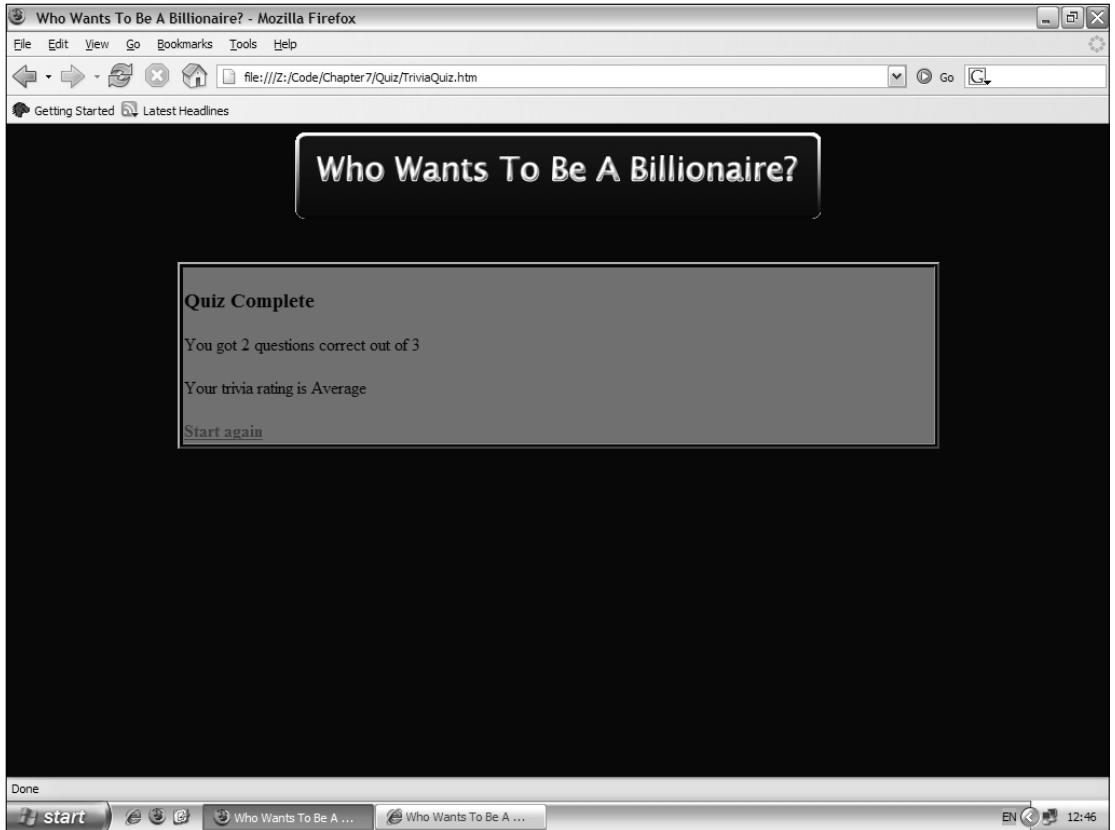


Figure 7-20

You'll now create that page, which should be saved as `AskQuestion.htm`.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<title>Ask Questions</title>
<script language="JavaScript" type="text/javascript">
var globalFunctions;
globalFunctions = window.top.fraTopFrame.fraGlobalFunctions;
function getAnswer()
{
var answer = 0;
while (document.QuestionForm.radQuestionChoice[answer].checked != true)
{
answer++;
}
}
```

```
return String.fromCharCode(65 + answer);
}
function buttonCheckQ_onclick()
{
var questionNumber = globalFunctions.currentQNumber;
if (globalFunctions.answerCorrect(questionNumber, getAnswer()) == true)
{
alert("You got it right");
}
else
{
alert("You got it wrong");
}
window.location.reload();
}
</script>
</head>
<body style="background-color: #000033;">
<table align=center border="2" width="70%">
<tr>
<td bgcolor=RoyalBlue>
<form name="QuestionForm">
<script language="JavaScript" type="text/javascript">
document.write(globalFunctions.getQuestion());
</script>
</form>
</td>
</tr>
</table>
</body>
</html>
```

fraTopFrame

The next frame you're looking at is `fraTopFrame`, whose position in the frames hierarchy is shown in Figure 7-21.

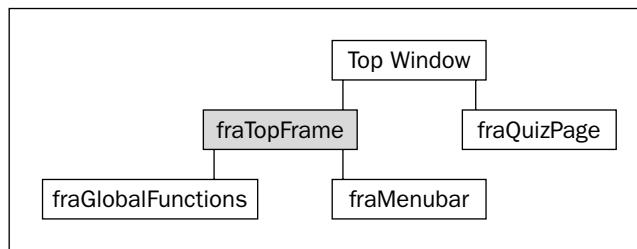


Figure 7-21

This other frame defined in `Topwindow` is, in fact, another frameset-defining page. It defines one visible frame, the one containing the page heading, and a second frame, which is not visible and which contains your global functions, but no HTML. Let's create that frameset-defining page next.

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
"http://www.w3.org/TR/html4/frameset.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<title>Top Frame</title>
</head>
<frameset rows="0,*" border="0">
<frame src="GlobalFunctions.htm" name="fraGlobalFunctions">
<frame src="Menubar.htm" name="fraMenubar">
</frameset>
</html>

```

Save this page as `TopFrame.htm`.

You can see that it defines the two frames called `fraGlobalFunctions` and `fraMenubar`, which you will look at next.

fraMenubar

You'll next create the page for the `fraMenubar` frame, whose position in the frames hierarchy is shown in Figure 7-22.

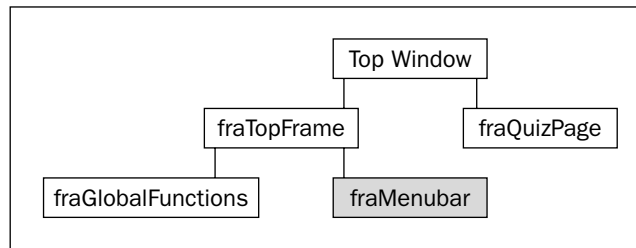


Figure 7-22

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Menu Bar</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
</head>
<body style="background-color:#000033">
<div style="text-align:center;">

</div>
</body>
</html>

```

Save this as `menubar.htm`.

This page just defines the heading that can be seen throughout the trivia quiz. It also uses an image called `MainLogo.gif`, which you will need to create or retrieve from the code download.

fraGlobalFunctions

In Figure 7-23, you can see where `fraGlobalFunctions` fits into your frames hierarchy.

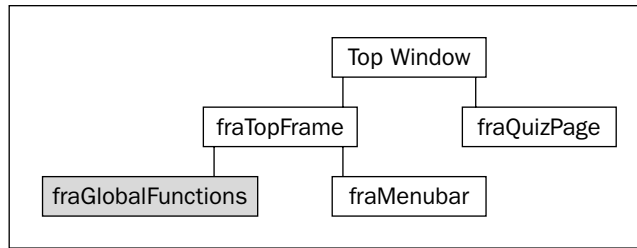


Figure 7-23

Now turn your attention to the final new page, namely `globalfunctions.htm`, which serves as a module containing all your general JavaScript functions. It is contained in the frame `fraGlobalFunctions`. You may recognize some of the code from the `trivia_quiz.htm` page that constituted the trivia quiz you created in Chapter 6.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<title>Global functions</title>
<script language="JavaScript" type="text/javascript">
// questions and answers variables will holds questions and answers
var questions = new Array();
var answers = new Array();
var questionsAsked;
var numberOfQuestionsAsked = 0;
var numberOfQuestionsCorrect = 0;
var currentQNumber = -1;
// define question 1
questions[0] = new Array();
questions[0][0] = "The Beatles were";
questions[0][1] = "A sixties rock group from Liverpool";
questions[0][2] = "Four musically gifted insects";
questions[0][3] = "I don't know - can I have the questions on baseball please";
// assign answer for question 1
answers[0] = "A";
// define question 2
questions[1] = new Array();
questions[1][0] = "Homer Simpson's favorite food is";
questions[1][1] = "Fresh salad";
questions[1][2] = "Doughnuts";
questions[1][3] = "Bread and water";
questions[1][4] = "Apples";
// assign answer for question 2
answers[1] = "B";
```

```

// define question 3
questions[2] = new Array();
questions[2][0] = "Lisa Simpson plays which musical instrument";
questions[2][1] = "Clarinet";
questions[2][2] = "Oboe";
questions[2][3] = "Saxophone";
questions[2][4] = "Tubular Bells";
// assign answer for question 3
answers[2] = "C";
function resetQuiz()
{
    var indexCounter;
    currentQNumber = -1;
    questionsAsked = new Array();
    for (indexCounter = 0; indexCounter < questions.length; indexCounter++)
    {
        questionsAsked[indexCounter] = false;
    }
    numberOfQuestionsAsked = 0;
    numberOfQuestionsCorrect = 0;
}
function answerCorrect(questionNumber, answer)
{
    // declare a variable to hold return value
    var correct = false;
    // if answer provided is same as answer then correct answer is true
    if (answer == answers[questionNumber])
    {
        numberOfQuestionsCorrect++;
        correct = true;
    }
    // return whether the answer was correct (true or false)
    return correct;
}
function getQuestion()
{
    if (questions.length != numberOfQuestionsAsked)
    {
        var questionNumber = Math.floor(Math.random() * questions.length)
        while (questionsAsked[questionNumber] == true)
        {
            questionNumber = Math.floor(Math.random() * questions.length);
        }
        var questionLength = questions[questionNumber].length;
        var questionChoice;
        numberOfQuestionsAsked++;
        var questionHTML = "<h4>Question " + numberOfQuestionsAsked + "</h4>";
        questionHTML = questionHTML + "<p>" + questions[questionNumber][0];
        questionHTML = questionHTML + "</p>";
        for (questionChoice = 1; questionChoice < questionLength; questionChoice++)
        {
            questionHTML = questionHTML + "<input type=radio "
            questionHTML = questionHTML + "name=radQuestionChoice"

```

```
if (questionChoice == 1)
{
questionHTML = questionHTML + " checked";
}
questionHTML = questionHTML + ">" +
questions[questionNumber][questionChoice];
questionHTML = questionHTML + "<br>"
}
questionHTML = questionHTML + "<br><input type='button' "
questionHTML = questionHTML + " value='Answer Question'";
questionHTML = questionHTML + "name=buttonNextQ ";
questionHTML = questionHTML + "onclick='return buttonCheckQ_onclick()'>";
currentQNumber = questionNumber;
questionsAsked[questionNumber] = true;
}
else
{
var questionHTML = "<h3>Quiz Complete</h3>";
questionHTML = questionHTML + "You got " + numberOfQuestionsCorrect;
questionHTML = questionHTML + " questions correct out of "
questionHTML = questionHTML + numberOfQuestionsAsked;
questionHTML = questionHTML + "<br><br>Your trivia rating is "
switch(Math.round(((numberOfQuestionsCorrect / numberOfQuestionsAsked) * 10)))
{
case 0:
case 1:
case 2:
case 3:
questionHTML = questionHTML + "Beyond embarrassing";
break;
case 4:
case 5:
case 6:
case 7:
questionHTML = questionHTML + "Average";
break;
default:
questionHTML = questionHTML + "Excellent"
}
questionHTML = questionHTML + "<br><br><A "
questionHTML = questionHTML + "href='quizpage.htm'><strong>"
questionHTML = questionHTML + "Start again</strong></A>"
}
return questionHTML;
}
</script>
</head>
<body>
</body>
</html>
```

Save this page as `GlobalFunctions.htm`. That completes all the pages, so now it's time to load the new trivia quiz and find out how it works.

How It Works

Load `TriviaQuiz.htm` into your browser to start the quiz and try it out.

Although there does appear to be a lot of new code, much of it is identical to that in the previous version of the trivia quiz.

You will take a closer look at the pages `QuizPage.htm` and `AskQuestion.htm`, which are loaded into the `fraQuizPage` frame, and `GlobalFunctions.htm`, which is loaded into the `fraGlobalFunction` frame.

Of the other pages, `TriviaQuiz.htm` and `TopFrame.htm` are simply frameset-defining pages, and `menubar.htm` simply defines the heading for the page.

QuizPage.htm

This is a simple page. You define a function, `cmdStartQuiz_onclick()`, which is connected to the `onclick` event handler of the Start Quiz button further down the page.

```
function cmdStartQuiz_onclick()
{
    window.top.fraTopFrame.fraGlobalFunctions.resetQuiz();
    window.location.href = "AskQuestion.htm";
}
```

In this function, you reset the quiz by calling the `resetQuiz()` function, which is in your `fraGlobalFunctions` frame. You'll be looking at this shortly. To get a reference to the window object of `fraGlobalFunctions`, you need to get a reference to the `fraTopFrame` that is under the top window.

On the second line of the function, you navigate the frame to `AskQuestion.htm`, the page where the questions are asked. Let's look at that next.

AskQuestion.htm

In this page, you'll access the functions in the `fraGlobalFunctions` frame a number of times, so you declare a page-level variable and set it to reference the window object of `fraGlobalFunctions`. This saves on typing and makes your code more readable.

```
var globalFunctions;
globalFunctions = window.top.fraTopFrame.fraGlobalFunctions;
```

You then come to the `getAnswer()` function, which retrieves from the form lower in the page the option the user chose as her answer. It does this by looping through each option in the form, incrementing the variable `answer` until it finds the option that has been checked by the user. Remember that the answers are stored as A, B, C, and so on, so you convert the index number to the correct character using the `fromCharCode()` method of the `String` object. This is identical to the action of the first half of the `buttonCheckQ_onclick()` function that you saw in the previous incarnation of the trivia quiz.

```
function getAnswer()
{
    var answer = 0;
    while (document.QuestionForm.radQuestionChoice[answer].checked != true)
    {
```

```
answer++;
    }
    return String.fromCharCode(65 + answer);
}
```

The second function, `buttonCheckQ_onclick()`, will be connected to the Check Answer button's `onclick` event. It is similar to the second half of the function with the same name in the previous version of the quiz. However, now it refers to the `answerCorrect()` function in the `fraGlobalFunctions` frame rather than the current page, and uses the `getAnswer()` function rather than the variable `answer`.

```
function buttonCheckQ_onclick()
{
    var questionNumber = globalFunctions.currentQNumber;
    if (globalFunctions.answerCorrect(questionNumber,getAnswer()) == true)
    {
        alert("You got it right");
    }
    else
    {
        alert("You got it wrong");
    }
    window.location.reload();
}
```

As in Chapter 6, the form that displays the question to the user is populated dynamically, by means of `document.write()`. However, this time the function, `getQuestion()`, is located in the `GlobalFunctions.htm` page.

GlobalFunctions.htm

Much of this page is taken from the `trivia_quiz.htm` page that you created in Chapter 6. At the top of the page you add four more page-level variables.

```
var questions = new Array();
var answers = new Array();
var questionsAsked;
var numberOfQuestionsAsked = 0;
var numberOfQuestionsCorrect = 0;
var currentQNumber = -1;
```

You then define the questions and answers arrays exactly as you did previously. The first new function, `resetQuiz()`, is shown here:

```
function resetQuiz()
{
    var indexCounter;
    currentQNumber = -1;
    questionsAsked = new Array();
    for (indexCounter = 0; indexCounter < questions.length;indexCounter++)
    {
        questionsAsked[indexCounter] = false;
    }
}
```



```

    }
    numberOfQuestionsAsked = 0;
    numberOfQuestionsCorrect = 0;
  }

```

When the quiz is started or restarted, this function is called to reset all the global quiz variables back to a default state. For example, the `questionsAsked` variable is reinitialized to a new array, the length of which will match the length of the `questions` array, with each element being set to a default value of `false`, indicating that the corresponding question has not yet been asked.

You then have the `answerCorrect()` function, which is the same as in the previous chapter. The rest of the page is made up of the `getQuestion()` function, which has undergone major changes since the previous version.

Previously, you asked questions randomly and kept going until the user got bored. Now you're going to keep track of which questions have been asked and how many questions have been asked. The `questionsAsked` array will store which questions have already been asked, so you can avoid repeating questions. The variable `numberOfQuestionsAsked` keeps track of how many have been asked so far, so you can stop when you've used up your question database. The variable `numberOfQuestionsCorrect` will be used to record the number of right answers given. These variables were defined in the head of the page.

Turning to `getQuestion()`, you can see that the very first thing the function does is use an `if` statement to see if you have asked as many questions as there are questions in the database. The `length` property of your `questions` array tells you how many elements there are in your array, and `numberOfQuestionsAsked` tells you how many have been asked so far. If you have asked all the questions, then later on you'll see that the function writes out an end page with details of how many the user got correct and rates her trivia knowledge.

```

function getQuestion()
{
  if (questions.length != numberOfQuestionsAsked)
  {
    var questionNumber = Math.floor(Math.random() * questions.length)
    while (questionsAsked[questionNumber] == true)
    {
      questionNumber = Math.floor(Math.random() * questions.length);
    }
  }
}

```

You can see from the preceding code that the selection of the question is random, as it was in the Chapter 6 version of the quiz. However, you have added a `while` loop that makes use of the `questionsAsked` array you declared earlier. Each time a question is asked, you set the value of the element in the `questionsAsked` array at the same position as its question number to `true`. By checking to see if a particular array position is `true`, you can tell if the question has already been asked, in which case the `while` loop keeps going until it hits a `false` value—that is, an unasked question.

Now that you know which question you want to ask, you just need to go ahead and ask it, which is the purpose of the next lines of code.

```
var questionLength = questions[questionNumber].length;
var questionChoice;
numberOfQuestionsAsked++;
var questionHTML = "<h4>Question " + numberOfQuestionsAsked + "</h4>";
questionHTML = questionHTML + "<p>" + questions[questionNumber][0];
questionHTML = questionHTML + "</p>";
for (questionChoice = 1; questionChoice < questionLength; questionChoice++)
{
    questionHTML = questionHTML + "<input type=radio "
    questionHTML = questionHTML + "name=radQuestionChoice"
    if (questionChoice == 1)
    {
        questionHTML = questionHTML + " checked";
    }
    questionHTML = questionHTML + ">" +
    questions[questionNumber][questionChoice];
    questionHTML = questionHTML + "<br>"
}
questionHTML = questionHTML + "<br><input type='button' "
questionHTML = questionHTML + " value='Answer Question'";
questionHTML = questionHTML + "name=buttonNextQ ";
questionHTML = questionHTML + "onclick='return buttonCheckQ_onclick()'>";
```

This code is almost identical to its previous form in Chapter 6, except that you now create the button as well as the answer options dynamically. Why? At the beginning of the function you saw that an `if` statement checked whether you had reached the end of the quiz. If not, you created another question, as you are doing here. If the quiz has come to an end, you don't want to create an array of answers. Instead you want to create an end-of-quiz form. The only way to avoid having the Answer Question button there is to make it part of the dynamic question creation.

Finally, you see that the `questionsAsked` array is updated—that is, the question just asked is stored in the array as follows:

```
currentQNumber = questionNumber;
questionsAsked[questionNumber] = true;
}
```

The `else` part of the `if` statement from the top of the function is shown next. Its purpose is to create the “quiz completed” message. You build up the HTML necessary, storing it in the `questionHTML` variable. You not only specify how many questions the user got right out of how many were asked, but you also rate his knowledge.

```
else
{
    questionHTML = "<h3>Quiz Complete</h3>";
    questionHTML = questionHTML + "You got " + numberOfQuestionsCorrect;
    questionHTML = questionHTML + " questions correct out of "
    questionHTML = questionHTML + numberOfQuestionsAsked;
    questionHTML = questionHTML + "<br><br>Your trivia rating is "
```

The rating is done with the `switch` statement, where the statement is based on questions answered correctly divided by the number of questions asked, which for simplicity you multiply by 10 and round to the nearest integer. Then you use the `case` statements to create the correct rating. Remember that code execution starts at the first `case` statement that matches and continues until either the `switch` statement ends or a `break` statement is reached. So if your rating calculation

```
Math.round(((numberOfQuestionsCorrect / numberOfQuestionsAsked) * 10)
```

were 1, the code would start executing from the `case 1:` statement and continue until the `break` statement in `case 3`. Essentially this means that a rating of 0–3 will be described as *Beyond embarrassing*, 4–7 as *Average*, and anything else, that is, the default case, as *Excellent*.

```
switch(Math.round(((numberOfQuestionsCorrect / numberOfQuestionsAsked) * 10)))
{
case 0:
case 1:
case 2:
case 3:
questionHTML = questionHTML + "Beyond embarrassing";
break;
case 4:
case 5:
case 6:
case 7:
questionHTML = questionHTML + "Average";
break;
default:
questionHTML = questionHTML + "Excellent"
}
```

Finally, you add a link to allow the user to restart the quiz.

```
questionHTML = questionHTML + "<br><br><A "
questionHTML = questionHTML + "href='quizpage.htm'><strong>"
questionHTML = questionHTML + "Start again</strong></A>"
}
```

At the end of the function, you return the HTML to be written into the page: either a new question or the end-of-quiz results.

```
return questionHTML;
}
```

That completes the discussion of the trivia quiz for this chapter. In the next chapter you'll use advanced string manipulation to pose questions requiring a text-based, rather than option-based, answer.

Summary

For various reasons, having a frame-based web site can prove very useful. Therefore, you need to be able to create JavaScript that can interact with frames and with the documents and code within those frames.

Chapter 7: Windows and Frames

- ❑ You saw that an advantage of frames is that, by putting all of your general functions in a single frame, you can create a JavaScript code module that all of your web site can use.
- ❑ You saw that the key to coding with frames is getting a reference to the `window` objects of other frames. You saw two ways of accessing frames higher in the hierarchy, using the `window` object's `parent` property and its `top` property.
- ❑ The `parent` property returns the `window` object that contains the current `window` object, which will be the page containing the frameset that created the window. The `top` property returns the `window` object of the window containing all the other frames.
- ❑ Each frame in a frameset can be accessed through three methods. One is to use the name of the frame. The second is to use the `frames[]` array and specify the index of the frame. The third way is to access the frame by its name in the frames array—for example, `parent.frames.frameName`. This the safest way, because it avoids any collision with global variables.
- ❑ If the frame you want to access is defined in another window, you need the `parent` or `top` property to get a reference to the `window` object defining that frame, and then you must specify the name or position in the `frames[]` array.

You then looked at how you can open new, additional browser windows using script.

- ❑ Using the `window` object's `open()` method, you can open new windows. The URL of the page you want to open is passed as the first parameter; the name of the new window is passed as the second parameter; the optional third parameter enables you to define what features the new window will have.
- ❑ The `window.open()` method returns a value, which is a reference to the `window` object of the new window. Using this reference, you can access the `document`, `script`, and methods of that window, much as you do with frames. You need to make sure that the reference is stored inside a variable if you want to do this.
- ❑ To close a window, you simply use the `window.close()` method. To check if a window is closed, you use the `closed` property of the `window` object, which returns `true` if it's closed and `false` if it's still open.
- ❑ For a newly opened `window` object to access the window that opened it, you need to use the `window.opener` property. Like `window.parent` for frames, this gives a reference to the `window` object that opened the new one and enables you to access the `window` object and its properties for that window.
- ❑ After a window is opened, you can resize it using `resizeTo(x,y)` and `resizeBy(x,y)`, and move it using `moveTo(x,y)` and `moveBy(x,y)`.

You also looked briefly at security restrictions for windows and frames that are not of the same origin. By “not of the same origin,” you’re referring to a situation in which the document in one frame is hosted on one server and the document in the other is hosted on a different server. In this situation, very severe restrictions apply, which limit the extent of scripting between frames or windows.

In the next chapter you look at advanced string manipulation and how you can use it to add different types of questions to your trivia quiz.

Exercise Questions

Suggested solutions to these questions can be found in Appendix A.

Question 1

In the previous chapter's exercise questions, you created a form that allowed the user to pick a computer system. He could view the details of his system and its total cost by clicking a button that wrote the details to a `textarea`. Change the example so it's a frames-based web page; instead of writing to a text area, the user should write the details to another frame.

Question 2

The first example in this chapter was a page with images of books, in which clicking on a book's image brought up information about that book in a pop-up window. Amend this so that the pop-up window also has a button or link that, when clicked, adds the item to the user's shopping basket. Also, on the main page, give the user some way of opening up a shopping basket window with details of all the items he has purchased so far, and give him a way of deleting items from this basket.

8

String Manipulation

In Chapter 4 you looked at the `String` object, which is one of the native objects that JavaScript makes available to you. You saw a number of its properties and methods, including the following:

- ❑ `length` — The length of the string in characters
- ❑ `charAt()` and `charCodeAt()` — The methods for returning the character or character code at a certain position in the string
- ❑ `indexOf()` and `lastIndexOf()` — The methods that allow you to search a string for the existence of another string and that return the character position of the string if found
- ❑ `substr()` and `substring()` — The methods that return just a portion of a string
- ❑ `toUpperCase()` and `toLowerCase()` — The methods that return a string converted to upper- or lowercase

In this chapter you'll look at four new methods of the `String` object, namely `split()`, `match()`, `replace()`, and `search()`. The last three, in particular, give you some very powerful text-manipulation functionality. However, to make full use of this functionality, you need to learn about a slightly more complex subject.

The methods `split()`, `match()`, `replace()`, and `search()` can all make use of *regular expressions*, something JavaScript wraps up in an object called the `RegExp` object. Regular expressions enable you to define a pattern of characters, which can be used for text searching or replacement. Say, for example, that you have a string in which you want to replace all single quotes enclosing text with double quotes. This may seem easy — just search the string for `'` and replace it with `"` — but what if the string is `Bob O'Hara said "Hello"`? You would not want to replace the single-quote character in `O'Hara`. You can perform this text replacement without regular expressions, but it would take more than the two lines of code needed if you do use regular expressions.

Although `split()`, `match()`, `replace()`, and `search()` are at their most powerful with regular expressions, they can also be used with just plain text. You'll take a look at how they work in this simpler context first, to become familiar with the methods.

Additional String Methods

In this section you will take a look at the `split()`, `replace()`, `search()`, and `match()` methods, and see how they work without regular expressions.

The split() Method

The `String` object's `split()` method splits a single string into an array of substrings. Where the string is split is determined by the separation parameter that you pass to the method. This parameter is simply a character or text string.

For example, to split the string "A, B, C" so that you have an array populated with the letters between the commas, the code would be as follows:

```
var myString = "A,B,C";  
var myTextArray = myString.split(',');
```

JavaScript creates an array with three elements. In the first element it puts everything from the start of the string `myString` up to the first comma. In the second element it puts everything from after the first comma to before the second comma. Finally, in the third element it puts everything from after the second comma to the end of the string. So, your array `myTextArray` will look like this:

A	B	C
1	1	1
1	1	0
1	0	1
1	0	0
0	1	1
0	1	0
0	0	1
0	0	0

If, however, your string were "A, B, C, " JavaScript would split it into four elements, the last element containing everything from the last comma to the end of the string; in other words, the last string would be an empty string.

A		B	C
---	--	---	---

This is something that can catch you off guard if you're not aware of it.

Try It Out Reversing the Order of Text

Let's create a short example using the `split()` method, in which you reverse the lines written in a `<textarea>` element.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<title>Example 1</title>
<script language="JavaScript" type="text/JavaScript">
function splitAndReverseText(textAreaControl)
{

    var textToSplit = textAreaControl.value;
    var textArray = textToSplit.split('\n');
    var numberOfParts = 0;
```



```

    numberOfParts = textArray.length;
    var reversedString = "";
    var indexCount;
    for (indexCount = numberOfParts - 1; indexCount >= 0; indexCount--)
    {
        reversedString = reversedString + textArray[indexCount];
        if (indexCount > 0)
        {
            reversedString = reversedString + "\n";
        }
    }

    textAreaControl.value = reversedString;
}
</script>
</head>
<body>
<form name=form1>
<textarea rows="20" cols="40" name="textareal" wrap="soft">Line 1
Line 2
Line 3
Line 4</textarea>
<br>
<input type="button" value="Reverse Line Order" name="buttonSplit"
        onclick="splitAndReverseText(document.form1.textareal)">
</form>
</body>
</html>

```

Save this as `ch8_examp1.htm` and load it into your browser. You should see the screen shown in Figure 8-1.

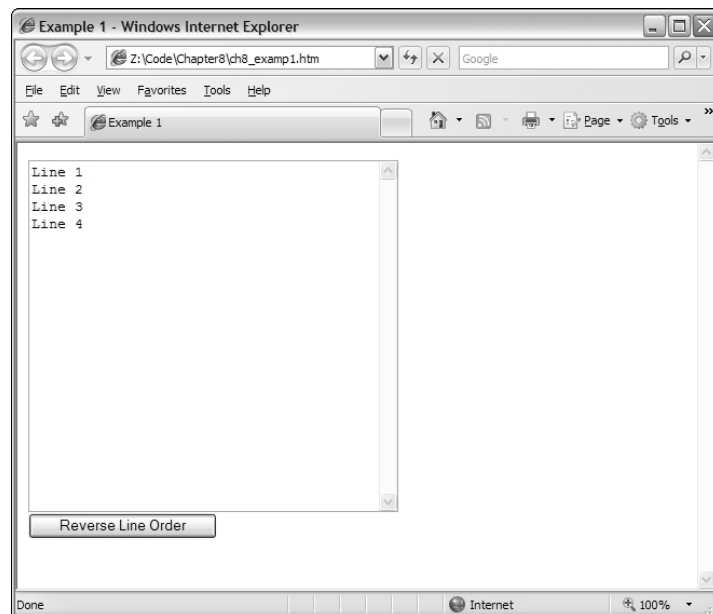


Figure 8-1

Chapter 8: String Manipulation

Clicking the Reverse Line Order button reverses the order of the lines, as shown in Figure 8-2.

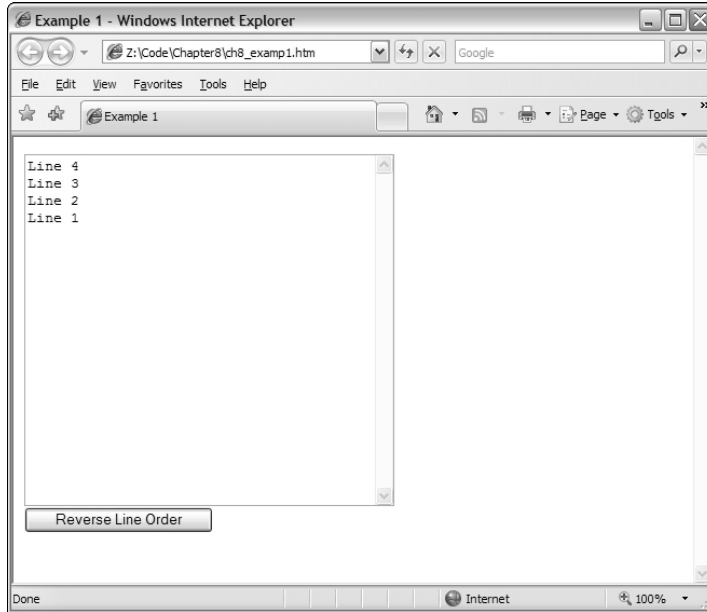


Figure 8-2

Try changing the lines within the text area to test it further.

Although this example works on Internet Explorer as it is, an extra line gets inserted. If this troubles you, you can fix it by replacing each instance of `\n` with `\r\n` for Internet Explorer.

How It Works

The key to how this code works is the function `splitAndReverseText()`. This function is defined in the script block in the head of the page and is connected to the `onclick` event handler of the button further down the page.

```
<input type="button" value="Reverse Line Order" name=buttonSplit  
      onclick="splitAndReverseText(document.form1.textarea1)">
```

As you can see, you pass a reference of the text area that you want to reverse as a parameter to the function. By doing it this way, rather than just using a reference to the element itself inside the function, you make the function more generic, so you can use it with any `textarea` element.

Now, on with the function. You start by assigning the value of the text inside the `textarea` element to the `textToSplit` variable. You then split that string into an array of lines of text using the `split()` method of the `String` object and put the resulting array inside the `textArray` variable.

```
function splitAndReverseText(textAreaControl)
{
    var textToSplit = textAreaControl.value;
    var textArray = textToSplit.split('\n');
```

So what do you use as the separator to pass as a parameter for the `split()` method? Recall from Chapter 2 that the escape character `\n` is used for a new line. Another point to add to the confusion is that Internet Explorer seems to need `\r\n` rather than `\n`.

You next define and initialize three more variables.

```
var numberOfParts = 0;
numberOfParts = textArray.length;
var reversedString = "";
var indexCount;
```

Now that you have your array of strings, you next want to reverse them. You do this by building up a new string, adding each string from the array, starting with the last and working toward the first. You do this in the `for` loop, where instead of starting at 0 and working up as you usually do, you start at a number greater than 0 and decrement until you reach 0, at which point you stop looping.

```
for (indexCount = numberOfParts - 1; indexCount >= 0; indexCount--)
{
    reversedString = reversedString + textArray[indexCount];
    if (indexCount > 0)
    {
        reversedString = reversedString + "\n";
    }
}
```

When you split the string, all your line formatting is removed. So in the `if` statement you add a linefeed (`\n`) onto the end of each string, except for the last string; that is, when the `indexCount` variable is 0.

Finally you assign the text in the `textarea` element to the new string you've built.

```
    textAreaControl.value = reversedString;
}
```

After you've looked at regular expressions, you'll revisit the `split()` method.

The replace() Method

The `replace()` method searches a string for occurrences of a substring. Where it finds a match for this substring, it replaces the substring with a third string that you specify.

Let's look at an example. Say you have a string with the word `May` in it, as shown in the following:

```
var myString = "The event will be in May, the 21st of June";
```

Chapter 8: String Manipulation

Now, say you want to replace `May` with `June`. You can use the `replace()` method like so:

```
myCleanedUpString = myString.replace("May", "June");
```

The value of `myString` will not be changed. Instead, the `replace()` method returns the value of `myString` but with `May` replaced with `June`. You assign this returned string to the variable `myCleanedUpString`, which will contain the corrected text.

```
"The event will be in June, the 21st of June"
```

The `search()` Method

The `search()` method enables you to search a string for a particular piece of text. If the text is found, the character position at which it was found is returned; otherwise `-1` is returned. The method takes only one parameter, namely the text you want to search for.

When used with plain text, the `search()` method provides no real benefit over methods like `indexOf()`, which you've already seen. However, you'll see later that it's when you use regular expressions that the power of this method becomes apparent.

In the following example, you want to find out if the word `Java` is contained within the string called `myString`.

```
var myString = "Beginning JavaScript, Beginning Java, Professional JavaScript";  
alert(myString.search("Java"));
```

The alert box that occurs will show the value `10`, which is the character position of the `J` in the first occurrence of `Java`, as part of the word `JavaScript`.

The `match()` Method

The `match()` method is very similar to the `search()` method, except that instead of returning the position at which a match was found, it returns an array. Each element of the array contains the text of each match that is found.

Although you can use plain text with the `match()` method, it would be completely pointless to do so. For example, take a look at the following:

```
var myString = "1997, 1998, 1999, 2000, 2000, 2001, 2002";  
myMatchArray = myString.match("2000");  
alert(myMatchArray.length);
```

This code results in `myMatchArray` holding an element containing the value `2000`. Given that you already know your search string is `2000`, you can see it's been a pretty pointless exercise.

However, the `match()` method makes a lot more sense when we use it with regular expressions. Then you might search for all years in the twenty-first century—that is, those beginning with `2`. In this case, your array would contain the values `2000`, `2000`, `2001`, and `2002`, which is much more useful information!

Regular Expressions

Before you look at the `split()`, `match()`, `search()`, and `replace()` methods of the `String` object again, you need to look at regular expressions and the `RegExp` object. Regular expressions provide a means of defining a pattern of characters, which you can then use to split, search for, or replace characters in a string when they fit the defined pattern.

JavaScript's regular expression syntax borrows heavily from the regular expression syntax of Perl, another scripting language. The latest versions of languages, such as VBScript, have also incorporated regular expressions, as do lots of applications, such as Microsoft Word, in which the Find facility allows regular expressions to be used. The same is true for Dreamweaver. You'll find your regular expression knowledge will prove useful even outside JavaScript.

Regular expressions in JavaScript are used through the `RegExp` object, which is a native JavaScript object, as are `String`, `Array`, and so on. There are two ways of creating a new `RegExp` object. The easier is with a regular expression literal, such as the following:

```
var myRegExp = /\b'|\b/;
```

The forward slashes (/) mark the start and end of the regular expression. This is a special syntax that tells JavaScript that the code is a regular expression, much as quote marks define a string's start and end. Don't worry about the actual expression's syntax yet (the `\b'|\b`)—that will be explained in detail shortly.

Alternatively, you could use the `RegExp` object's constructor function `RegExp()` and type the following:

```
var myRegExp = new RegExp("\\b'|\\b");
```

Either way of specifying a regular expression is fine, though the former method is a shorter, more efficient one for JavaScript to use, and therefore generally preferred. For much of the remainder of the chapter, you'll use the first method. The main reason for using the second method is that it allows the regular expression to be determined at runtime (as the code is executing and not when you are writing the code). This is useful if, for example, you want to base the regular expression on user input.

Once you get familiar with regular expressions, you will come back to the second way of defining them, using the `RegExp()` constructor. As you can see, the syntax of regular expressions is slightly different with the second method, so we'll return to this subject later.

Although you'll be concentrating on the use of the `RegExp` object as a parameter for the `String` object's `split()`, `replace()`, `match()`, and `search()` methods, the `RegExp` object does have its own methods and properties. For example, the `test()` method enables you to test to see if the string passed to it as a parameter contains a pattern matching the one defined in the `RegExp` object. You'll see the `test()` method in use in an example shortly.

Simple Regular Expressions

Defining patterns of characters using regular expression syntax can get fairly complex. In this section you'll explore just the basics of regular expression patterns. The best way to do this is through examples.

Chapter 8: String Manipulation

Let's start by looking at an example in which you want to do a simple text replacement using the `replace()` method and a regular expression. Imagine you have the following string:

```
var myString = "Paul, Paula, Pauline, paul, Paul";
```

and you want to replace any occurrence of the name "Paul" with "Ringo."

Well, the pattern of text you need to look for is simply `Paul`. Representing this as a regular expression, you just have this:

```
var myRegExp = /Paul/;
```

As you saw earlier, the forward-slash characters mark the start and end of the regular expression. Now let's use this expression with the `replace()` method.

```
myString = myString.replace(myRegExp, "Ringo");
```

You can see that the `replace()` method takes two parameters: the `RegExp` object that defines the pattern to be searched and replaced, and the replacement text.

If you put this all together in an example, you have the following:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<body>
<script language="JavaScript" type="text/JavaScript">
  var myString = "Paul, Paula, Pauline, paul, Paul";
  var myRegExp = /Paul/;
  myString = myString.replace(myRegExp, "Ringo");
  alert(myString);
</script>
</body>
</html>
```

If you load this code into a browser, you will see the screen shown in Figure 8-3.

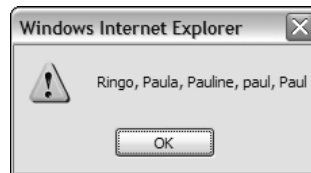


Figure 8-3

You can see that this has replaced the first occurrence of `Paul` in your string. But what if you wanted all the occurrences of `Paul` in the string to be replaced? The two at the far end of the string are still there, so what happened?

Well, by default the `RegExp` object looks only for the first matching pattern, in this case the first `Paul`, and then stops. This is a common and important behavior for `RegExp` objects. Regular expressions tend to start at one end of a string and look through the characters until the first complete match is found, then stop.

What you want is a global match, which is a search for all possible matches to be made and replaced. To help you out, the `RegExp` object has three attributes you can define. You can see these listed in the following table.

Attribute Character	Description
<code>g</code>	Global match. This looks for all matches of the pattern rather than stopping after the first match is found.
<code>i</code>	Pattern is case-insensitive. For example, <code>Paul</code> and <code>paul</code> are considered the same pattern of characters.
<code>m</code>	Multi-line flag. Only available in IE 5.5+ and NN 6+, this specifies that the special characters <code>^</code> and <code>\$</code> can match the beginning and the end of lines as well as the beginning and end of the string. You'll learn about these characters later in the chapter.

If you change our `RegExp` object in the code to the following, a global case-insensitive match will be made.

```
var myRegExp = /Paul/gi;
```

Running the code now produces the result shown in Figure 8-4.



Figure 8-4

This looks as if it has all gone horribly wrong. The regular expression has matched the `Paul` substrings at the start and the end of the string, and the penultimate `paul`, just as you wanted. However, the `Paul` substrings inside `Pauline` and `Paula` have also been replaced.

The `RegExp` object has done its job correctly. You asked for all patterns of the characters `Paul` to be replaced and that's what you got. What you actually meant was for all occurrences of `Paul`, when it's a single word and not part of another word, such as `Paula`, to be replaced. The key to making regular expressions work is to define exactly the pattern of characters you mean, so that only that pattern can match and no other. So let's do that.

1. You want `paul` or `Paul` to be replaced.
2. You don't want it replaced when it's actually part of another word, as in `Pauline`.

Chapter 8: String Manipulation

How do you specify this second condition? How do you know when the word is joined to other characters, rather than just joined to spaces or punctuation or the start or end of the string?

To see how you can achieve the desired result with regular expressions, you need to enlist the help of regular expression special characters. You'll look at these in the next section, by the end of which you should be able to solve the problem.

Regular Expressions: Special Characters

You will be looking at three types of special characters in this section.

Text, Numbers, and Punctuation

The first group of special characters you'll look at contains the character class's special characters. *Character class* means digits, letters, and whitespace characters. The special characters are displayed in the following table.

Character Class	Characters It Matches	Example
<code>\d</code>	Any digit from 0 to 9	<code>\d\d</code> matches 72, but not aa or 7a
<code>\D</code>	Any character that is not a digit	<code>\D\D\D</code> matches abc, but not 123 or 8ef
<code>\w</code>	Any word character; that is, A–Z, a–z, 0–9, and the underscore character (<code>_</code>)	<code>\w\w\w\w</code> matches Ab_2, but not £\$%* or Ab_@
<code>\W</code>	Any non-word character	<code>\W</code> matches @, but not a
<code>\s</code>	Any whitespace character, including tab, newline, carriage return, formfeed, and vertical tab	<code>\s</code> matches <i>tab</i>
<code>\S</code>	Any non-whitespace character	<code>\S</code> matches A, but not the tab character
<code>.</code>	Any single character other than the newline character (<code>\n</code>)	<code>.</code> matches a or 4 or @
<code>[. . .]</code>	Any one of the characters between the brackets	<code>[abc]</code> will match a or b or c, but nothing else <code>[a–z]</code> will match any character in the range a to z
<code>[^ . . .]</code>	Any one character, but not one of those inside the brackets	<code>[^abc]</code> will match any character except a or b or c <code>[^a–z]</code> will match any character that is not in the range a to z

Note that uppercase and lowercase characters mean very different things, so you need to be extra careful with case when using regular expressions.

Let's look at an example. To match a telephone number in the format 1-800-888-5474, the regular expression would be as follows:

```
\d-\d\d\d\d-\d\d\d\d-\d\d\d\d
```

You can see that there's a lot of repetition of characters here, which makes the expression quite unwieldy. To make this simpler, regular expressions have a way of defining repetition. You'll see this a little later in the chapter, but first let's look at another example.

Try It Out Checking a Passphrase for Alphanumeric Characters

You'll use what you've learned so far about regular expressions in a full example in which you check that a passphrase contains only letters and numbers — that is, alphanumeric characters, and not punctuation or symbols like @, %, and so on.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Example</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
</head>
<body>
<script language="JavaScript" type="text/JavaScript">
function regExpIs_valid(text)
{
    var myRegExp = /^[a-z\d ]/i;
    return !(myRegExp.test(text));
}
function butCheckValid_onclick()
{
    if (regExpIs_valid(document.form1.txtPhrase.value) == true)
    {
        alert("Your passphrase contains only valid characters");
    }
    else
    {
        alert("Your passphrase contains one or more invalid characters");
    }
}
</script>
<form name=form1>
Enter your passphrase:
<br>
<input type="text" name=txtPhrase>
<br>
<input type="button" value="Check Character Validity" name=butCheckValid
    onclick="butCheckValid_onclick()">
</form>
</body>
</html>
```

Chapter 8: String Manipulation

Save the page as `ch8_exam2.htm`, and then load it into your browser. Type just letters, numbers, and spaces into the text box; click the Check Character Validity button; and you'll be told that the phrase contains valid characters. Try putting punctuation or special characters like `@`, `^`, `$`, and so on into the text box, and you'll be informed that your passphrase is invalid.

How It Works

Let's start by looking at the `regExpIs_valid()` function defined at the top of the script block in the head of the page. That does the validity checking of our passphrase using regular expressions.

```
function RegExpIs_valid(text)
{
    var myRegExp = /^[^a-z\d ]/i;
    return !(myRegExp.test(text));
}
```

The function takes just one parameter: the text you want to check for validity. You then declare a variable, `myRegExp`, and set it to a new regular expression, which implicitly creates a new `RegExp` object.

The regular expression itself is fairly simple, but first let's think about what pattern you are looking for. What you want to find out is whether your passphrase string contains any characters that are not letters between A and Z or between a and z, numbers between 0 and 9, or spaces. Let's see how this translates into a regular expression.

First you use square brackets with the `^` symbol.

```
[^]
```

This means you want to match any character that is not one of the characters specified inside the square brackets. Next you add `a-z`, which specifies any character in the range a through z.

```
[^a-z]
```

So far your regular expression matches any character that is not between a and z. Note that, because you added the `i` to the end of the expression definition, you've made the pattern case-insensitive. So our regular expression actually matches any character not between A and Z or a and z.

Next you add `\d` to indicate any digit character, or any character between 0 and 9.

```
[^a-z\d]
```

So your expression matches any character that is not between a and z, A and Z, or 0 and 9. Finally, you decide that a space is valid, so you add that inside the square brackets.

```
[^a-z\d ]
```

Putting this all together, you have a regular expression that will match any character that is not a letter, a digit, or a space.

On the second and final line of the function you use the `RegExp` object's `test()` method to return a value.

```
return !(myRegExp.test(text));
```

The `test()` method of the `RegExp` object checks the string passed as its parameter to see if the characters specified by the regular expression syntax match anything inside the string. If they do, `true` is returned; if not, `false` is returned. Your regular expression will match the first invalid character found, so if you get a result of `true`, you have an invalid passphrase. However, it's a bit illogical for an `is_valid` function to return `true` when it's invalid, so you reverse the result returned by adding the NOT operator (`!`).

Previously you saw the two-line validity checker function using regular expressions. Just to show how much more coding is required to do the same thing without regular expressions, here is a second function that does the same thing as `regExpIs_valid()` but without regular expressions.

```
function is_valid(text)
{
    var isValid = true;
    var validChars = "abcdefghijklmnopqrstuvwxyz1234567890 ";
    var charIndex;
    for (charIndex = 0; charIndex < text.length; charIndex++)
    {
        if ( validChars.indexOf(text.charAt(charIndex).toLowerCase()) < 0)
        {
            isValid = false;
            break;
        }
    }
    return isValid;
}
```

This is probably as small as the non-regular expression version can be, and yet it's still 15 lines long. That's six times the amount of code for the regular expression version.

The principle of this function is similar to that of the regular expression version. You have a variable, `validChars`, which contains all the characters you consider to be valid. You then use the `charAt()` method in a `for` loop to get each character in the passphrase string and check whether it exists in your `validChars` string. If it doesn't, you know you have an invalid character.

In this example, the non-regular expression version of the function is 15 lines, but with a more complex problem you could find it takes 20 or 30 lines to do the same thing a regular expression can do in just a few.

Back to your actual code: The other function defined in the head of the page is `butCheckValid_onclick()`. As the name suggests, this is called when the `butCheckValid` button defined in the body of the page is clicked.

This function calls your `regExpIs_valid()` function in an `if` statement to check whether the passphrase entered by the user in the `txtPhrase` text box is valid. If it is, an alert box is used to inform the user.

```
function butCheckValid_onclick()
{
    if (regExpIs_valid(document.form1.txtPhrase.value) == true)
    {
        alert("Your passphrase contains valid characters");
    }
}
```

Chapter 8: String Manipulation

If it isn't, another alert box is used to let the user know that his text was invalid.

```
    else
    {
        alert("Your passphrase contains one or more invalid characters");
    }
}
```

Repetition Characters

Regular expressions include something called repetition characters, which are a means of specifying how many of the last item or character you want to match. This proves very useful, for example, if you want to specify a phone number that repeats a character a specific number of times. The following table lists some of the most common repetition characters and what they do.

Special Character	Meaning	Example
{n}	Match n of the previous item	x{2} matches xx
{n, }	Match n or more of the previous item	x{2, } matches xx, xxx, xxxx, xxxxx, and so on
{n,m}	Match at least n and at most m of the previous item	x{2, 4} matches xx, xxx, and xxxx
?	Match the previous item zero or one time	x? matches nothing or x
+	Match the previous item one or more times	x+ matches x, xx, xxx, xxxx, xxxxx, and so on
*	Match the previous item zero or more times	x* matches nothing, or x, xx, xxx, xxxxx, and so on

You saw earlier that to match a telephone number in the format 1-800-888-5474, the regular expression would be `\d-\d\d\d\d-\d\d\d\d-\d\d\d\d\d`. Let's see how this would be simplified with the use of the repetition characters.

The pattern you're looking for starts with one digit followed by a dash, so you need the following:

```
\d-
```

Next are three digits followed by a dash. This time you can use the repetition special characters — `\d{3}` will match exactly three `\d`, which is the any-digit character.

```
\d-\d{3}-
```

Next there are three digits followed by a dash again, so now your regular expression looks like this:

```
\d-\d{3}-\d{3}-
```

Finally, the last part of the expression is four digits, which is `\d{4}`.

```
\d-\d{3}-\d{3}-\d{4}
```

You'd declare this regular expression like this:

```
var myRegExp = /\d-\d{3}-\d{3}-\d{4}/
```

Remember that the first / and last / tell JavaScript that what is in between those characters is a regular expression. JavaScript creates a `RegExp` object based on this regular expression.

As another example, what if you have the string `Paul Paula Pauline`, and you want to replace `Paul` and `Paula` with `George`? To do this, you would need a regular expression that matches both `Paul` and `Paula`.

Let's break this down. You know you want the characters `Paul`, so your regular expression starts as

```
Paul
```

Now you also want to match `Paula`, but if you make your expression `Paula`, this will exclude a match on `Paul`. This is where the special character `?` comes in. It enables you to specify that the previous character is optional — it must appear zero (not at all) or one time. So, the solution is

```
Paula?
```

which you'd declare as

```
var myRegExp = /Paula?/
```

Position Characters

The third group of special characters you'll look at are those that enable you to specify either where the match should start or end or what will be on either side of the character pattern. For example, you might want your pattern to exist at the start or end of a string or line, or you might want it to be between two words. The following table lists some of the most common position characters and what they do.

Position Character	Description
<code>^</code>	The pattern must be at the start of the string, or if it's a multi-line string, then at the beginning of a line. For multi-line text (a string that contains carriage returns), you need to set the multi-line flag when defining the regular expression using <code>/myreg ex/m</code> . Note that this is only applicable to IE 5.5 and later and NN 6 and later.
<code>\$</code>	The pattern must be at the end of the string, or if it's a multi-line string, then at the end of a line. For multi-line text (a string that contains carriage returns), you need to set the multi-line flag when defining the regular expression using <code>/myreg ex/m</code> . Note that this is only applicable to IE 5.5 and later and NN 6 and later.
<code>\b</code>	This matches a word boundary, which is essentially the point between a word character and a non-word character.
<code>\B</code>	This matches a position that's not a word boundary.

Chapter 8: String Manipulation

For example, if you wanted to make sure your pattern was at the start of a line, you would type the following:

```
^myPattern
```

This would match an occurrence of `myPattern` if it was at the beginning of a line.

To match the same pattern, but at the end of a line, you would type the following:

```
myPattern$
```

The word-boundary special characters `\b` and `\B` can cause confusion, because they do not match characters but the positions between characters.

Imagine you had the string `"Hello world!, let's look at boundaries said 007."` defined in the code as follows:

```
var myString = "Hello world!, let's look at boundaries said 007.";
```

To make the word boundaries (that is, the boundaries between the words) of this string stand out, let's convert them to the `|` character.

```
var myRegExp = /\b/g;
myString = myString.replace(myRegExp, "|");
alert(myString);
```

You've replaced all the word boundaries, `\b`, with a `|`, and your message box looks like the one in Figure 8-5.

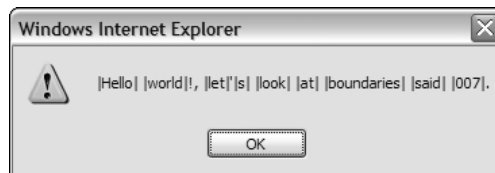


Figure 8-5

You can see that the position between any word character (letters, numbers, or the underscore character) and any non-word character is a word boundary. You'll also notice that the boundary between the start or end of the string and a word character is considered to be a word boundary. The end of this string is a full stop. So the boundary between the full stop and the end of the string is a non-word boundary, and therefore no `|` has been inserted.

If you change the regular expression in the example, so that it replaces non-word boundaries as follows:

```
var myRegExp = /\B/g;
```

you get the result shown in Figure 8-6.

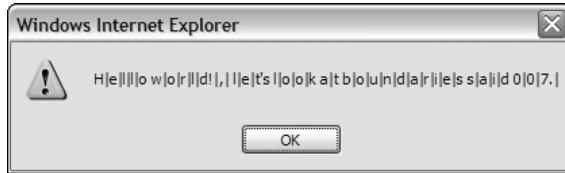


Figure 8-6

Now the position between a letter, number, or underscore and another letter, number, or underscore is considered a non-word boundary and is replaced by an | in our example. However, what is slightly confusing is that the boundary between two non-word characters, such as an exclamation mark and a comma, is also considered a non-word boundary. If you think about it, it actually does make sense, but it's easy to forget when creating regular expressions.

You'll remember this example from when we started looking at regular expressions:

```
<html>
<body>
<script language="JavaScript" type="text/JavaScript">
  var myString = "Paul, Paula, Pauline, paul, Paul";
  var myRegExp = /Paul/gi;
  myString = myString.replace(myRegExp, "Ringo");
  alert(myString);
</script>
</body>
</html>
```

We used this code to convert all instances of Paul or paul to Ringo.

However, we found that this code actually converts all instances of Paul to Ringo, even when the word Paul is inside another word.

One way to solve this problem would be to replace the string Paul only where it is followed by a non-word character. The special character for non-word characters is \W, so you need to alter our regular expression to the following:

```
var myRegExp = /Paul\W/gi;
```

This gives the result shown in Figure 8-7.

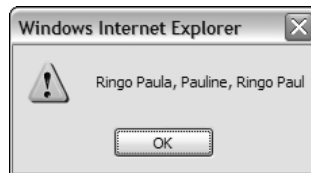


Figure 8-7

Chapter 8: String Manipulation

It's getting better, but it's still not what you want. Notice that the commas after the second and third Paul substrings have also been replaced because they matched the `\w` character. Also, you're still not replacing Paul at the very end of the string. That's because there is no character after the letter l in the last Paul. What is after the l in the last Paul? Nothing, just the boundary between a word character and a non-word character, and therein lies the answer. What you want as your regular expression is Paul followed by a word boundary. Let's alter the regular expression to cope with that by entering the following:

```
var myRegExp = /Paul\b/gi;
```

Now you get the result you want, as shown in Figure 8-8.

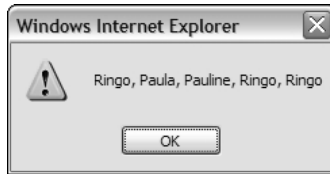


Figure 8-8

At last you've got it right, and this example is finished.

Covering All Eventualities

Perhaps the trickiest thing about a regular expression is making sure it covers all eventualities. In the previous example your regular expression works with the string as defined, but does it work with the following?

```
var myString = "Paul, Paula, Pauline, paul, Paul, JeanPaul";
```

Here the Paul substring in JeanPaul will be changed to Ringo. You really only want to convert the substring Paul where it is on its own, with a word boundary on either side. If you change your regular expression code to

```
var myRegExp = /\bPaul\b/gi;
```

you have your final answer and can be sure only Paul or paul will ever be matched.

Grouping Regular Expressions

The final topic under regular expressions, before we look at examples using the `match()`, `replace()`, and `search()` methods, is how you can group expressions. In fact it's quite easy. If you want a number of expressions to be treated as a single group, you just enclose them in parentheses, for example `/(\d\d)/`. Parentheses in regular expressions are special characters that group together character patterns and are not themselves part of the characters to be matched.

The question is, Why would you want to do this? Well, by grouping characters into patterns, you can use the special repetition characters to apply to the whole group of characters, rather than just one.

Let's take the following string defined in `myString` as an example:

```
var myString = "JavaScript, VBScript and Perl";
```

How could you match both `JavaScript` and `VBScript` using the same regular expression? The only thing they have in common is that they are whole words and they both end in `Script`. Well, an easy way would be to use parentheses to group the patterns `Java` and `VB`. Then you can use the `?` special character to apply to each of these groups of characters to make the pattern match any word having zero or one instances of the characters `Java` or `VB`, and ending in `Script`.

```
var myRegExp = /\b(VB)?(Java)?Script\b/gi;
```

Breaking this expression down, you can see the pattern it requires is as follows:

1. A word boundary: `\b`
2. Zero or one instance of `VB`: `(VB)?`
3. Zero or one instance of `Java`: `(Java)?`
4. The characters `Script`: `Script`
5. A word boundary: `\b`

Putting these together, you get this:

```
var myString = "JavaScript, VBScript and Perl";  
var myRegExp = /\b(VB)?(Java)?Script\b/gi;  
myString = myString.replace(myRegExp, "xxxx");  
alert(myString);
```

The output of this code is shown in Figure 8-9.

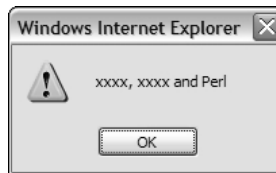


Figure 8-9

If you look back at the special repetition characters table, you'll see that they apply to the item preceding them. This can be a character, or, where they have been grouped by means of parentheses, the previous group of characters.

However, there is a potential problem with the regular expression you just defined. As well as matching `VBScript` and `JavaScript`, it also matches `VBJavaScript`. This is clearly not exactly what you meant.

To get around this you need to make use of both grouping and the special character `|`, which is the alternation character. It has an or-like meaning, similar to `||` in `if` statements, and will match the characters on either side of itself.

Chapter 8: String Manipulation

Let's think about the problem again. You want the pattern to match `VBScript` or `JavaScript`. Clearly they have the `Script` part in common. So what you want is a new word starting with `Java` or starting with `VB`; either way it must end in `Script`.

First, you know that the word must start with a word boundary.

```
\b
```

Next you know that you want either `VB` or `Java` to be at the start of the word. You've just seen that in regular expressions `|` provides the "or" you need, so in regular expression syntax you want the following:

```
\b(VB|Java)
```

This matches the pattern `VB` or `Java`. Now you can just add the `Script` part.

```
\b(VB|Java)Script\b
```

Your final code looks like this:

```
var myString = "JavaScript, VBScript and Perl";
var myRegExp = /\b(VB|Java)Script\b/gi;
myString = myString.replace(myRegExp, "xxxx");
alert(myString);
```

Reusing Groups of Characters

You can reuse the pattern specified by a group of characters later on in our regular expression. To refer to a previous group of characters, you just type `\` and a number indicating the order of the group. For example, the first group can be referred to as `\1`, the second as `\2`, and so on.

Let's look at an example. Say you have a list of numbers in a string, with each number separated by a comma. For whatever reason, you are not allowed to have two instances of the same number in a row, so although

```
009,007,001,002,004,003
```

would be okay, the following:

```
007,007,001,002,002,003
```

would not be valid, because you have `007` and `002` repeated after themselves.

How can you find instances of repeated digits and replace them with the word `ERROR`? You need to use the ability to refer to groups in regular expressions.

First let's define the string as follows:

```
var myString = "007,007,001,002,002,003,002,004";
```

Now you know you need to search for a series of one or more number characters. In regular expressions the `\d` specifies any digit character, and `+` means one or more of the previous character. So far, that gives you this regular expression:

```
\d+
```

You want to match a series of digits followed by a comma, so you just add the comma.

```
\d+,
```

This will match any series of digits followed by a comma, but how do you search for any series of digits followed by a comma, then followed again by the same series of digits? As the digits could be any digits, you can't add them directly into our expression like so:

```
\d+,007
```

This would not work with the `002` repeat. What you need to do is put the first series of digits in a group; then you can specify that you want to match that group of digits again. This can be done with `\1`, which says, "Match the characters found in the first group defined using parentheses." Put all this together, and you have the following:

```
(\d+),\1
```

This defines a group whose pattern of characters is one or more digit characters. This group must be followed by a comma and then by the same pattern of characters as in the first group. Put this into some JavaScript, and you have the following:

```
var myString = "007,007,001,002,002,003,002,004";  
var myRegExp = /(\d+),\1/g;  
myString = myString.replace(myRegExp, "ERROR");  
alert(myString);
```

The alert box will show this message:

```
ERROR,1,ERROR,003,002,004
```

That completes your brief look at regular expression syntax. Because regular expressions can get a little complex, it's often a good idea to start simple and build them up slowly, as we have done here. In fact, most regular expressions are just too hard to get right in one step—at least for us mere mortals without a brain the size of a planet.

If it's still looking a bit strange and confusing, don't panic. In the next sections, you'll be looking at the `String` object's `split()`, `replace()`, `search()`, and `match()` methods with plenty more examples of regular expression syntax.

The String Object — `split()`, `replace()`, `search()`, and `match()` Methods

The main functions making use of regular expressions are the `String` object's `split()`, `replace()`, `search()`, and `match()` methods. You've already seen their syntax, so you'll concentrate on their use with regular expressions and at the same time learn more about regular expression syntax and usage.

The `split()` Method

You've seen that the `split()` method enables us to split a string into various pieces, with the split being made at the character or characters specified as a parameter. The result of this method is an array with each element containing one of the split pieces. For example, the following string:

```
var myListString = "apple, banana, peach, orange"
```

could be split into an array in which each element contains a different fruit, like this:

```
var myFruitArray = myListString.split(", ");
```

How about if your string is this instead?

```
var myListString = "apple, 0.99, banana, 0.50, peach, 0.25, orange, 0.75";
```

The string could, for example, contain both the names and prices of the fruit. How could you split the string, but retrieve only the names of the fruit and not the prices? You could do it without regular expressions, but it would take many lines of code. With regular expressions you can use the same code, and just amend the `split()` method's parameter.

Try It Out Splitting the Fruit String

Let's create an example that solves the problem just described — it must split your string, but include only the fruit names, not the prices.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<body>
<script language="JavaScript" type="text/JavaScript">
var myListString = "apple, 0.99, banana, 0.50, peach, 0.25, orange, 0.75";
var theRegExp = /^[^a-z]+/i;
var myFruitArray = myListString.split(theRegExp);
document.write(myFruitArray.join("<br>"));
</script>
</body>
</html>
```

Save the file as `ch8_examp3.htm` and load it in your browser. You should see the four fruits from your string written out to the page, with each fruit on a separate line.

How It Works

Within the script block, first you have your string with fruit names and prices.

```
var myListString = "apple, 0.99, banana, 0.50, peach, 0.25, orange, 0.75";
```

How do you split it in such a way that only the fruit names are included? Your first thought might be to use the comma as the `split()` method's parameter, but of course that means you end up with the prices. What you have to ask is, "What is it that's between the items I want?" Or in other words, what is between the fruit names that you can use to define your split? The answer is that various characters are between the names of the fruit, such as a comma, a space, numbers, a full stop, more numbers, and finally another comma. What is it that these things have in common and makes them different from the fruit names that you want? What they have in common is that none of them are letters from a through z. If you say "Split the string at the point where there is a group of characters that are not between a and z," then you get the result you want. Now you know what you need to create your regular expression.

You know that what you want is not the letters a through z, so you start with this:

```
[^a-z]
```

The `^` says "Match any character that does not match those specified inside the square brackets." In this case you've specified a range of characters not to be matched — all the characters between a and z. As specified, this expression will match only one character, whereas you want to split wherever there is a single group of one or more characters that are not between a and z. To do this you need to add the `+` special repetition character, which says "Match one or more of the preceding character or group specified."

```
[^a-z] +
```

The final result is this:

```
var theRegExp = /^[^a-z] +/i
```

The `/` and `/` characters mark the start and end of the regular expression whose `RegExp` object is stored as a reference in the variable `theRegExp`. You add the `i` on the end to make the match case-insensitive.

Don't panic if creating regular expressions seems like a frustrating and less-than-obvious process. At first, it takes a lot of trial and error to get it right, but as you get more experienced, you'll find creating them becomes much easier and will enable you to do things that without regular expressions would be either very awkward or virtually impossible.

In the next line of script you pass the `RegExp` object to the `split()` method, which uses it to decide where to split the string.

```
var myFruitArray = myListString.split(theRegExp);
```

After the split, the variable `myFruitArray` will contain an `Array` with each element containing the fruit name, as shown here:

Chapter 8: String Manipulation

Array Element Index	0	1	2	3
Element value	apple	banana	peach	orange

You then join the string together again using the `Array` object's `join()` methods, which you saw in Chapter 4.

```
document.write(myFruitArray.join("<BR>"))
```

The replace() Method

You've already looked at the syntax and usage of the `replace()` method. However, something unique to the `replace()` method is its ability to replace text based on the groups matched in the regular expression. You do this using the `$` sign and the group's number. Each group in a regular expression is given a number from 1 to 99; any groups greater than 99 are not accessible. Note that in earlier browsers, groups could only go from 1 to 9 (for example, in IE 5 or earlier or Netscape 4 and earlier). To refer to a group, you write `$` followed by the group's position. For example, if you had the following:

```
var myRegExp = /(\d)(\W)/g;
```

then `$1` refers to the group `(\d)`, and `$2` refers to the group `(\W)`. You've also set the global flag `g` to ensure that all matching patterns are replaced — not just the first one.

You can see this more clearly in the next example. Say you have the following string:

```
var myString = "1999, 2000, 2001";
```

If you wanted to change this to "the year 1999, the year 2000, the year 2001", how could you do it with regular expressions?

First you need to work out the pattern as a regular expression, in this case four digits.

```
var myRegExp = /\d{4}/g;
```

But given that the year is different every time, how can you substitute the year value into the replaced string?

Well, you change your regular expression so that it's inside a group, as follows:

```
var myRegExp = /(\d{4})/g;
```

Now you can use the group, which has group number 1, inside the replacement string like this:

```
myString = myString.replace(myRegExp, "the year $1");
```

The variable `myString` now contains the required string "the year 1999, the year 2000, the year 2001".

Let's look at another example in which you want to convert single quotes in text to double quotes. Your test string is this:

```
'Hello World' said Mr. O'Connerly.  
He then said 'My Name is O'Connerly, yes that's right, O'Connerly'.
```

One problem that the test string makes clear is that you want to replace the single-quote mark with a double only where it is used in pairs around speech, not when it is acting as an apostrophe, such as in the word *that's*, or when it's part of someone's name, such as in *O'Connerly*.

Let's start by defining the regular expression. First you know that it must include a single quote, as shown in the following code:

```
var myRegExp = '/';
```

However, as it is this would replace every single quote, which is not what you want.

Looking at the text, you should also notice that quotes are always at the start or end of a word — that is, at a boundary. On first glance it might be easy to assume that it would be a word boundary. However, don't forget that the `'` is a non-word character, so the boundary will be between it and another non-word character, such as a space. So the boundary will be a non-word boundary, or in other words, `\B`.

Therefore, the character pattern you are looking for is either a non-word boundary followed by a single quote, or a single quote followed by a non-word boundary. The key is the “or,” for which you use `|` in regular expressions. This leaves your regular expression as the following:

```
var myRegExp = /\B'|'\B/g;
```

This will match the pattern on the left of the `|` or the character pattern on the right. You want to replace all the single quotes with double quotes, so the `g` has been added at the end, indicating that a global match should take place.

Try It Out Replacing Single Quotes with Double Quotes

Let's look at an example using the regular expression just defined.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"  
"http://www.w3.org/TR/html4/loose.dtd">  
<html>  
<head>  
<title>example</title>  
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">  
<script language="JavaScript" type="text/JavaScript">  
function replaceQuote(textAreaControl)  
{  
    var myText = textAreaControl.value;  
    var myRegExp = /\B'|'\B/g;  
    myText = myText.replace(myRegExp, '"');  
    textAreaControl.value = myText;  
}  
</script>  
</head>
```

Chapter 8: String Manipulation

```
<body>
<form name="form1">
<textarea rows="20" cols="40" name="textareal">
'Hello World' said Mr O'Connerly.
He then said 'My Name is O'Connerly, yes that's right, O'Connerly'.
</textarea>
<br>
<input type="button" VALUE="Replace Single Quotes" name="buttonSplit"
onclick="replaceQuote(document.form1.textareal) ">
</form>
</body>
</html>
```

Save the page as `ch8_examp4.htm`. Load the page into your browser and you should see what is shown in Figure 8-10.

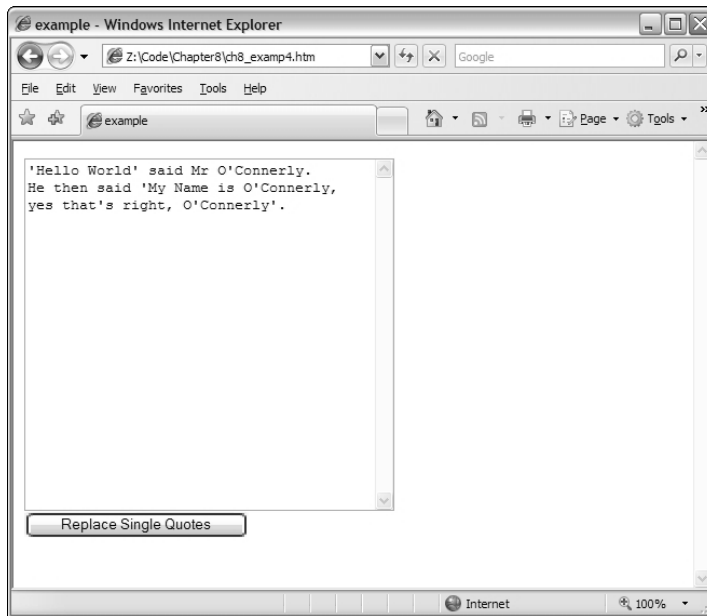


Figure 8-10

Click the Replace Single Quotes button to see the single quotes in the text area replaced as in Figure 8-11.

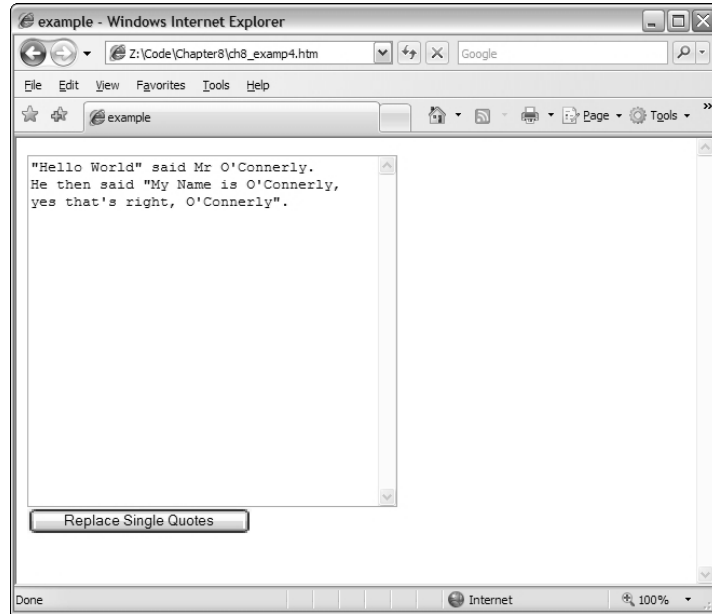


Figure 8-11

Try entering your own text with single quotes into the text area and check the results.

How It Works

You can see that by using regular expressions, you have completed a task in a couple of lines of simple code. Without regular expressions, it would probably take four or five times that amount.

Let's look first at the `replaceQuote()` function in the head of the page where all the action is.

```
function replaceQuote(textAreaControl)
{
    var myText = textAreaControl.value;
    var myRegExp = /\B'|'\B/g;
    myText = myText.replace(myRegExp, '');
    textAreaControl.value = myText;
}
```

The function's parameter is the `textarea` object defined further down the page—this is the text area in which you want to replace the single quotes. You can see how the `textarea` object was passed in the button's tag definition.

```
<input type="button" value="Replace Single Quotes" name="buttonSplit"
    onclick="replaceQuote(document.form1.textarea1)">
```

In the `onclick` event handler, you call `replaceQuote()` and pass `document.form1.textarea1` as the parameter—that is the `textarea` object.

Chapter 8: String Manipulation

Returning to the function, you get the value of the `textarea` on the first line and place it in the variable `myText`. Then you define your regular expression (as discussed previously), which matches any non-word boundary followed by a single quote or any single quote followed by a non-word boundary. For example, `'H` will match, as will `H'`, but `O'R` won't because the quote is between two word boundaries. Don't forget that a word boundary is the position between the start or end of a word and a non-word character, such as a space or punctuation mark.

In the function's final two lines, you first use the `replace()` method to do the character pattern search and replace, and finally you set the `textarea` object's value to the changed string.

The `search()` Method

The `search()` method enables you to search a string for a pattern of characters. If the pattern is found, the character position at which it was found is returned, otherwise `-1` is returned. The method takes only one parameter, the `RegExp` object you have created.

Although for basic searches the `indexOf()` method is fine, if you want more complex searches, such as a search for a pattern of any digits or one in which a word must be in between a certain boundary, then `search()` provides a much more powerful and flexible, but sometimes more complex, approach.

In the following example, you want to find out if the word `Java` is contained within the string. However, you want to look just for `Java` as a whole word, not part of another word such as `JavaScript`.

```
var myString = "Beginning JavaScript, Beginning Java 2, Professional JavaScript";
var myRegExp = /\bJava\b/i;
alert(myString.search(myRegExp));
```

First you have defined your string, and then you've created your regular expression. You want to find the character pattern `Java` when it's on its own between two word boundaries. You've made your search case-insensitive by adding the `i` after the regular expression. Note that with the `search()` method, the `g` for global is not relevant, and its use has no effect.

On the final line you output the position at which the search has located the pattern, in this case `32`.

The `match()` Method

The `match()` method is very similar to the `search()` method, except that instead of returning the position at which a match was found, it returns an array. Each element of the array contains the text of a match made.

For example, if you had the string

```
var myString = "The years were 1999, 2000 and 2001";
```

and wanted to extract the years from this string, you could do so using the `match()` method. To match each year, you are looking for four digits in between word boundaries. This requirement translates to the following regular expression:

```
var myRegExp = /\b\d{4}\b/g;
```

You want to match all the years so the `g` has been added to the end for a global search.

To do the match and store the results, you use the `match()` method and store the `Array` object it returns in a variable.

```
var resultsArray = myString.match(myRegExp);
```

To prove it has worked, let's use some code to output each item in the array. You've added an `if` statement to double-check that the results array actually contains an array. If no matches were made, the results array will contain `null`—doing `if (resultsArray)` will return `true` if the variable has a value and not `null`.

```
if (resultsArray)
{
    var indexCounter;
    for (indexCounter = 0; indexCounter < resultsArray.length; indexCounter++)
    {
        alert(resultsArray[indexCounter]);
    }
}
```

This would result in three alert boxes containing the numbers 1999, 2000, and 2001.

Try It Out Splitting HTML

In the next example, you want to take a string of HTML and split it into its component parts. For example, you want the HTML `<P>Hello</P>` to become an array, with the elements having the following contents:

<P>	Hello	</P>
-----	-------	------

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>example</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<script language="JavaScript" type="text/JavaScript">
function button1_onclick()
{
    var myString = "<table align=center><tr><td>";
    myString = myString + "Hello World</td></tr></table>";
    myString = myString + "<br><h2>Heading</h2>";
    var myRegExp = /<[^>\r\n]+>|<[^<\\r\\n]+/g;
    var resultsArray = myString.match(myRegExp);
    document.form1.textarea1.value = "";
    document.form1.textarea1.value = resultsArray.join ("\r\n");
}
</script>
</head>
<body>
```

Chapter 8: String Manipulation

```
<form name="form1">
  <textarea rows="20" cols="40" name="textareal"></textarea>
  <input type="button" value="Split HTML" name="button1"
    onclick="return button1_onclick();" />
</form>
</body>
</html>
```

Save this file as `ch8_examp5.htm`. When you load the page into your browser and click the Split HTML button, a string of HTML is split, and each tag is placed on a separate line in the text area, as shown in Figure 8-12.

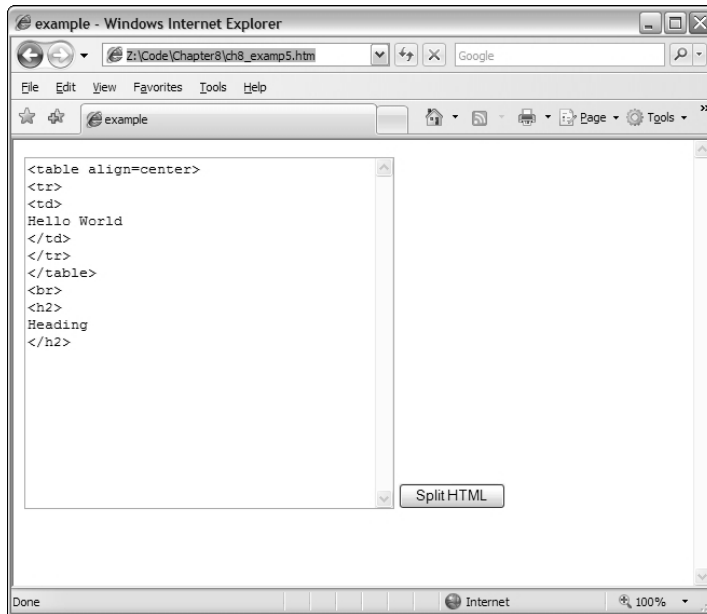


Figure 8-12

How It Works

The function `button1_onclick()` defined at the top of the page fires when the Split HTML button is clicked. At the top, the following lines define the string of HTML that you want to split:

```
function button1_onclick()
{
  var myString = "<table align=center><tr><td>";
  myString = myString + "Hello World</td></tr></table>";
  myString = myString + "<br><h2>Heading</h2>";
}
```

Next you create your `RegExp` object and initialize it to your regular expression.

```
var myRegExp = /<[^>\r\n]+>|[\^<>\r\n]+/g;
```

Let's break it down to see what pattern you're trying to match. First note that the pattern is broken up by an alternation symbol: `|`. This means that you want the pattern on the left or the right of this symbol. You'll look at these patterns separately. On the left you have the following:

- ❑ The pattern must start with a `<`.
- ❑ In `[^>\r\n]+`, you specify that you want one or more of any character except the `>` or a `\r` (carriage return) or a `\n` (linefeed).
- ❑ `>` specifies that the pattern must end with a `>`.

On the right, you have only the following:

- ❑ `[^<>\r\n]+` specifies that the pattern is one or more of any character, so long as that character is not a `<`, `>`, `\r`, or `\n`. This will match plain text.

After the regular expression definition you have a `g`, which specifies that this is a global match.

So the `<[^\r\n]+>` regular expression will match any start or close tags, such as `<p>` or `</p>`. The alternative pattern is `[^<>\r\n]+`, which will match any character pattern that is not an opening or closing tag.

In the following line you assign the `resultsArray` variable to the `Array` object returned by the `match()` method:

```
var resultsArray = myString.match(myRegExp);
```

The remainder of the code deals with populating the text area with the split HTML. You use the `Array` object's `join()` method to join all the array's elements into one string with each element separated by a `\r\n` character, so that each tag or piece of text goes on a separate line, as shown in the following:

```
document.form1.textarea1.value = "";
document.form1.textarea1.value = resultsArray.join("\r\n");
}
```

Using the RegExp Object's Constructor

So far you've been creating `RegExp` objects using the `/` and `/` characters to define the start and end of the regular expression, as shown in the following example:

```
var myRegExp = /[a-z]/;
```

Although this is the generally preferred method, it was briefly mentioned that a `RegExp` object can also be created by means of the `RegExp()` constructor. You might use the first way most of the time. However, there are occasions, as you'll see in the trivia quiz shortly, when the second way of creating a `RegExp` object is necessary (for example, when a regular expression is to be constructed from user input).

As an example, the preceding regular expression could equally well be defined as

```
var myRegExp = new RegExp("[a-z]");
```

Chapter 8: String Manipulation

Here you pass the regular expression as a string parameter to the `RegExp()` constructor function.

A very important difference when you are using this method is in how you use special regular expression characters, such as `\b`, that have a backward slash in front of them. The problem is that the backward slash indicates an escape character in JavaScript strings—for example, you may use `\b`, which means a backspace. To differentiate between `\b` meaning a backspace in a string and the `\b` special character in a regular expression, you have to put another backward slash in front of the regular expression special character. So `\b` becomes `\\b` when you mean the regular expression `\b` that matches a word boundary, rather than a backspace character.

For example, say you have defined your `RegExp` object using the following:

```
var myRegExp = /\b/;
```

To declare it using the `RegExp()` constructor, you would need to write this:

```
var myRegExp = new RegExp("\\b");
```

and not this:

```
var myRegExp = new RegExp("\b");
```

All special regular expression characters, such as `\w`, `\b`, `\d`, and so on, must have an extra `\` in front when you create them using `RegExp()`.

When you defined regular expressions with the `/` and `/` method, you could add after the final `/` the special flags `m`, `g`, and `i` to indicate that the pattern matching should be multi-line, global, or case-insensitive, respectively. When using the `RegExp()` constructor, how can you do the same thing?

Easy. The optional second parameter of the `RegExp()` constructor takes the flags that specify a global or case-insensitive match. For example, this will do a global case-insensitive pattern match:

```
var myRegExp = new RegExp("hello\\b", "gi");
```

You can specify just one of the flags if you wish—such as the following:

```
var myRegExp = new RegExp("hello\\b", "i");
```

or

```
var myRegExp = new RegExp("hello\\b", "g");
```

The Trivia Quiz

The goal for the trivia quiz in this chapter is to enable it to set questions with answers that have to be typed in by the user, in addition to the multiple-choice questions you already have. To do this you'll be making use of your newfound knowledge of regular expressions to search the reply that the user types in for a match with the correct answer.

The problem you face with text answers is that a number of possible answers may be correct and you don't want to annoy the user by insisting on only *one* specific version. For example, the answer to the question "Which president was involved in the Watergate scandal?" is Richard Milhous Nixon. However, most people will type Nixon, or maybe Richard Nixon or even R Nixon. Each of these variations is valid, and using regular expressions you can easily check for all of them (or at least many plausible alternatives) in just a few lines of code.

What will you need to change to add this extra functionality? In fact changes are needed in only two pages: the `GlobalFunctions.htm` page and the `AskQuestion.htm` page.

In the `GlobalFunctions.htm` page, you need to define your new questions and answers, change the `getQuestion()` function, which builds up the HTML to display the question to the user, and change the `answerCorrect()` function, which checks whether the user's answer is correct.

In the `AskQuestion.htm` page, you need to change the function `getAnswer()`, which retrieves the user's answer from the page's form.

You'll start by making the changes to `GlobalFunctions.htm` that you created in the last chapter, so open this up in your HTML editor.

All the existing multiple-choice questions that you define near the top of the page can remain in exactly the same format, so there's no need for any changes there. How can this be if you're using regular expressions?

Previously you checked to see that the answer the user selected, such as A, B, C, and so on, was equal to the character in the `answers` array. Well, you can do the same thing here, but using a very simple regular expression that matches the character supplied by the user with the character in the `answers` array. If they match, you know the answer is correct.

Now you'll add the first new text-based question and answer directly underneath the last multiple-choice question in the `GlobalFunctions.htm` file.

```
// define question 4
questions[3] = "In the Simpsons, Bleeding Gums Murphy played which instrument?";
// assign answer for question 4
answers[3] = "\\bsax(ophone)?\\b";
```

The question definition is much simpler for text-based questions than for the multiple-choice questions: it's just the question text itself.

The answer definition is a regular expression. Note that you use `\\b` rather than `\b`, since you'll be creating your regular expressions using `new RegExp()` rather than using the `/` and `/` method. The valid answers to this question are `sax` and `saxophone`, so you need to define your regular expression to match either of those. You'll see later that the case flag will be set so that even `SaxoPhone` is valid, though dubious, English! Let's break it down stage by stage as shown in the following table.

Chapter 8: String Manipulation

Expression	Description
<code>\\b</code>	The <code>\\b</code> indicates that the answer must start with a word boundary; in other words, it must be a whole word and not contained inside another word. You do this just in case the user for some reason puts characters before his answer, such as <code>My answer is saxophone</code> .
<code>Sax</code>	The user's answer must start with the characters <code>sax</code> .
<code>(ophone)?</code>	You've grouped the pattern <code>ophone</code> by putting it in parentheses. By putting the <code>?</code> just after it, you are saying that that pattern can appear zero or one time. If the user types <code>sax</code> , it appears zero times, and if the user types <code>saxophone</code> , it appears once — either way you make a match.
<code>\\b</code>	Finally you want the word to end at a word boundary.

The second question you'll create is

```
"Which American president was involved in the Watergate scandal?"
```

The possible correct answers for this are quite numerous and include the following:

```
Richard Milhous Nixon
Richard Nixon
Richard M. Nixon
Richard M Nixon
R Milhous Nixon
R. Milhous Nixon
R. M. Nixon
R M Nixon
R.M. Nixon
RM Nixon
R Nixon
R. Nixon
Nixon
```

This is a fairly exhaustive list of possible correct answers. You could perhaps accept only `Nixon` and `Richard Nixon`, but the longer list makes for a more challenging regular expression.

```
// define question 5
questions[4] = "Which American president was involved in the Watergate scandal?";
// assign answer for question 5
answers[4] = "\\b((Richard |R\\.? ?)(Milhous |M\\.? ?)?Nixon\\b";
```

Add the question-and-answer code under the other questions and answers in the `GlobalFunctions.htm` file.

Let's analyze this regular expression now.

Expression	Description
<code>\\b</code>	This indicates that the answer must start with a word boundary, so the answer must be a whole word and not contained inside another word. You do this just in case the user for some reason puts characters before his answer, such as <i>My answer is President Nixon</i> .
<code>((Richard R\\.?.?)?)</code>	This part of the expression is grouped together with the next part, <code>(Milhous M\\.?.?)?</code> . The first parenthesis creates the outer group. Inside this is an inner group, which can be one of two patterns. Before the <code> </code> is the pattern <i>Richard</i> , and after it is the pattern <i>R</i> followed by an optional dot (<code>.</code>) followed by an optional space. So either <i>Richard</i> or <i>R</i> will match. Since the <code>.</code> is a special character in regular expressions, you have to tell JavaScript you mean a literal dot and not a special-character dot. You do this by placing the <code>\</code> in front. However, because you are defining this regular expression using the <code>RegExp()</code> constructor, you need to place an additional <code>\</code> in front.
<code>(Milhous M\\.?.?)?)?</code>	This is the second subgroup within the outer group. It works in a similar way to the first subgroup, but it's <i>Milhous</i> rather than <i>Richard</i> and <i>M</i> rather than <i>R</i> that you are matching. Also, the space after the initial is not optional, since you don't want <i>RMNixon</i> . The second <code>?</code> outside this inner group indicates that the middle name/initial is optional. The final parenthesis indicates the end of the outer group. The final <code>?</code> indicates that the outer group pattern is optional — this is to allow the answer <i>Nixon</i> alone to be valid.
<code>Nixon\\b</code>	Finally the pattern <i>Nixon</i> must be matched, and followed by a word boundary.

That completes the two additional text-based questions. Now you need to alter the question creation function, `getQuestion()`, again inside the file `GlobalFunctions.htm`, as follows:

```
function getQuestion()
{
    if (questions.length != numberOfQuestionsAsked)
    {
        var questionNumber = Math.floor(Math.random() * questions.length);
        while (questionsAsked[questionNumber] == true)
        {
            questionNumber = Math.floor(Math.random() * questions.length);
        }
        var questionLength = questions[questionNumber].length;
        var questionChoice;
        numberOfQuestionsAsked++;
        var questionHTML = "<h4>Question " + numberOfQuestionsAsked + "</h4>";
        // Check if array or string
        if (typeof questions[questionNumber] == "string")
        {
            questionHTML = questionHTML + "<p>" + questions[questionNumber] + "</p>";
        }
    }
}
```

Chapter 8: String Manipulation

```
        questionHTML = questionHTML + "<p><input type=text name=txtAnswer ";
        questionHTML = questionHTML + " maxlength=100 size=35></p>";
        questionHTML = questionHTML + '<script type="text/javascript">';
            + 'document.QuestionForm.txtAnswer.value = "";</script>';
    }
    else
    {
        questionHTML = questionHTML + "<p>" + questions[questionNumber][0];
        questionHTML = questionHTML + "</p>";
        for (questionChoice = 1;questionChoice < questionLength;questionChoice++)
        {
            questionHTML = questionHTML + "<input type=radio ";
            questionHTML = questionHTML + "name=radQuestionChoice";
            if (questionChoice == 1)
            {
                questionHTML = questionHTML + " checked";
            }
            questionHTML = questionHTML + ">" +
                questions[questionNumber][questionChoice];
            questionHTML = questionHTML + "<br>"
        }
    }
    questionHTML = questionHTML + "<br><input type='button' "
    questionHTML = questionHTML + "value='Answer Question'";
    questionHTML = questionHTML + "name=buttonNextQ ";
    questionHTML = questionHTML + "onclick='return buttonCheckQ_onclick()'>";
    currentQNumber = questionNumber;
    questionsAsked[questionNumber] = true;
}
else
{
    questionHTML = "<h3>Quiz Complete</h3>";
    questionHTML = questionHTML + "You got " + numberOfQuestionsCorrect;
    questionHTML = questionHTML + " questions correct out of ";
    questionHTML = questionHTML + numberOfQuestionsAsked;
    questionHTML = questionHTML + "<br><br>Your trivia rating is ";
    switch(Math.round(((numberOfQuestionsCorrect / numberOfQuestionsAsked) * 10)))
    {
        case 0:
        case 1:
        case 2:
        case 3:
            questionHTML = questionHTML + "Beyond embarrassing";
            break;
        case 4:
        case 5:
        case 6:
        case 7:
            questionHTML = questionHTML + "Average";
            break;
        default:
            questionHTML = questionHTML + "Excellent"
    }
    questionHTML = questionHTML + "<br><br><A href='quizpage.htm'><strong>";
```

```

        questionHTML = questionHTML + "Start again</strong></A>";
    }
    return questionHTML;
}

```

You can see that the `getQuestion()` function is mostly unchanged by your need to ask text-based questions. The only code lines that have changed are the following:

```

    if (typeof questions[questionNumber] == "string")
    {
        questionHTML = questionHTML + "<p>" + questions[questionNumber] + "</p>";
        questionHTML = questionHTML + "<p><input type=text name=txtAnswer ";
        questionHTML = questionHTML + " maxlength=100 size=35></P>";
        // Next line necessary due to bugs in Netscape 7.x
        questionHTML = questionHTML + '<script type="text/javascript">'
            + 'document.QuestionForm.txtAnswer.value = "";</script>';
    }
    else
    {

```

The reason for this change is that the questions for multiple-choice and text-based questions are displayed differently. Having obtained your question number, you then need to check to see if this is a text question or a multiple-choice question. For text-based questions, you store the string containing the text inside the `questions[]` array; for multiple-choice questions, you store an array inside the `questions[]` array, which contains the question and options. You can check to see whether the type of data stored in the `questions[]` array at the index for that particular question is a string type. If it's a string type, you know you have a text-based question; otherwise you can assume it's a multiple-choice question. Note that Netscape 7.x has a habit of keeping previously entered data in text fields. This means that when the second text-based question is asked, the answer given for the previous text question is automatically pre-entered.

You use the `typeof` operator as part of the condition in your `if` statement in the following line:

```

    if (typeof questions[questionNumber] == "string")

```

If the condition is true, you then create the HTML for the text-based question; otherwise the HTML for a multiple-choice question is created.

The second function inside `GlobalFunctions.htm` that needs to be changed is the `answerCorrect()` function, which actually checks the answer given by the user.

```

function answerCorrect(questionNumber, answer)
{
    // declare a variable to hold return value
    var correct = false;
    // if answer provided is same as answer then correct answer is true
    var answerRegExp = new RegExp(answers[questionNumber], "i");
    if (answer.search(answerRegExp) != -1)
    {
        numberOfQuestionsCorrect++;
        correct = true;
    }
}

```

Chapter 8: String Manipulation

```
        // return whether the answer was correct (true or false)
        return correct;
    }
```

Instead of doing a simple comparison of the user's answer to the value in the `answers[]` array, you're now using regular expressions.

First you create a new `RegExp` object called `answerRegExp` and initialize it to the regular expression stored as a string inside your `answers[]` array. You want to do a case-insensitive match, so you pass the string `i` as the second parameter.

In your `if` statement, you search for the regular-expression answer pattern in the answer given by the user. This answer will be a string for a text-based question or a single character for a multiple-choice question. If a match is found, you'll get the character match position. If no match is found, `-1` is returned. Therefore, if the match value is not `-1`, you know that the user's answer is correct, and the `if` statement's code executes. This increments the value of the variable `numberOfQuestionsCorrect`, and sets the `correct` variable to the value `true`.

That completes the changes to `GlobalFunctions.htm`. Remember to save the file before you close it.

Finally, you have just one more function you need to alter before your changes are complete. This time the function is in the file `AskQuestion.htm`. The function is `getAnswer()`, which is used to retrieve the user's answer from the form on the page. The changes are shown in the following code:

```
function getAnswer()
{
    var answer = 0;
    if (document.QuestionForm.elements[0].type == "radio")
    {
        while (document.QuestionForm.radQuestionChoice[answer].checked != true)
            answer++;
        answer = String.fromCharCode(65 + answer);
    }
    else
    {
        answer = document.QuestionForm.txtAnswer.value;
    }
    return answer;
}
```

The user's answer can now be given via one of two means: an option being chosen in an option group, or text being entered in a text box. You determine which way was used for this question by using the `type` property of the first control in the form. If the first control is a radio button, you know this is a multiple-choice question; otherwise you assume it's a text-based question.

If it is a multiple-choice question, you obtain the answer, a character, as you did before you added text questions. If it's a text-based question, it's simply a matter of getting the text value from the text control written into the form dynamically by the `getQuestion()` function in the `GlobalFunctions.htm` page.

Save the changes to the page. You're now ready to give your updated trivia quiz a test run. Load `TriviaQuiz.htm` to start the quiz. You should now see the text questions you've created (see Figure 8-13).

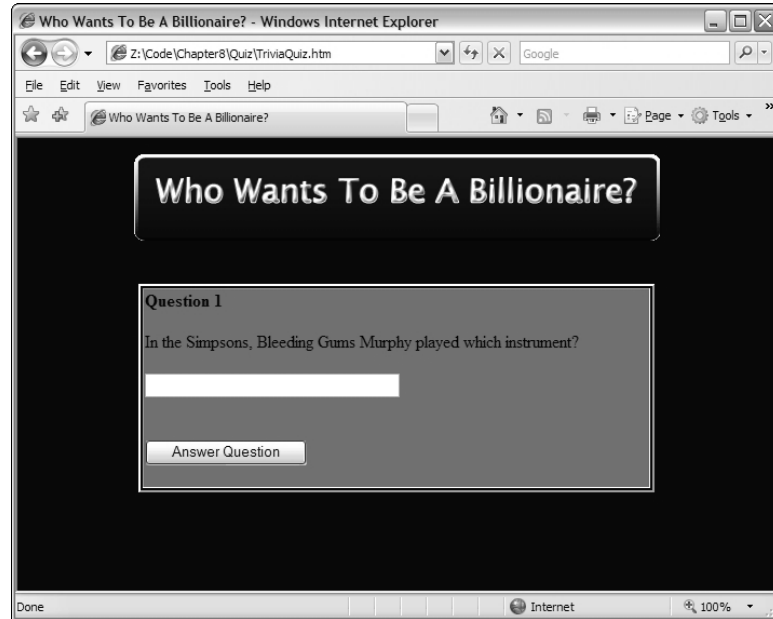


Figure 8-13

Although you've learned a bit more about regular expressions while altering the trivia quiz, perhaps the most important lesson has been that using general functions, and where possible placing them inside common code modules, makes later changes quite simple. In less than 20 lines, mostly in one file, you have made a significant addition to the quiz.

Summary

In this chapter you've looked at some more advanced methods of the `String` object and how you can optimize their use with regular expressions.

To recap, the chapter covered the following points:

- ❑ The `split()` method splits a single string into an array of strings. You pass a string or a regular expression to the method that determines where the split occurs.
- ❑ The `replace()` method enables you to replace a pattern of characters with another pattern that you specify as a second parameter.
- ❑ The `search()` method returns the character position of the first pattern matching the one given as a parameter.
- ❑ The `match()` method matches patterns, returning the text of the matches in an array.
- ❑ Regular expressions enable you to define a pattern of characters that you want to match. Using this pattern, you can perform splits, searches, text replacement, and matches on strings.

Chapter 8: String Manipulation

- ❑ In JavaScript the regular expressions are in the form of a `RegExp` object. You can create a `RegExp` object using either `myRegExp = /myRegularExpression/` or `myRegExp = new RegExp("myRegularExpression")`. The second form requires that certain special characters that normally have a single `\` in front now have two.
- ❑ The `g` and `i` characters at the end of a regular expression (as in, for example, `myRegExp = /Pattern/gi;`) ensure that a global and case-insensitive match is made.
- ❑ As well as specifying actual characters, regular expressions have certain groups of special characters, which allow any of certain groups of characters, such as digits, words, or non-word characters, to be matched.
- ❑ Special characters can also be used to specify pattern or character repetition. Additionally, you can specify what the pattern boundaries must be, for example at the beginning or end of the string, or next to a word or non-word boundary.
- ❑ Finally, you can define groups of characters that can be used later in the regular expression or in the results of using the expression with the `replace()` method.
- ❑ You also updated the trivia quiz in this chapter to allow questions to be set that require a text-based response from the user, in addition to the multiple-choice questions you have already seen.

In the next chapter you'll take a look at using and manipulating dates and times using JavaScript, and time conversion between different world time zones. Also covered is how to create a timer that executes code at regular intervals after the page is loaded. You'll be adapting the trivia quiz so that the user can select a time within which it must be completed — enabling him to specify, for example, that five questions must be answered in one minute.

Exercise Questions

Suggested solutions to these questions can be found in Appendix A.

Question 1

What problem does the code below solve?

```
var myString = "This sentence has has a fault and and we need to fix it."
var myRegExp = /(\\b\\w+\\b) \\1/g;
myString = myString.replace(myRegExp, "$1");
```

Now imagine that you change that code, so that you create the `RegExp` object like this:

```
var myRegExp = new RegExp("(\\b\\w+\\b) \\1");
```

Why would this not work, and how could you rectify the problem?

Question 2

Write a regular expression that finds all of the occurrences of the word “a” in the following sentence and replaces them with “the”:

“a dog walked in off a street and ordered a finest beer”

The sentence should become:

“the dog walked in off the street and ordered the finest beer”

Question 3

Imagine you have a web site with a message board. Write a regular expression that would remove barred words. (I’ll let you make up your own words!)

9

Date, Time, and Timers

In Chapter 4 we discuss that the concepts of date and time are embodied in JavaScript through the `Date` object. You looked at some of the properties and methods of the `Date` object, including the following:

- ❑ The methods `getDate()`, `getDay()`, `getMonth()`, and `getFullYear()` enable you to retrieve date values from inside a `Date` object.
- ❑ The `setDate()`, `setMonth()`, and `setFullYear()` methods enable you to set the date values of an existing `Date` object.
- ❑ The `getHours()`, `getMinutes()`, `getSeconds()`, and `getMilliseconds()` methods retrieve the time values in a `Date` object.
- ❑ The `setHours()`, `setMinutes()`, `setSeconds()`, and `setMilliseconds()` methods enable you to set the time values of an existing `Date` object.

One thing not covered in that chapter was the idea that the time depends on your location around the world. In this chapter you'll be correcting that omission by looking at date and time in relation to *world time*.

For example, imagine you have a chat room on your web site and want to organize a chat for a certain date and time. Simply stating 15:30 is not good enough if your web site attracts international visitors. The time 15:30 could be Eastern Standard Time, Pacific Standard Time, the time in the United Kingdom, or even the time in Kuala Lumpur. You could of course say 15:30 EST and let your visitors work out what that means, but even that isn't foolproof. There is an EST in Australia as well as in the United States. Wouldn't it be great if you could automatically convert the time to the user's time zone? In this chapter you'll see how.

As well as looking at world time, you'll also be looking at how to create a *timer* in a web page. You'll see that by using the timer you can trigger code, either at regular intervals or just once (for example, five seconds after the page has loaded). You'll see how you can use timers to add a real-time clock to a web page and how to create scrolling text in the status bar. Timers can also be useful for creating animations or special effects in your web applications. Finally, you'll be using the timer to enable the users of your trivia quiz to give themselves a time limit for answering the questions.

World Time

The concept of *now* means the same point in time everywhere in the world. However, when that point in time is represented by numbers, those numbers differ depending on where you are. What is needed is a standard number to represent that moment in time. This is achieved through Coordinated Universal Time (UTC), which is an international basis of civil and scientific time and was implemented in 1964. It was previously known as GMT (Greenwich Mean Time), and, indeed, at 0:00 UTC it is midnight in Greenwich, London.

The following table shows local times around the world at 0:00 UTC time.

San Francisco	New York (EST)	Greenwich, London	Berlin, Germany	Tokyo, Japan
4:00 pm	7:00 pm	0:00 (midnight)	1:00 am	9:00 am

Note that the times given are winter times — no daylight saving hours are taken into account.

The support for UTC in JavaScript comes from a number of methods of the `Date` object that are similar to those you have already seen. For each of the set-date- and get-date-type methods you've seen so far, there is a UTC equivalent. For example, whereas `setHours()` sets the local hour in a `Date` object, `setUTCHours()` does the same thing for UTC time. You'll be looking at these methods in more detail in the next section.

In addition, three more methods of the `Date` object involve world time.

You have the methods `toUTCString()` and `toLocaleString()`, which return the date and time stored in the `Date` object as a string based on either UTC or local time. Most modern browsers also have these additional methods: `toLocaleTimeString()`, `toTimeString()`, `toLocaleDateString()`, and `toDateString()`.

If you simply want to find out the difference in minutes between the current locale's time and UTC, you can use the `getTimezoneOffset()` method. If the time zone is behind UTC, such as in the United States, it will return a positive number. If the time zone is ahead, such as in Australia or Japan, it will return a negative number.

Try It Out The World Time Method of the Date Object

In the following code you use the `toLocaleString()`, `toUTCString()`, `getTimezoneOffset()`, `toLocaleTimeString()`, `toTimeString()`, `toLocaleDateString()`, and `toDateString()` methods and write their values out to the page.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>example</title>
<script language="JavaScript" type="text/javascript">
    var localTime = new Date();
</script>
</head>
```

```

<body>
<h4>
    UTC Time is
    <script language="JavaScript" type="text/javascript">

        document.write(localTime.toUTCString());
    </script>
</h4>
<h4>
    Local Time is
    <script language="JavaScript" type="text/javascript">
        document.write(localTime.toLocaleString());
    </script>
</h4>
<h4>
    Time Zone Offset is
    <script language="JavaScript" type="text/javascript">
        document.write(localTime.getTimezoneOffset());
    </script>
</h4>

<h4>
    Using toLocalTimeString() gives:
    <script language="JavaScript" type="text/javascript">
    if (localTime.toLocaleTimeString)
    {
        document.write(localTime.toLocaleTimeString())
    }
    </script>
</h4>

<h4>
    Using toTimeString() gives:
    <script language="JavaScript" type="text/javascript">
    if (localTime.toTimeString)
    {
        document.write(localTime.toTimeString() )
    }
    </script>
</h4>

<h4>
    Using toLocaleDateString() gives:
    <script language="JavaScript" type="text/javascript">
    if (localTime.toLocaleDateString)
    {
        document.write(localTime.toLocaleDateString())
    }
    </script>
</h4>

<h4>
    Using toDateString() gives:

```

Chapter 9: Date, Time, and Timers

```
<script language="JavaScript" type="text/javascript">
if (localTime.toString())
{
    document.write(localTime.toString())
}
</script>
</h4>

</body>
</html>
```

Save this as `timetest.htm` and load it into your browser. What you see, of course, depends on which time zone your computer is set to, but your browser should show something similar to Figure 9-1.

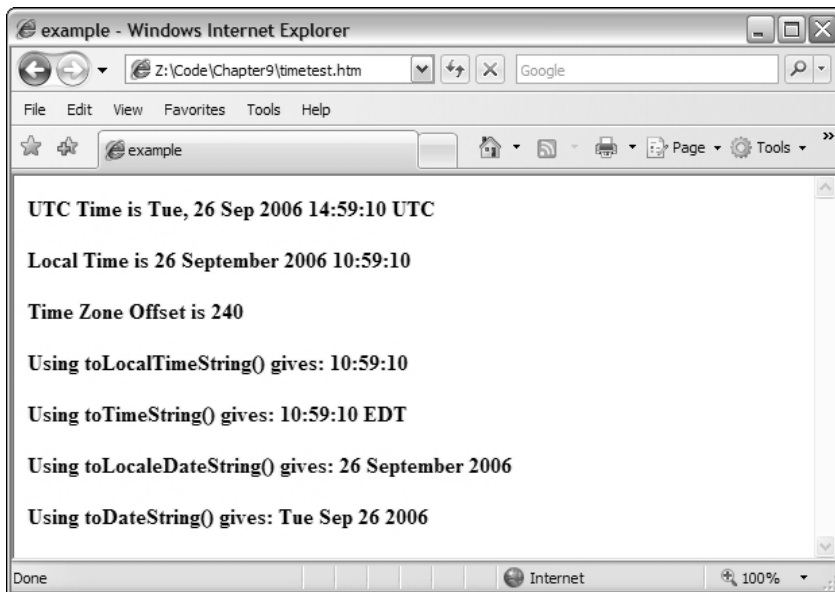


Figure 9-1

Here the computer's time is set to 10:59:10 a.m. on September 26, 2006, in America's Eastern Standard Time (for example, New York).

How It Works

So how does this work? In the script block at the top of the page, you have just this one line:

```
var localTime = new Date();
```

This creates a new `Date` object and initializes it to the current date and time based on the client computer's clock. (Note that in fact the `Date` object simply stores the number of milliseconds between the date and time on your computer's clock and midnight UTC on January 1, 1970.)

Within the body of the page you have seven more script blocks that use the three world time methods you looked at earlier. Note that some of them are enclosed in an `if` statement, for example this one:

```
if (localTime.toLocaleTimeString)
{
    document.write(localTime.toLocaleTimeString())
}
```

The `if` statement checks to see if the browser supports the method and only makes use of it if so. Older browsers don't support all of the date/time methods so doing this prevents ugly errors.

In the following line you write the string returned by the `toUTCString()` method to the page:

```
document.write(localTime.toUTCString());
```

This converts the date and time stored inside the `localTime` `Date` object to the equivalent UTC date and time.

Then the following line returns a string with the local date and time value:

```
document.write(localTime.toLocaleString());
```

Since this time is just based on the user's computer's clock, the string returned by this method also adjusts for Daylight Saving Time (as long as the clock adjusts for it).

Next, this code writes out the difference, in minutes, between the local time zone's time and that of UTC.

```
document.write(localTime.getTimezoneOffset());
```

You may notice in Figure 9-1 that the difference between New York time and UTC time is written to be 240 minutes, or 4 hours. Yet in the previous table, you saw that New York time is 5 hours behind UTC. So what is happening?

Well, in New York on September 26, daylight saving hours are in use. While in the summer it's 8:00 p.m. in New York when it's 0:00 UTC, in the winter it's 7:00 p.m. in New York when it's 0:00 UTC. Therefore, in the summer the `getTimezoneOffset()` method returns 240, whereas in the winter the `getTimezoneOffset()` method returns 300.

To illustrate this, compare Figure 9-1 to Figure 9-2, where the date on the computer's clock has been advanced by two months.

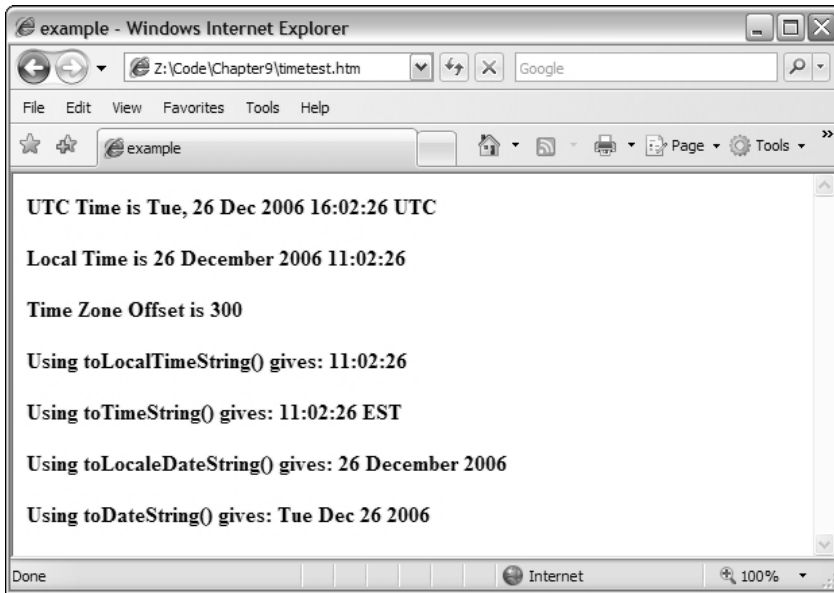


Figure 9-2

The next two methods are `toLocaleTimeString()` and `toTimeString()`, as follows:

```
<h4>
Using toLocalTimeString() gives:
<script language="JavaScript" type="text/javascript">
document.write(localTime.toLocaleTimeString())
</script>
</h4>
```

```
<h4>
Using toTimeString() gives:
<script language="JavaScript" type="text/javascript">
document.write(localTime.toTimeString() )
</script>
</h4>
```

These methods display just the time part of the date and time held in the `Date` object. The `toLocaleTimeString()` method displays the time as specified by the user on his computer. The second method displays the time but also gives an indication of the time zone (in the example, EST for Eastern Standard Time in America).

The final two methods display the date part of the date and time. The `toLocaleDateString()` displays the date in the format the user has specified on his computer. On Windows operating systems, this is set in the regional settings of the PC's Control Panel. However, because it relies on the user's PC setup, the look of the date varies from computer to computer. The `toDateString()` method displays the current date contained in the `Date` object in a standard format.

Of course, this example relies on the fact that the user's computer's clock is set correctly, not something you can be 100 percent sure of — it's amazing how many users have their local time zone settings set completely wrong.

Setting and Getting a Date Object's UTC Date and Time

When you create a new `Date` object, you can either initialize it with a value or let JavaScript set it to the current date and time. Either way, JavaScript assumes you are setting the *local* time values. If you want to specify UTC time, you need to use the `setUTC` type methods, such as `setUTCHours()`.

The following are the seven methods for setting UTC date and time:

- ☐ `setUTCDate()`
- ☐ `setUTCFullYear()`
- ☐ `setUTCHours()`
- ☐ `setUTCMilliseconds()`
- ☐ `setUTCMinutes()`
- ☐ `setUTCMonth()`
- ☐ `setUTCSeconds()`

The names pretty much give away exactly what each of the methods does, so let's launch straight into a simple example, which sets the UTC time.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<body>
<script language="JavaScript" type="text/javascript">
var myDate = new Date();
myDate.setUTCHours(12);
myDate.setUTCMinutes(0);
myDate.setUTCSeconds(0);
document.write("<h3>" + myDate.toUTCString() + "</h3>")
document.write("<h3>" + myDate.toLocaleString() + "</h3>")
</script>
</body>
</html>
```

Save this as `settimetest.htm`. When you load it in your browser, you should see something like that shown in Figure 9-3 in your web page, although the actual date will depend on the current date and where you are in the world.

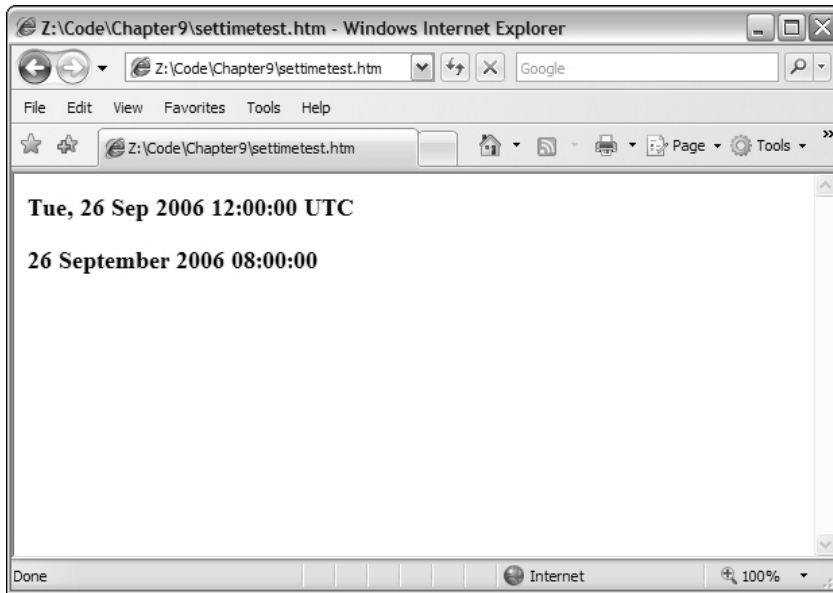


Figure 9-3

You might want to change your computer's time zone and time of year to see how it varies in different regions and with daylight saving changes. For example, although I'm in the United Kingdom, I have changed the settings on my computer for this example to Eastern Standard Time in the U.S. In Windows you can make the changes by opening the Control Panel and then double-clicking the Date/Time icon.

So how does this example work? You declare a variable, `myDate`, and set it to a new `Date` object. Because you haven't initialized the `Date` object to any value, it contains the local current date and time.

Then, using the `setUTC` methods, you set the hours, minutes, and seconds so that the time is 12:00:00 UTC (midday, not midnight).

Now, when you write out the value of `myDate` as a UTC string, you get 12:00:00 and today's date. When you write out the value of the `Date` object as a local string, you get today's date and a time that is the UTC time 12:00:00 converted to the equivalent local time. The local values you'll see, of course, depend on your time zone. For example, New Yorkers will see 08:00:00 during the summer and 07:00:00 during the winter because of daylight savings. In the United Kingdom, in the winter you'll see 12:00:00, but in the summer you'll see 13:00:00.

For getting UTC dates and times, you have the same functions you would use for setting UTC dates and times, except that this time, for example, it's `getUTCHours()` and not `setUTCHours()`.

- ☐ `getUTCDate()`
- ☐ `getUTCDay()`
- ☐ `getUTCFullYear()`

- ☐ `getUTCHours()`
- ☐ `getUTCMilliseconds()`
- ☐ `getUTCMinutes()`
- ☐ `getUTCMonth()`
- ☐ `getUTCSeconds()`

Notice that this time there is an additional method, `getUTCDay()`. This works in the same way as the `getDay()` method and returns the day of the week as a number, from 0 for Sunday to 6 for Saturday. Because the day of the week is decided by the day of the month, the month, and the year, there is no `setUTCDay()` method.

Before moving on to look at timers, let's use your newly gained knowledge of the `Date` object and world time to create a world time converter. Later in this chapter, when you've learned how to use timers, you'll update the example to produce a world time clock.

Try It Out World Time Converter (Part I)

The world time converter consists of two pages. The first is a frameset page, and the second is the page where the time conversion form exists. Let's start by creating the frameset page.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
"http://www.w3.org/TR/html4/frameset.dtd">
<html>
<head>
</head>
<frameset cols="250,*" frameborder="0">
  <frame src="worldtimeconverter.htm" name=formFrame>
  <frame src="about:blank" name=resultsFrame>
</frameset>
</html>
```

This simply divides the page into two frames. However, by setting the border between the frames to 0, you can make it look like just a single page. Save this frameset page as `WorldTimeConverterFrameset.htm`.

The left frame will contain the form in which the user can select the city whose time she wants, and the right frame will show the results of the conversion. The right frame will be written using code, so there is no page to create for that.

Next, create the left frame's page.

```
<html>
<head>
<script language="javascript" type="text/javascript">
var timeDiff;
var selectedCity;
var daylightSavingAdjust = 0;
function updateTimeZone()
{
```

```
var lstCity = document.form1.lstCity;
timeDiff = lstCity.options[lstCity.selectedIndex].value;
selectedCity = lstCity.options[lstCity.selectedIndex].text;
updateTime();
}
function getTimeString(dateObject)
{
    var timeString;
    var hours = dateObject.getHours();
    if (hours < 10)
        hours = "0" + hours;
    var minutes = dateObject.getMinutes();
    if (minutes < 10)
        minutes = "0" + minutes;
    var seconds = dateObject.getSeconds();
    if (seconds < 10)
        seconds = "0" + seconds;
    timeString = hours + ":" + minutes + ":" + seconds;
    return timeString;
}
function updateTime()
{
    var nowTime = new Date();
    var resultsFrame = window.top.resultsFrame.document;
    resultsFrame.open()
    resultsFrame.write("Local Time is " + getTimeString(nowTime) + "<br>");
    nowTime.setMinutes(nowTime.getMinutes() + nowTime.getTimezoneOffset() +
        parseInt(timeDiff) + daylightSavingAdjust);
    resultsFrame.write(selectedCity + " time is " + getTimeString(nowTime));
    resultsFrame.close();
}
function chkDaylightSaving_onclick()
{
    if (document.form1.chkDaylightSaving.checked)
    {
        daylightSavingAdjust = 60;
    }
    else
    {
        daylightSavingAdjust = 0;
    }
    updateTime();
}
</script>
</head>
<body onload="updateTimeZone()">
<form name=form1>
<select size=5 name=lstCity language=JavaScript onchange="updateTimeZone();">
<option value=60 selected>Berlin
<option value=330>Bombay
<option value=0>London
<option value=180>Moscow
<option value=-300>New York (EST)
<option value=60>Paris
<option value=-480>San Francisco (PST)
```

```

<option value=600>Sydney
</select>
<p>
It's summertime in the selected city
and its country adjusts for summertime daylight saving
<input type="checkbox" name=chkDaylightSaving language=JavaScript
    onclick="return chkDaylightSaving_onclick()">
</p>
</form>
</body>
</html>

```

Save this page as `WorldTimeConverter.htm`. Then load the `WorldTimeConverterFrameset.htm` page into your browser.

The form layout looks something like that shown in Figure 9-4. Whenever the user clicks a city in the list, her local time and the equivalent time in the selected city are shown. In the example shown in Figure 9-4, the local region is set to Eastern Standard Time in the U.S. (for a city such as New York), and the selected city is Berlin, with the daylight saving box checked.

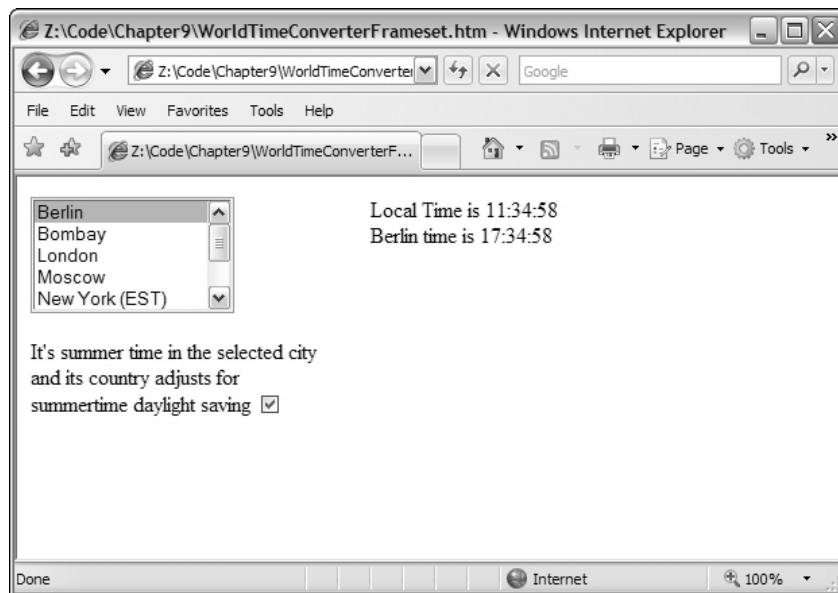


Figure 9-4

It's worth pointing out that this is an example and not a totally foolproof one, because of the problems presented by daylight saving. Some countries don't have it, others do at fixed times of year, and yet others do but at varying times of the year. This makes it difficult to predict accurately when a country will have its daylight saving period. You have tried to solve this problem by adding a check box for the user to click if the city she chooses from the list is using daylight saving hours (which you assume will put the time in the city forward by one hour).

Chapter 9: Date, Time, and Timers

In addition, don't forget that some users may not even have their regional settings set correctly — there's no easy way around this problem.

How It Works

Before you look at `WorldTimeConverter.htm`, let's just pick up on one point in the frameset-defining page `WorldTimeConverterFrameset.htm`.

```
<frame src="about:blank" name=resultsFrame>
```

Notice how you have set the `src` attribute of the right-hand frame to `"about:blank"`. This is necessary for some browsers, mainly Netscape — without it, `document.write()` will sometimes fail because there is no page loaded to write to. Note that as it stands, this example doesn't work with Opera 7. To make it work you simply need to create a blank HTML page and load that into the results frame.

Now you'll turn to the page `WorldTimeConverter.htm`, where most of the action is. In the body of the page is a form in which you've defined a list box using a `<select>` element.

```
<select size=5 name=lstCity language=JavaScript onchange="updateTimeZone();">
<option value=60 selected>Berlin
<option value=330>Bombay
<option value=0>London
<option value=180>Moscow
<option value=-300>New York (EST)
<option value=60>Paris
<option value=-480>San Francisco (PST)
<option value=600>Sydney
</select>
```

Each of the options displays the city's name in the list box and has its value set to the difference in minutes between that city's time zone (in winter) and UTC. So London, which uses UTC, has a value of 0. Paris, which is an hour ahead of UTC, has a value of 60 (that is, 60 minutes). New York, which is five hours behind UTC, has a value of -300.

You'll see that you have captured the `change` event of the `<select>` element and connected it to the function `updateTimeZone()` defined in a script block in the head of the page. This function involves three global variables defined at the top of the script block.

```
var timeDiff;
var selectedCity;
var daylightSavingAdjust = 0;
```

The function `updateTimeZone()` updates two of these, setting the variable `timeDiff` to the value of the list's selected option (that is, the time difference between the selected city and UTC time) and the variable `selectedCity` to the text shown for the selected option (that is, the selected city).

```
function updateTimeZone()
{
    var lstCity = document.form1.lstCity;
    timeDiff = lstCity.options[lstCity.selectedIndex].value;
    selectedCity = lstCity.options[lstCity.selectedIndex].text;
```

In the final part of the function `updateTimeZone()`, the function `updateTime()` is called, as shown in the following:

```
    updateTime();  
}
```

Before you go on to look at this function, you return to the final part of the form on the page. This is a check box, which the user clicks if the city she has chosen from the select list is in the summertime of a country that uses daylight saving hours.

```
<input type="checkbox" name=chkDaylightSaving language=JavaScript  
      onclick="return chkDaylightSaving_onclick()">
```

As you can see, this check box's click event is connected to another function, `chkDaylightSaving_onclick()`.

```
function chkDaylightSaving_onclick()  
{  
    if (document.form1.chkDaylightSaving.checked)  
    {  
        daylightSavingAdjust = 60;  
    }  
    else  
    {  
        daylightSavingAdjust = 0;  
    }  
}
```

Inside the `if` statement, the code accesses the check box's `checked` property, which returns `true` if it is checked and `false` otherwise. If it has been checked, you set the global variable `daylightSavingAdjust` to 60 for summertime daylight saving; otherwise it's set to 0.

```
    updateTime();  
}
```

At the end of this function (as at the end of the function `updateTimeZone()` you saw earlier), the `updateTime()` function is called. You'll look at that next.

In the function `updateTime()`, you write the current local time and the equivalent time in the selected city to the right-hand frame, named `resultsFrame`, which you defined in the `frameset` page.

You start at the top of the function by creating a new `Date` object, which is stored in the variable `nowTime`. The `Date` object will be initialized to the current local time.

```
function updateTime()  
{  
    var nowTime = new Date();
```

Next, to make your code more compact and easier to understand, you define a variable, `resultsFrame`, which will reference the `document` object contained in the `resultsFrame` window.

```
    var resultsFrame = window.top.resultsFrame.document;
```

Chapter 9: Date, Time, and Timers

With your reference to the `resultsFrame` document, you then open the document to write to it. Doing this will clear anything currently in the document and provide a nice blank document to write your HTML into. The first thing you write is the local time based on the new `Date` object you just created. However, you want the time to be nicely formatted as *hours:minutes:seconds*, so you've written another function, `getTimeString()`, which does this for you. You'll look at that shortly.

```
resultsFrame.open()
resultsFrame.write("Local Time is " + getTimeString(nowTime) + "<br>");
```

Having written the current time to your `resultsFrame`, you now need to calculate what the time would be in the selected city before also writing that to the `resultsFrame`.

You saw in Chapter 4 that if you set the value of a `Date` object's individual parts (such as hours, minutes, and seconds) to a value beyond their normal range, JavaScript assumes you want to adjust the date, hours, or minutes to take this into account. For example, if you set the hours to 36, JavaScript simply changes the hours to 12 and adds one day to the date stored inside the `Date` object. You use this to your benefit in the following line:

```
nowTime.setMinutes(nowTime.getMinutes() + nowTime.getTimezoneOffset() +
    parseInt(timeDiff) + daylightSavingsAdjust);
```

Let's break this line down to see how it works. Suppose that you're in New York, with the local summer time of 5:11, and you want to know what time it is in Berlin. How does your line of code calculate this?

First you get the minutes of the current local time; it's 5:11, so `nowTime.getMinutes()` returns 11.

Then you get the difference, in minutes, between the user's local time and UTC using `nowTime.getTimezoneOffset()`. If you are in New York, which is different from UTC by 4 hours during the summer, this is 240 minutes.

Then you get the integer value of the time difference between the standard winter time in the selected city and UTC time, which is stored in the variable `timeDiff`. You've used `parseInt()` here because it's one of the few situations where JavaScript gets confused and assumes you want to join two strings together rather than treat the values as numbers and add them together. Remember that you got `timeDiff` from an HTML element's value, and that an HTML element's values are strings, even when they hold characters that are digits. Since you want the time in Berlin, which is 60 minutes different from UTC time, this value will be 60.

Finally, you add the value of `daylightSavingsAdjust`. This variable is set in the function `chkdaylightsaving_onclick()`, which we discussed earlier. Since it's summer where you are and Berlin uses daylight saving hours, this value is 60.

So you have the following:

$$11 + 240 + 60 + 60 = 371$$

Therefore `nowTime.setMinutes()` is setting the minutes to 371. Clearly, there's no such thing as 371 minutes past the hour, so instead JavaScript assumes you mean 6 hours and 11 minutes after 5:00, that being 11:11 — the time in Berlin that you wanted.

Finally, the `updateTime()` function writes the results to the `resultsFrame`, and then closes off the document writing.

```
resultsFrame.write(selectedCity + " time is " + getTimeString(nowTime));
resultsFrame.close();
}
```

In the `updateTime()` function, you saw that it uses the function `getTimeString()` to format the time string. Let's look at that function now. This function is passed a `Date` object as a parameter and uses it to create a string with the format *hours:minutes:seconds*.

```
function getTimeString(dateObject)
{
    var timeString;
    var hours = dateObject.getHours();
    if (hours < 10)
        hours = "0" + hours;
    var minutes = dateObject.getMinutes();
    if (minutes < 10)
        minutes = "0" + minutes;
    var seconds = dateObject.getSeconds();
    if (seconds < 10)
        seconds = "0" + seconds;
    timeString = hours + ":" + minutes + ":" + seconds;
    return timeString;
}
```

Why do you need this function? Well, you can't just use this:

```
getHours() + ":" + getMinutes() + ":" + getSeconds()
```

That won't take care of those times when any of the three results of these functions is less than 10. For example, 1 minute past noon would look like 12:1:00 rather than 12:01:00.

The function therefore gets the values for hours, minutes, and seconds and checks each to see if it is below 10. If it is, a zero is added to the front of the string. When all the values have been retrieved, they are concatenated in the variable `timeString` before being returned to the calling function.

In the next section you're going to look at how, by adding a timer, you can make the displayed time update every second like a clock.

Timers in a Web Page

You can create two types of timers, one-shot timers and continually firing timers. The *one-shot timer* triggers just once after a certain period of time, and the second type of timer continually triggers at set intervals. You will investigate each of these types of timers in the next two sections.

Within reasonable limits you can have as many timers as you want and can set them going at any point in your code, such as at the window `onload` event or at the click of a button. Common uses for timers

include advertisement banner pictures that change at regular intervals or display the changing time in a web page. Also all sorts of animations done with DHTML need `setTimeout()` or `setInterval()` — you'll be looking at DHTML later on in the book.

One-Shot Timer

Setting a one-shot timer is very easy: you just use the `window` object's `setTimeout()` method.

```
window.setTimeout("your JavaScript code", milliseconds_delay)
```

The method `setTimeout()` takes two parameters. The first is the JavaScript code you want executed, and the second is the delay, in milliseconds (thousandths of a second), until the code is executed.

The method returns a value (an integer), which is the timer's unique ID. If you decide later that you want to stop the timer firing, you use this ID to tell JavaScript which timer you are referring to.

For example, to set a timer that fires three seconds after the page has loaded, you could use the following code:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<script language="JavaScript" type="text/javascript">
var timerID;
function window_onload()
{
    timerID = setTimeout("alert('Times Up!')",3000);
    alert("Timer Set");
}
</script>
</head>
<body language = JavaScript onload="window_onload()">
</body>
</html>
```

Save this file as `timertest.htm`, and load it into your browser. In this page a message box appears 3,000 milliseconds (that is, 3 seconds) after the `onload` event of the window has fired.

Although `setTimeout()` is a method of the `window` object, you'll remember that because the `window` object is at the top of the hierarchy, you don't need to use its name when referring to its properties and methods. Hence, you can use `setTimeout()` instead of `window.setTimeout()`.

It's important to note that setting a timer does not stop the script from continuing to execute. The timer runs in the background and fires when its time is up. In the meantime the page runs as usual, and any script after you start the timer's countdown will run immediately. So, in this example, the alert box telling you that the timer has been set appears immediately after the code setting the timer has been executed.

What if you decided that you wanted to stop the timer before it fired?

To clear a timer you use the window object's `clearTimeout()` method. This takes just one parameter, the unique timer ID that the `setTimeout()` method returns.

Let's alter the preceding example and provide a button that you can click to stop the timer.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<script language=JavaScript type="text/javascript">
var timerID;
function window_onload()
{
    timerID = setTimeout("alert('Times Up!')",3000);
    alert("Timer Set");
}
function butStopTimer_onclick()
{
    clearTimeout(timerID);
    alert("Timer has been cleared");
}
</script>
</head>
<body onload="window_onload()">
<form name=form1>
<input type="button" value="Stop Timer" name=butStopTimer language=JavaScript
    onclick="return butStopTimer_onclick()">
</form>
</body>
</html>
```

Save this as `timertest2.htm` and load it into your browser. Now if you click the Stop Timer button before the three seconds are up, the timer will be cleared. This is because the button is connected to the `butStopTimer_onclick()` function, which uses the timer's ID `timerID` with the `clearTimeout()` method of the window object.

Try It Out Updating a Banner Advertisement

You'll now look at a bigger example using the `setTimeout()` method. The following example creates a web page with an image banner advertisement that changes every few seconds.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<script language=JavaScript type="text/javascript">
var currentImgNumber = 1;
var numberOfImages = 3;
function window_onload()
{
    setTimeout("switchImage()",3000);
}
function switchImage()
```

Chapter 9: Date, Time, and Timers

```
{
    currentImgNumber++;
    document.imgAdvert.src = "AdvertImage" + currentImgNumber + ".jpg";
    if (currentImgNumber < numberOfImages)
    {
        setTimeout("switchImage()", 3000);    }
    }
</script>
</head>
<body onload="window_onload()">

</body>
</html>
```

After you've typed in the code, save the page as `Adverts.htm`. You'll also need to create three images named `AdvertImage1.jpg`, `AdvertImage2.jpg`, and `AdvertImage3.jpg` (alternatively, the three images are supplied with the downloadable code for the book).

When the page is loaded, you start with a view of `AdvertImage1.jpg`, as shown in Figure 9-5.



Figure 9-5

In three seconds this changes to the second image, shown in Figure 9-6.



Figure 9-6

Finally, three seconds later, a third and final image loads, shown in Figure 9-7.



Figure 9-7

How It Works

When the page loads, the `` tag has its `src` attribute set to the first image.

```

```

Within the `<body>` tag, you connect the window object's `onload` event handler to the function `window_onload()`.

```
function window_onload()
{
    setTimeout("switchImage()", 3000)
}
```

In this function you use the `setTimeout()` method to start a timer running that will call the function `switchImage()` in three seconds. Since you don't have to clear the timer, you haven't bothered to save the timer ID returned by the `setTimeout()` method.

The `switchImage()` function changes the value of the `src` property of the `img` object corresponding to the `` tag in your page.

```
function switchImage()
{
    currentImgNumber++;
    document.imgAdvert.src = "AdvertImage" + currentImgNumber + ".jpg";
}
```

Your advertisement images are numbered from one to three: `AdvertImage1.jpg`, `AdvertImage2.jpg`, and `AdvertImage3.jpg`. You keep track of the number of the advertisement image that is currently loaded in the page in the global variable `currentImgNumber`, which you defined at the top of the script block and initialized to 1. To get the next image you simply increment that variable by one, and then update the image loaded by setting the `src` property of the `img` object, using the variable `currentImgNumber` to build up its full name.

```
if (currentImgNumber < numberOfImages)
{
    setTimeout("switchImage()", 3000);
}
```

You have three advertisement images you want to show. In the `if` statement you check to see whether `currentImgNumber`, which is the number of the current image, is less than three. If it is, it means there are more images to show, and so you set another timer going, identical to the one you set in the window object's `onload` event handler. This timer will call this function again in three seconds.

In earlier browsers, this was the only method of creating a timer that fired continually at regular intervals. However, in version 4 and later browsers, you'll see next that there's an easier way.

Setting a Timer that Fires at Regular Intervals

Modern browsers saw new methods added to the window object for setting timers, namely the `setInterval()` and `clearInterval()` methods. These work in a very similar way to `setTimeout()` and `clearTimeout()`, except that the timer fires continually at regular intervals rather than just once.

The method `setInterval()` takes the same parameters as `setTimeout()`, except that the second parameter now specifies the interval, in milliseconds, between each firing of the timer, rather than just the length of time before the timer fires.

For example, to set a timer that fires the function `myFunction()` every five seconds, the code would be as follows:

```
var myTimerID = setInterval("myFunction()",5000);
```

As with `setTimeout()`, the `setInterval()` method returns a unique timer ID that you'll need if you want to clear the timer with `clearInterval()`, which works identically to `clearTimeout()`. So to stop the timer started in the preceding code, you would use the following:

```
clearInterval(myTimerID);
```

Try It Out World Time Converter (Part 2)

Let's change the world time example that you saw earlier, so that it displays local time and selected city time as a continually updating clock.

You'll be making changes to the `WorldTimeConverter.htm` file, so open that in your text editor. Add the following function before the functions that are already defined:

```
var daylightSavingAdjust = 0;
function window_onload()
{
    updateTimeZone();
    window.setInterval("updateTime()",1000);
}
function updateTimeZone()
{

```

Next edit the `<body>` tag so it looks like this:

```
<body onload="return window_onload()">
```

Resave the file, and then load `WorldTimeConverterFrameset.htm` into your browser. The page should look the same as the previous version of the time converter, except that the time is updated every second.

How It Works

The changes were short and simple. In the function `window_onload()`, you have added a timer that will call the `updateTime()` function every 1,000 milliseconds—that is, every second. It'll keep doing this until you leave the page. Previously your `updateTime()` function was called only when the user clicked either a different city in the list box or the summertime check box.

The `window_onload()` function is connected to the `window` object's `onload` event in the `<body>` tag, so after the page has loaded your clock starts running.

That completes your look at this example and also your introduction to timers. Next you're going to use this knowledge to alter the trivia quiz so that it is a time-limit-based quiz.

The Trivia Quiz

In this chapter you'll be making two changes to the trivia quiz. You'll allow the user to select first how long she has to complete the quiz and second how many questions she wants to answer.

Converting the quiz to a timer-based one requires only that you change two pages, namely `QuizPage.htm` and `GlobalFunctions.htm`.

Your first change in `QuizPage.htm` will be to the form at the start of the quiz, because you need to allow the user to select her time limit and number of questions. Then you'll change the `cmdStartQuiz_onclick()` function so that when it calls the `resetQuiz()` function in the `GlobalFunctions.htm` page, it also passes (as parameters) the time limit and number of questions that the user has selected.

Now you go on to the `GlobalFunctions.htm` page itself, where you need to alter the `resetQuiz()` function so that, if necessary, it starts a timer based on the time limit selected. Then new functions dealing with the time limit need to be created: one puts a message in the scroll bar notifying the user how much time she has left; the other deals with the situation when the time limit is up.

You'll start by making the changes to `QuizPage.htm`, so open this in your text editor.

The first change you'll make is to the form, which currently contains just a button.

```
<form name="frmQuiz">
<p>
Number of Questions <br>
<select name="cboNoQuestions" size="1">
  <option value="3">3
  <option value="5">5
</select>
</p>
<p>
Time Limit <br>
<select name="cboTimeLimit" size="1">
  <option value="-1">No Time Limit
  <option value="60">1 Minute
  <option value="180">3 Minutes
  <option value="300">5 Minutes
</select>
</p>
<input name="cmdStartQuiz" type="button" value="Start Quiz"
  onclick="return cmdStartQuiz_onclick()" >
</form>
```

You've added two new controls; both are drop-down list boxes created using the `<select>` tag. In the first list box you enable the user to choose how many questions she wants to answer, and in the second the time limit within which she must answer the questions.

Next you need to alter the `cmdStartQuiz_onclick()` function defined at the top of the page.

```
<script language=JavaScript>
function cmdStartQuiz_onclick()
```

```

{
    var cboNoQuestions = document.frmQuiz.cboNoQuestions;
    var noQuestions = cboNoQuestions.options[cboNoQuestions.selectedIndex].value;
    var cboTimeLimit = document.frmQuiz.cboTimeLimit;
    var timeLimit = cboTimeLimit.options[cboTimeLimit.selectedIndex].value;
    window.top.fraTopFrame.fraGlobalFunctions.resetQuiz(noQuestions,timeLimit);
    window.location.href = "AskQuestion.htm";
}
</script>

```

This function is connected to the `cmdStartQuiz` button's `onclick` event handler and is how the user kicks off the quiz. Previously you just called the `resetQuiz()` function in the global module and then loaded the `AskQuestion.htm` page. Now you need to get the values the user has selected in the select elements for the number of questions and time limit. As you'll see in a minute, the `resetQuiz()` function has been changed and now takes two parameters, namely the number of questions to be answered and a time limit.

At the start of the `cmdStartQuiz_onclick()` function, you first set a variable `cboNoQuestions` to reference the `cboNoQuestions` control in the form (where the user chooses how many questions she wants to answer). You can then use that reference instead of the more long-winded full reference via the document and form. The benefit of doing this is that it keeps the lines shorter and more readable.

On the second line you get the value of the selected option in the select control for the number of questions and store this value in the `noQuestions` variable.

In the following two lines you do the same thing again, except that this time it's the time limit control and value that you need to deal with.

Finally, in the last new line of the function, you reset the quiz, this time passing the number of questions to be answered (`noQuestions`) and time limit (`timeLimit`).

That completes all the changes for the page `QuizPage.htm`, so you can resave the page in your text editor and close it.

Now turn your attention to the `GlobalFunctions.htm` page and start by looking at the changes you need to make to the `resetQuiz()` function, as shown in the following:

```

function resetQuiz(numberOfQuestions, SelectedTimeLimit)
{
    timeLeft = SelectedTimeLimit;
    totalQuestionsToAsk = numberOfQuestions;
    var indexCounter;
    currentQNumber = -1;
    questionsAsked = new Array();
    for (indexCounter = 0; indexCounter < questions.length; indexCounter++)
    {
        questionsAsked[indexCounter] = false;
    }
    numberOfQuestionsAsked = 0;
    numberOfQuestionsCorrect = 0;
    if (timeLeft == -1)
    {

```

```
        window.status = "No Time Limit";
    }
    else
    {
        quizTimerId = window.setInterval("updateTimeLeft()",1000);
    }
}
```

The first change is to your function's definition. Previously it took no parameters, now it takes two: the number of questions to be answered and the time limit within which the quiz must be completed.

Your next change is to set two new global variables, `timeLeft` and `totalQuestionsToAsk`, to the values passed to the function. You'll see later that these global variables are used elsewhere to determine if enough questions have been asked and to check to see if the time limit has been reached.

The final change to this function is the setting of the timer that will monitor how much time is left. One of the options open to the user is no time limit at all, which is represented by the value `-1`. If the time limit is `-1`, you just put a message in the status bar of the browser window using the `window` object's `status` property. Note that on Netscape browsers the `No Time Limit` text in the status bar gets overwritten by `Document:Done` when the frames change. If, however, the time limit is not `-1`, you start a timer, using `setInterval()`, which will call the function `updateTimeLeft()` every second.

The `updateTimeLeft()` function is a new function, so let's create that now. Add it to the script block underneath all the other function definitions.

```
function updateTimeLeft()
{
    timeLeft--;
    if (timeLeft == 0)
    {
        alert("Time's Up");
        numberOfQuestionsAsked = totalQuestionsToAsk;
        window.top.fraQuizPage.location.href = "AskQuestion.htm";
    }
    else
    {
        var minutes = Math.floor(timeLeft / 60);
        var seconds = timeLeft - (60 * minutes);
        if (minutes < 10)
            minutes = "0" + minutes;
        if (seconds < 10)
            seconds = "0" + seconds;
        window.status = "Time left is " + minutes + ":" + seconds;
    }
}
```

This function does three things: It decrements the time left, it stops the quiz if the time left has reached 0, and finally it notifies the user of the time remaining by putting a message in the browser's status bar.

As you saw earlier, when the quiz is reset with the `resetQuiz()` function, the global variable `timeLeft` is set to the amount of time, in seconds, that the user has to complete the quiz. The `updateTimeLeft()` function is called every second, and in the first line you decrement the number of seconds left by one.

```
timeLeft--;
```

In the following `if` statement you check to see if `timeLeft` is 0. If it is, that means no seconds are left, and you end the quiz by setting the global variable `numberOfQuestionsAsked` to the same value as the number of questions the user wanted to answer, which is stored in global variable `totalQuestionsToAsk`. Then, when you navigate the page to `AskQuestion.htm`, that page will think that all the questions to be asked have been asked and will end the quiz rather than ask another question.

If there is still time left, the `else` part of the `if` statement executes and updates the status bar with the number of minutes and seconds left.

You need to split `timeLeft`, which is in seconds, into minutes and seconds. First you get the number of minutes using the following line:

```
var minutes = Math.floor(timeLeft / 60);
```

This returns just the whole-number part of the seconds when divided by 60, which is the number of minutes you need. You get the seconds in the following line:

```
var seconds = timeLeft - (60 * minutes);
```

This is just `timeLeft` (the total seconds) minus the number of seconds represented by the `minutes` value. So if `timeLeft` is 61, you have this:

```
minutes = 61 / 60 = 1.01667
```

The result is 1 as a whole number, followed by this:

```
seconds = 61 - (60 * 1) = 1
```

You want to display this as a string in the status bar in the format *minutes:seconds*, which is 01:01 in this case. However, just concatenating minutes and seconds will leave you with something like 1:1 when the value of either is less than 10. Currently minutes can't go above 5, so you could just add the 0 anyway, but by using an `if` statement, you are future-proofing it for the situation where you allow the user a time limit over nine minutes.

To fix this problem you add an extra 0 when the values are less than 10.

```
if (minutes < 10)
    minutes = "0" + minutes;
if (seconds < 10)
    seconds = "0" + seconds;
```

Finally, you change the value displayed in the window's status bar.

```
window.status = "Time left is " + minutes + ":" + seconds;
```

Chapter 9: Date, Time, and Timers

In the two preceding functions, you used some new globally defined variables you have not yet defined, so let's do that now.

```
var timeLeft = -1;
var totalQuestionsToAsk = 0;
var quizTimerId = 0;
```

Place these at the top of the script block.

Finally you've got just two small changes to make to the function `getQuestion()`, and that completes the trivia quiz for this chapter.

The first change is to the `if` statement at the top of the function.

```
if (totalQuestionsToAsk != numberOfQuestionsAsked)
{
    var questionNumber = Math.floor(Math.random() * questions.length);
```

Previously you asked another question as long as you had not used up all the questions available. Now you ask another question as long as the `totalQuestionsToAsk` variable does not equal the number of questions actually asked. The global variable `totalQuestionsToAsk` had its value determined by the user when she selected the number of questions she wanted to answer from the drop-down list. You passed this value to the `resetQuiz()` function, which did the actual setting of the variable's value.

The second change is to code in the `else` statement when the quiz is actually ended and you write out the summary of how the user did. Remember that you set a timer to keep track of how much time is left before the quiz must end. Well, now that the quiz has ended you need to stop that timer, which you do by using the `clearInterval()` method, passing the timer ID that you stored in the global variable `quizTimerId`, which you set when the timer was started. However, you mustn't try to stop the timer if no timer was set because the user did not select a time limit.

```
        currentQNumber = questionNumber;
        questionsAsked[questionNumber] = true;
    }
    else
    {
        if (timeLeft != -1)
        {
            clearInterval(quizTimerId);
        }
        questionHTML = "<h3>Quiz Complete</h3>";
        questionHTML = questionHTML + "You got " + numberOfQuestionsCorrect;
```

Well, that's all the changes made, so resave `GlobalFunctions.htm`. You can start the quiz by loading `TriviaQuiz.htm` into your browser.

Hopefully you should see a page like that shown in Figure 9-8.

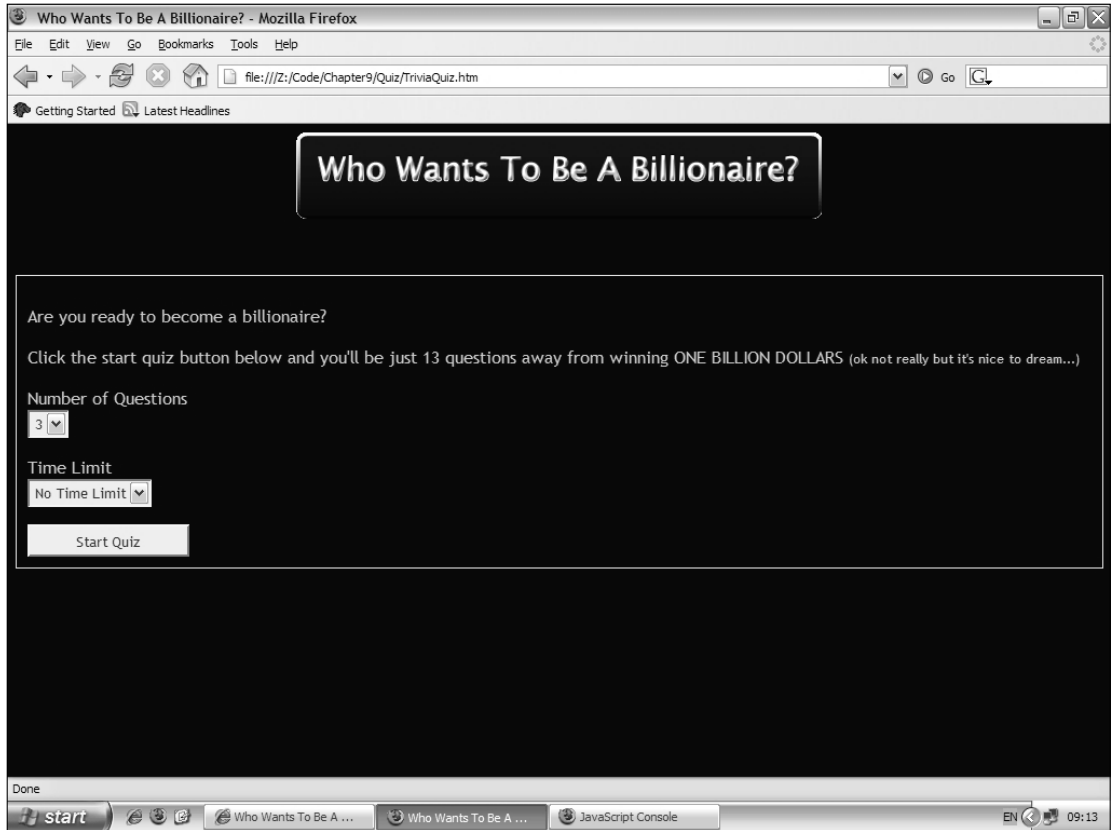


Figure 9-8

If you choose a time limit and then click the Start Quiz button, you'll be presented with the first question (randomly selected), and the timer, which is displayed in the status bar at the bottom of the page, will start counting down.

In Firefox and IE 7 browsers the ability of JavaScript to change the status bar text is disabled by default. To enable it in Firefox you'll need to go to Tools ⇨ Options and select the Content tab. Then, to enable JavaScript, click the Advanced Options button. Select the check box Change Status Bar Text. In IE7 you'll need to go to Tools ⇨ Internet Options. Then select the Security tab. Click Internet, then click custom level, and scroll down to the Allow status bar updates via script check box and select it.

That completes all the changes to the trivia quiz for this chapter. You'll be returning to the trivia quiz in Chapter 11 to see how you can store information on the user's computer so that you can provide a table of previous results.

Summary

You started the chapter by looking at Coordinated Universal Time (UTC), which is an international standard time. You then looked at how to create timers in web pages.

The particular points we covered were the following:

- ❑ The `Date` object enables you to set and get UTC time in a way similar to setting a `Date` object's local time by using methods (such as `setUTCHours()` and `getUTCHours()`) for setting and getting UTC hours with similar methods for months, years, minutes, seconds, and so on.
- ❑ A useful tool in international time conversion is the `getTimezoneOffset()` method, which returns the difference, in minutes, between the user's local time and UTC. One pitfall of this is that you are assuming the user has correctly set his time zone on his computer. If not, `getTimezoneOffset()` is rendered useless, as will be any local date and time methods if the user's clock is incorrectly set.
- ❑ Using the `setTimeout()` method, you found you could start a timer going that would fire just once after a certain number of milliseconds. `setTimeout()` takes two parameters: the first is the code you want executed, and the second is the delay before that code is executed. It returns a value, the unique timer ID that you can use if you later want to reference the timer; for example, to stop it before it fires, you use the `clearTimeout()` method.
- ❑ To create a timer that fires at regular intervals, you used the `setInterval()` method, which works in the same way as `setTimeout()`, except that it keeps firing unless the user leaves the page or you call the `clearInterval()` method.
- ❑ Finally, using your new knowledge of timers, you changed the trivia quiz so that the user can determine how many questions she wants to answer and within what time limit the quiz must be completed.

In the next chapter you'll be looking at the top seven mistakes of all time, ones every programmer makes at some point regardless of how much of an expert he or she may claim to be. You'll also be looking at a script debugger for IE and how it can be used to step through live code, line by line. Finally, you'll be looking at how to avoid errors and how to deal with errors when you get them.

Exercise Questions

Suggested solutions to these questions can be found in Appendix A.

Question 1

Create a web page with an advertisement image at the top. When the page loads, select a random image for that advertisement. Every four seconds, make the image change to a different one, making sure a different advertisement is selected until all the advertisement images have been seen.

Question 2

Create a form that gets the user's date of birth. Then, using that information, tell her on what day of the week she was born.

10

Common Mistakes, Debugging, and Error Handling

Even a JavaScript guru makes mistakes, even if they are just annoying typos. In particular, when code expands to hundreds of lines, the chance of something going wrong becomes much greater. In proportion, the difficulty in finding these mistakes, or bugs, also increases. In this chapter you will look at various techniques that will help you minimize the problems that arise from this situation.

You'll start by taking a look at the top seven JavaScript coding mistakes. After you know what they are, you'll be able to look out for them when writing code. Hopefully, so that you won't make them so often!

Then you'll look at the Microsoft script debugger, which can be used with Internet Explorer. You'll see how you can use it to step through your code and check the contents of variables while the code is running, a process that enables us to hunt for difficult bugs. You'll also take a briefer look at the debugging tools available for Firefox.

Finally, you'll look at how you can cope with errors when they do happen, so that you prevent users from seeing your coding mistakes.

I Can't Believe I Just Did That: Some Common Mistakes

It's time to do a rundown of seven common mistakes. Some of these you'll learn to avoid as you become more experienced, but others may haunt you forever!

You'll find it very useful in this chapter if your browser is set up to show errors. You did this in Chapter 2 in the section "Setting Up Your Browser for Errors." So if you don't already have error display set up, now would be a good time to do so.

1: Undefined Variables

JavaScript is actually very easygoing when it comes to defining your variables before assigning values to them. For example, the following will implicitly create the new variable `abc` and assign it to the value 23.

```
abc = 23;
```

Although strictly speaking, you should define the variable explicitly.

```
var abc = 23;
```

(Actually, whether you use the `var` keyword has a consequence on the scope that the variable has, so in fact it is always best to use the `var` keyword.)

However, if a variable is actually used before it has been defined, an error will arise. For example, the following code will cause the error shown in Figure 10-1 if the variable `abc` has not been previously defined (explicitly or implicitly).

```
alert(abc);
```

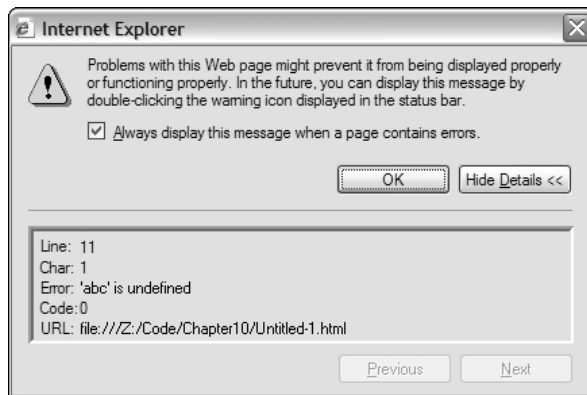


Figure 10-1

In Firefox you'll need to look in the JavaScript console, which you can view by choosing Tools → JavaScript Console.

In addition, you must remember that function definitions also have parameters, which if not declared correctly can lead to the same type of error.

Take a look at the following code:

```
function resetQuiz(numberOfQuestions, timeLimit)
{
    timeLeft = timeLimit;
    totalQuestionsToAsk = numberOfQuestions;
    currentQNumber = -1;
    questionsAsked = new Array();
```

```
numberOfQuestionsAsked = 0;  
numberOfQuestionsCorrect = 0;  
}
```

If you call this function, you get an error message similar to the one shown in Figure 10-2.

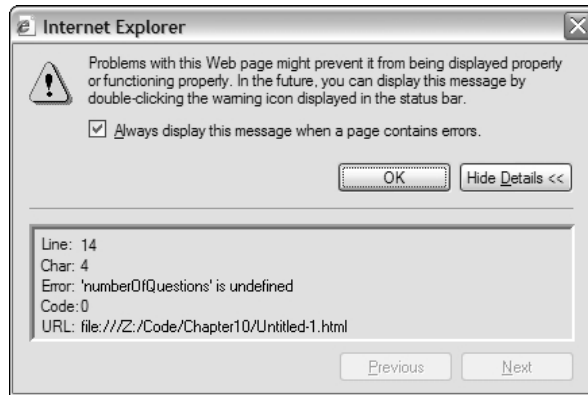


Figure 10-2

The error here is actually a simple typo in the function definition. The first parameter has the typo: it should read `numberOfQuestions`, not `numberOfQustions`. What can be confusing with this type of error is that although the browser tells us the error is on one line, in fact the source of the error is on some other line.

2: Case Sensitivity

This is a major source of errors, particularly because it can be difficult to spot at times.

For example, spot the three case errors in the following code:

```
var myName = "Paul";  
If (myName == "paul")  
    alert(myName.toUpperCase());
```

The first error is that you have typed `If` rather than `if`. However, JavaScript won't tell us that the error is an incorrect use of case, but instead IE will tell us `Object expected` and Firefox will tell us that `If` is not defined. Although error messages give us some idea of what's gone wrong, they often do so in an oblique way. In this case IE thinks you are trying to use an object called an `If` object and Firefox thinks you are trying to use an undefined variable called `If`.

Okay, with that error cleared up, you come to the next error, not one of JavaScript syntax, but a logic error. Remember that `Paul` does not equal `paul` in JavaScript, so `myName == "paul"` is `false`, even though it's quite likely that you didn't care whether the word is `paul` or `Paul`. This type of error will result in no error message at all, just the code not executing as you'd planned.

The third fault is with the `toUpperCase()` method of the `String` object contained in `myName`. You've written `toUpperCase`, with the `C` in lowercase. IE will give us the message `Object doesn't support`

Chapter 10: Common Mistakes, Debugging, and Error Handling

this property or method and Firefox will report that `myName.toUpperCase` is not a function. On first glance it would be easy to miss such a small mistake and start checking your JavaScript reference guide for that method. You might wonder why it's there, but your code is not working. Again, you always need to be aware of case, something that even experts get wrong from time to time.

3: *Incorrect Number of Closing Braces*

In the following code you define a function and then call it. However, there's a deliberate mistake. See if you can spot where it is.

```
function myFunction()
{
  x = 1;
  y = 2;
  if (x <= y)
  {
    if (x == y)
    {
      alert("x equals y");
    }
  }
  myFunction();
}
```

If you properly format the code you'll have a much easier time spotting the error.

```
function myFunction()
{
  x = 1;
  y = 2;
  if (x <= y)
  {
    if (x == y)
    {
      alert("x equals y");
    }
  }
}
myFunction();
```

Now you can see that you've forgotten to mark the end of the function with a closing curly brace. When there are a lot of `if`, `for`, or `do while` statements, it's easy to have too many or too few closing braces. With proper formatting, this problem is much easier to spot.

4: *Missing Plus Signs During Concatenation*

In the following code, there's a deliberate concatenation mistake.

```
var myName = "Paul";
var myString = "Hello";
var myOtherString = "World";
myString = myName + " said " + myString + " " myOtherString;
alert(myString);
```


There should be a `+` operator between `" "` and `myOtherString` in the fourth line of code.

Although easy to spot in just a few lines, this kind of mistake can be harder to spot in large chunks of code. Also, the error message that a mistake like this causes can be misleading. Load this code into a browser and you'll be told `Error : Expected by IE and Missing ; before statement` by Firefox. It's surprising how often this error crops up.

5: Equals Rather than Is Equal To

Take a look at the following code:

```
var myNumber = 99;
if (myNumber = 101)
{
    alert("myNumber is 101");
}
else
{
    alert("myNumber is " + myNumber);
}
```

You'd expect, at first glance, that the `alert()` method in the `else` part of the `if` statement would execute, telling us that the number in `myNumber` is 99, but it won't. You've made the classic "one equals sign instead of two equals signs" mistake. Hence, instead of comparing `myNumber` with 101, you have set `myNumber` to equal 101. If, like me, you program in languages, such as Visual Basic, that use only one equals sign for both comparison and assignment, you'll find that every so often this mistake crops up. It's just so easy to make.

What makes things even trickier is that no error message will be raised; it is just your data and logic that will suffer. Assigning a variable a value in an `if` statement may be perverse, but it's perfectly legal so there will be no complaints from JavaScript. When embedded in a large chunk of code, a mistake like this is easily overlooked. Just remember, next time your program's logic seems crazy, that it's worth checking for this error.

6: Incorrect Number of Closing Parentheses

Take a look at the following code:

```
if (myVariable + 12) / myOtherVariable < myString.length)
```

Spot the mistake?

The problem is that you've missed a parenthesis at the beginning. You want `myVariable + 12` to be calculated before the division by `myOtherVariable` is calculated, so quite rightly you know you need to put it in parentheses.

```
(myVariable + 12) / myOtherVariable
```

Chapter 10: Common Mistakes, Debugging, and Error Handling

However, the `if` statement's condition must also be in parentheses. Not only is the initial parenthesis missing, but also you have one more closing parenthesis than you have opening parentheses; the numbers must match. For each parenthesis opened, there must be a corresponding closing parenthesis. Your code should be as follows:

```
if ((myVariable + 12) / myOtherVariable < myString.length)
```

When you have lots of opening and closing parentheses, it's very easy to miss one or have one too many.

7: Using a Method as a Property and Vice Versa

The final common error is where either you forget to put parentheses after a method with no parameters, or you use a property and do put parentheses after it.

When calling a method you must always have parentheses following its name; otherwise JavaScript thinks that it must be a property. For example, examine the following code:

```
var nowDate = new Date;  
alert(nowDate.getMonth);
```

In the first line you have used the `Date` constructor, which is simply a method of the `Date` object, with no parentheses.

On the second line, you call the `getMonth()` method of the `Date` object, except that you've forgotten the parentheses here also.

This is how lines should be:

```
var nowDate = new Date();  
alert(nowDate.getMonth());
```

Just as you should always have parentheses after a method, you should never have parentheses after a property, otherwise JavaScript thinks you are trying to use a method of that object:

```
var myString = new String("Hello");  
alert(myString.length());
```

In the second line you have used parentheses after the `length` property, making JavaScript think it is a method. You should have written it like this:

```
var myString = new String("Hello");  
alert(myString.length);
```

This mistake may seem like an obvious one in two lines, but it's easy to slip up when you're pounding out lots and lots of code.

Now that you've seen these top seven mistakes, you'll take a look at one way to make remedying them easier. This is through the use of the Microsoft script debugger.

Microsoft Script Debugger

The Microsoft script debugger for use with IE is a very useful tool for discovering what's gone wrong and why. Using it, you can halt the execution of your script and then step through code line by line to see exactly what is happening.

You can also find out which data are being held in variables and execute statements on the fly. Without the script debugger, the best you can do is use the `alert()` method in your code to show the state of variables at various points.

The script debugger works with IE 5+.

Obtaining the Script Debugger

You can currently download the script debugger from the following URL:

`www.microsoft.com/downloads/details.aspx?FamilyID=2f465be0-94fd-4569-b3c4-dffdf19ccd99&displaylang=en`

If the URL changes, a search for “script debugger” on the Microsoft web site, `www.microsoft.com`, ought to find its new home.

Other programs, such as Microsoft Visual Studio, also come with a built-in script debugger so there's no need to install this one.

In addition, Windows 2000 automatically comes with the script debugger, although it may not be set up on your system. To install it, open the Control Panel and choose Add/Remove Programs. Click the Add/Remove Windows Components button, and in the window that opens, scroll down the list to the script debugger. If it is not checked, then check it and click Next to install it.

To see if the script debugger is already installed, launch Internet Explorer and select the View menu. If one of the menu options is Script Debugger, as shown in Figure 10-3, the debugger is installed. If you do not find this menu option, it is still possible that the script debugger is already installed, but disabled. See the end of the next section to learn how to enable the script debugger, and then check for the menu option again.

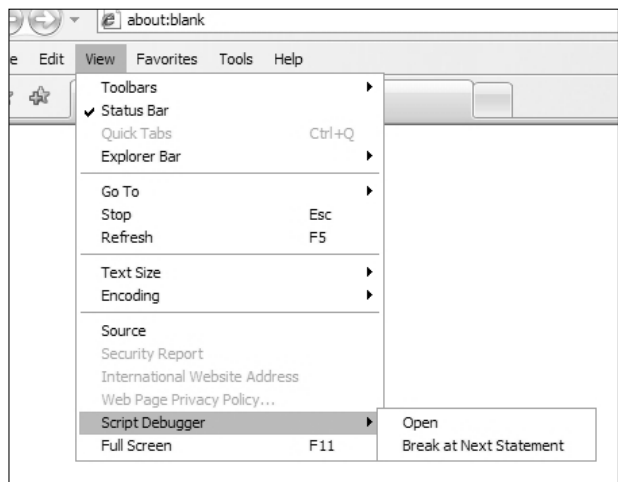


Figure 10-3

Installing the Script Debugger

After downloading the script debugger, you need to install it. First, you need to run the file you have just downloaded, for example from the Windows Start bar’s Run menu option. You should then see the dialog box shown in Figure 10-4, asking whether you want to install the debugger: Click the Yes button.

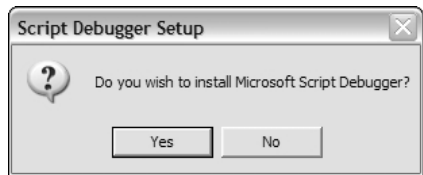


Figure 10-4

Next the license screen appears, as shown in Figure 10-5. Check the license, and then click the Yes button to agree to the conditions and install the debugger.



Figure 10-5

Next you get to choose where you want to install the debugger (see Figure 10-6). Anywhere on your local machine is fine.

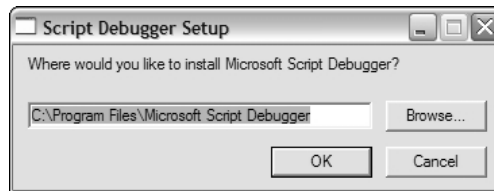


Figure 10-6

Click OK, and if a screen appears asking whether you want to create the directory, just click Yes.

The script debugger will now install. When it's complete, you see the message shown in Figure 10-7.

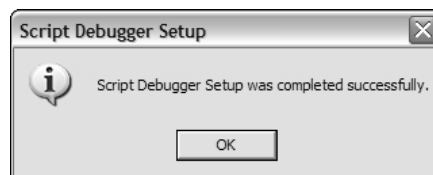


Figure 10-7

Click OK and if asked whether you want to restart the computer, click Yes.

Chapter 10: Common Mistakes, Debugging, and Error Handling

After the computer has restarted, open Internet Explorer. If you go to the View menu, you should see the Script Debugger option. If not, the script debugger may be disabled. To enable it in IE 5 and later, go to Tools ⇨ Internet Options. Then select the Advanced tab to see a screen similar to that shown in Figure 10-8.

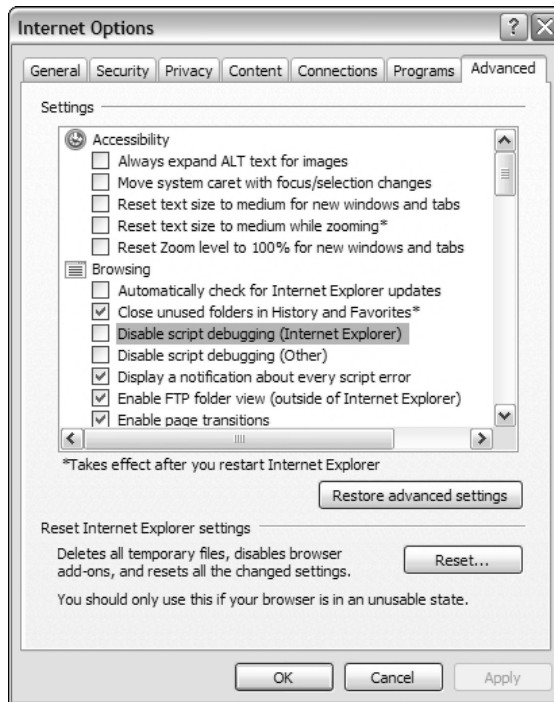


Figure 10-8

Make sure that the Disable script debugging (Internet Explorer) check box is cleared, as shown in Figure 10-8. Click OK, and then close the browser. When you reopen the browser you should see the Script Debugger option in the View menu.

Using the Script Debugger

It's important to point out that there are actually two versions of the script debugger: the basic version that you installed in the previous section and a more sophisticated version that comes with programs like Microsoft Visual Studio .Net. The more sophisticated version does everything that the basic version does, but the screen layout and look will vary slightly, as will some of the keys and icons. You'll be looking at just the basic version here, so all screenshots are applicable to that.

Opening a Page in the Debugger

You can open a page in the script debugger in a number of ways, but here you'll just look at the three that are most useful. However, before you start let's create a page you can debug. Note the deliberate typo in line 14. Be sure to include this typo if creating the page from scratch.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<script language="JavaScript" type="text/javascript">
function writeTimesTable(timesTable)
{
    var counter;
    var writeString;
    for (counter = 1; counter < 12; counter++)
    {
        writeString = counter + " * " + timesTable + " = ";
        writeString = writeString + (timesTable * counter);
        writeString = writeString + "<br>";
        documents.write(writeString);
    }
}
</script>
</head>
<body>
<div><script language=JavaScript type="text/javascript">
    writeTimesTable(2)
</script>
</div>
</body>
</html>
```

Save this page as `debug_timestable.htm`.

When you load this page into Internet Explorer, you'll discover the first way of activating the script debugger: It is activated automatically when there is an error in your code.

You should see a message box, similar to that shown in Figure 10-9, asking whether you wish to debug. Click Yes.

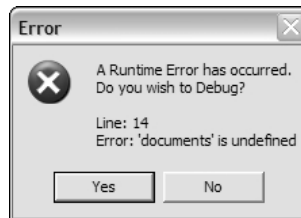


Figure 10-9

Having said that you want to debug, you should see the screen shown in Figure 10-10. The debugger has opened and stopped on the line where the error is and highlighted it in yellow, although this may not be obvious from the black-and-white screenshots in this chapter. The deliberate mistake is that you've written `documents.write` rather than `document.write`. The view is read-only so you can't edit it here; you need to return to your text editor to correct it. Let's do that, and then reload the page.

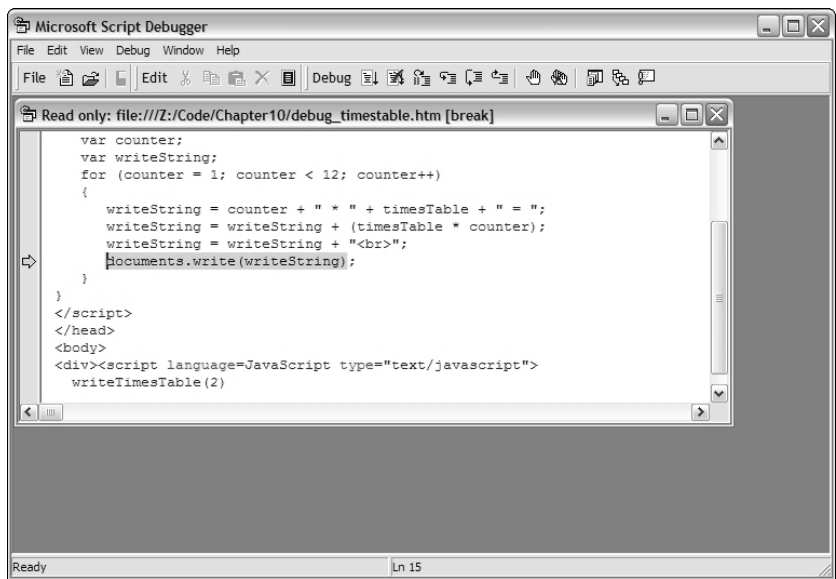


Figure 10-10

Having corrected the mistake and reloaded the page, you should see the two times table in your web page, as shown in Figure 10-11.

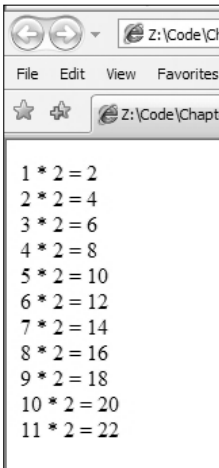


Figure 10-11

The second method you’re going to use to open the debugger begins with loading the page you want to debug into your browser; then you use the Break at Next Statement menu option under Script Debugger on the Internet Explorer View menu, as shown in Figure 10-12.

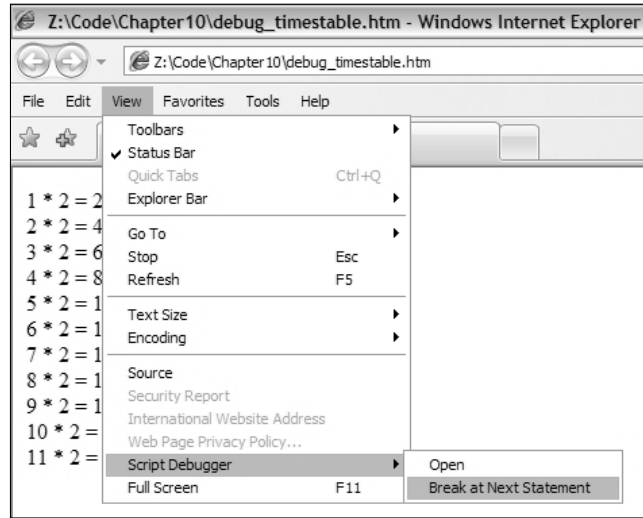


Figure 10-12

You've already got `debug_timestable.htm` loaded, so select **View** ⇨ **Script Debugger** ⇨ **Break at Next Statement**. Not much appears to happen, but reload the page by clicking the refresh icon or pressing F5, and the debugger will open at the next JavaScript statement executed.

Where the next JavaScript statement occurs in your page depends on your code. If you have code in the page other than a function or code connected to an event handler, the first line the browser executes will be the next JavaScript statement. In your case, this is the code calling the following function:

```
<script>writeTimesTable(2)</script>
```

If there is no code in the page except code inside event handlers or functions, the next statement executed will be that fired in an event handler, such as the `window` object's `onload` event handler or a button's `onclick` event handler.

Note that with some setups of the script debugger, the browser brings up a dialog box saying that an exception of type "Runtime Error" was not handled. This does not mean that there is an error in the code, but is simply the way **Break at Next Statement** works.

As you can see from Figure 10-13, the next statement executed in your example occurs when the browser reaches the script embedded in your web page that calls the `writeTimesTable()` function. Again, this script is highlighted in yellow by the debugger.

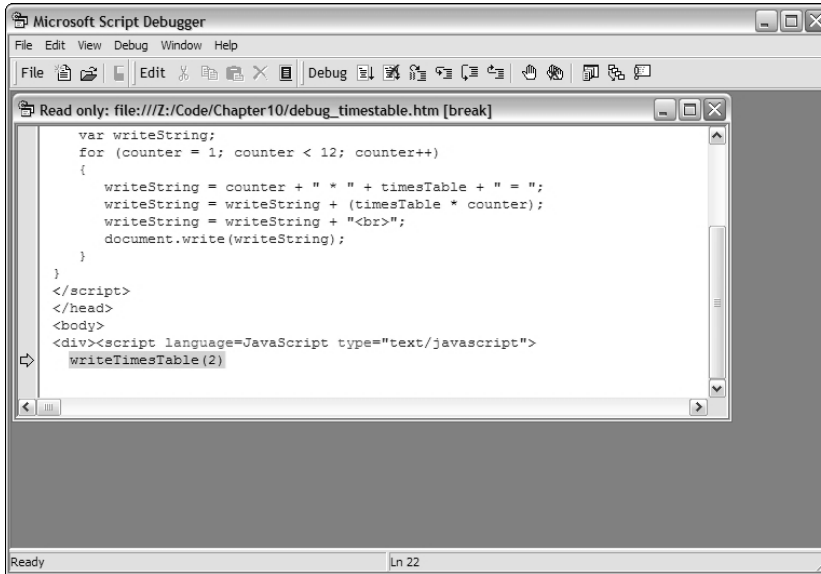


Figure 10-13



You're now going to use the Step Into icon, illustrated here, which can be found on the top toolbar.

Click this icon and the debugger will execute the current line of code and move to the next line, in this case your function. (You'll look more fully at stepping through code and examining the contents of variables shortly.) You can also press the F8 key to step into the code, which is often easier than clicking the icon.

Finally, let's look at a third way of opening the debugger, which is probably the easiest and most useful.

Imagine you want to stop your code's execution and open the debugger just before the `for` loop is executed. You can do this by simply adding the keyword `debugger` to your script, which will stop the execution at that point and open the debugger. Let's do that now.

You need to close the debugger, return to your text editor, and add the following code to `debug_timestable.htm`:

```
function writeTimesTable(timesTable)
{
    var counter;
    var writeString;
    debugger
    for (counter = 1; counter < 12; counter++)
    {
        writeString = counter + " * " + timesTable + " = ";
```

```
        writeString = writeString + (timesTable * counter);  
        writeString = writeString + "<br>";  
        document.write(writeString);  
    }  
}
```

Now refresh the page in Internet Explorer, and you'll find that the debugger opens, with code execution paused on the line with the `debugger` keyword in it (see Figure 10-14).

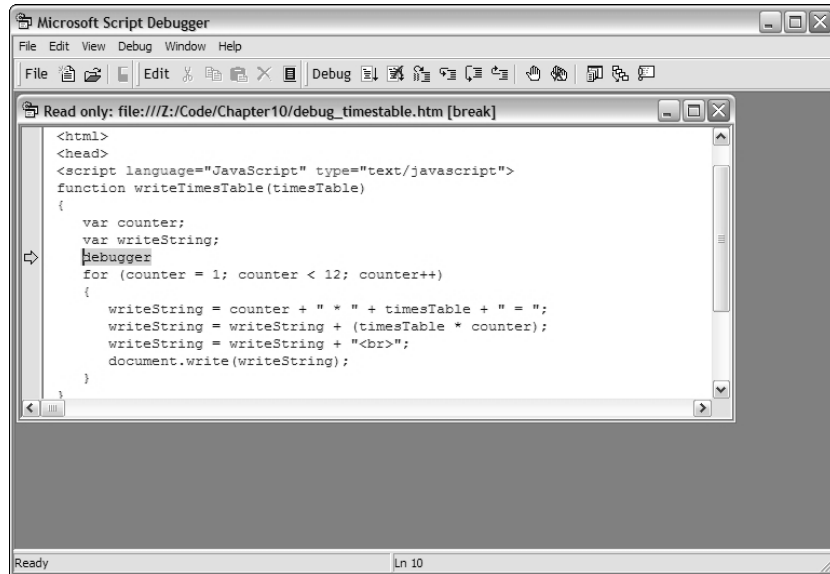



Figure 10-14

Again you can click the Step Into icon you used previously, and watch the code execute statement by statement.

Stepping Through Code

There are three important ways of stepping through code, each involving one of the icons from the top toolbar of the script debugger.

You've seen that one way is to step into the code. This simply means that every line of code is executed on a line-by-line basis. If a function is called, you step into that function and start executing the code inside the function statement by statement, before stepping out again at the end and returning to the calling line. To do this you use the Step Into icon.

 You may find that having stepped into a function you get halfway through and decide the function is not the source of the bug, and that you want to execute the remaining lines of code in the function, then continue step by step from the point at which the function was called. This is called stepping out of the function, and to do it you use the Step Out icon.

Chapter 10: Common Mistakes, Debugging, and Error Handling

Instead of the icon you can press Ctrl+Shift+F8.



There may also be times when you have some code with a bug in it that calls a number of functions. If you know that some of the functions are bug-free, then you may want to just execute those functions instead of stepping into them and seeing them executed line by line. For this the debugger has the Step Over icon, which executes the code within a function but without your having to go through it line by line.

Or you can press Shift+F8.

Let's alter your times-table code in `debug_timestable.htm` and demonstrate the three kinds of stepping action. Note that the `debugger` keyword has been removed from the `writeTimesTable()` function and is now in the second script block.

```
<html>
<head>
<script language=JavaScript type="text/javascript">
function writeTimesTable(timesTable)
{
    var counter;
    var writeString;
    for (counter = 1; counter < 12; counter++)
    {
        writeString = counter + " * " + timesTable + " = ";
        writeString = writeString + (timesTable * counter);
        writeString = writeString + "<BR>";
        document.write(writeString);
    }
}
</script>
</head>
<body>
<div><script language=JavaScript type="text/javascript">
var timesTable;
debugger
for (timesTable = 1; timesTable <= 12; timesTable++)
{
    document.write("<P>")
    writeTimesTable(timesTable)
    document.write("</P>")
}
</script></div>
</body>
</html>
```

Save this as `debug_timestable2.htm`. When you load it into your browser, the script debugger will be opened by the `debugger` statement, as shown in Figure 10-15.

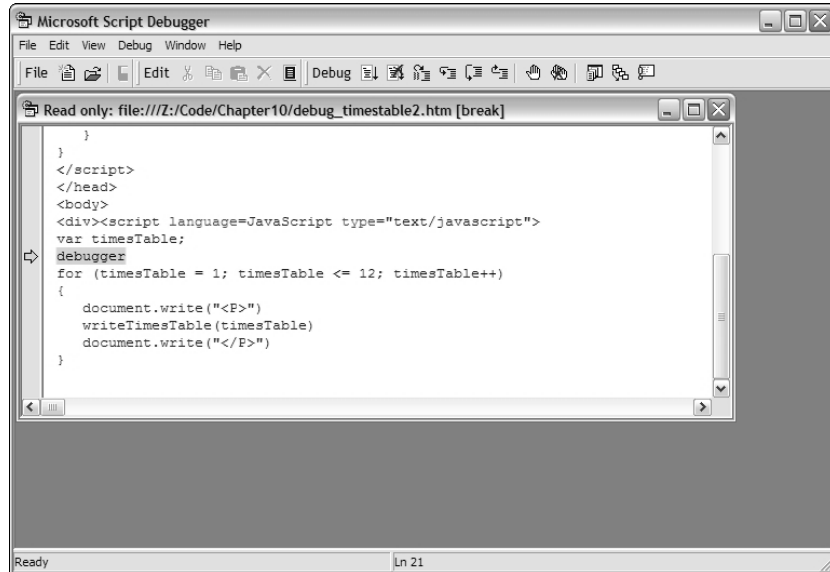


Figure 10-15

Click the Step Into icon and code execution will move to the next statement. In this case, the next statement is the first statement in the `for` loop in which you initialized the variable `timesTable` to the value of 1, as shown in Figure 10-16.

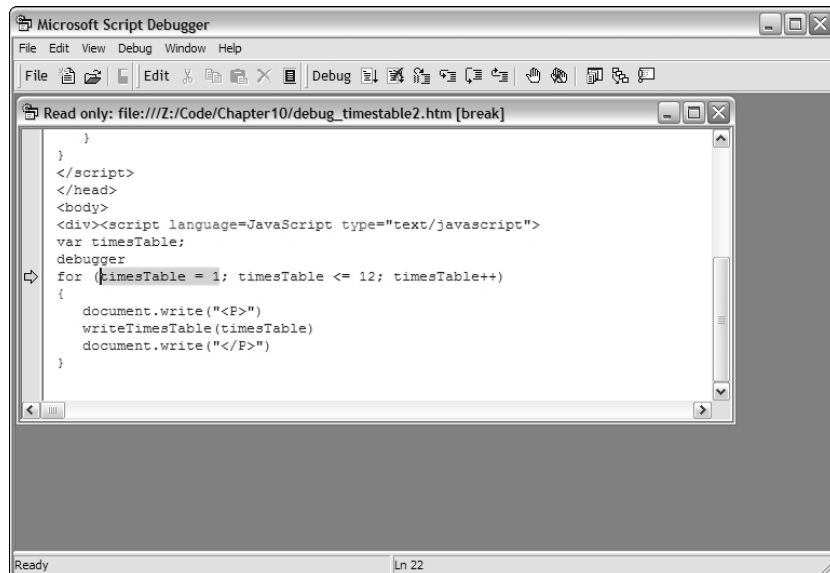


Figure 10-16

Chapter 10: Common Mistakes, Debugging, and Error Handling

When you click the Step Into icon again, the `timesTable = 1` statement is executed and you step to the next statement due to be executed, which is the condition part of the `for` loop. With `timesTable` set to 1, you know that the condition `timesTable <= 12` is going to be `true`. Click the Step Into icon and the condition executes and indeed you find you're right, the condition is `true`. Now the first statement inside the `for` loop, `document.write("<P>")`, is up for execution.

When you click the Step Into icon again it will take you to the first calling of your `writeTimesTable()` function. You want to see what's happening inside that function, so click Step Into again and you'll step into the function. Your screen should look like the one shown in Figure 10-17.

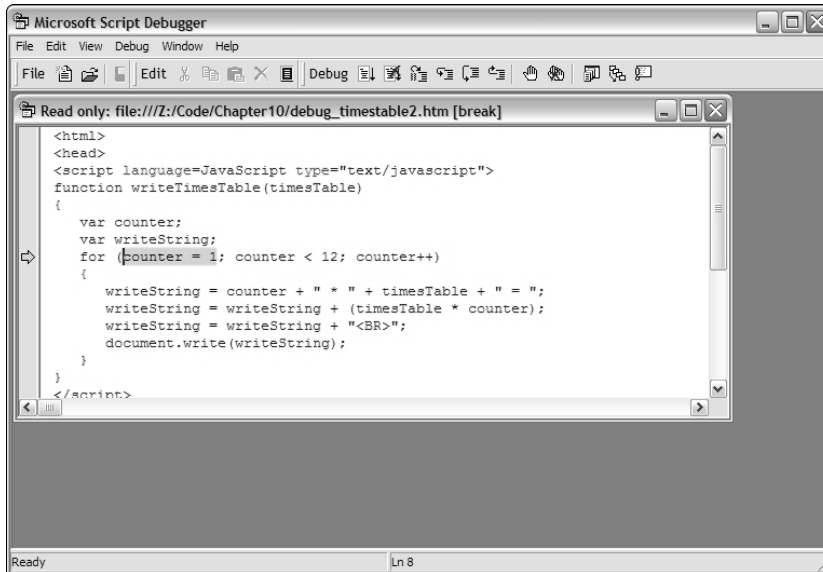


Figure 10-17

The next statement to be executed is not the `var counter;` or `var writeString;` line, but instead the `for` loop's initialization condition. The first two lines have been executed, but the script debugger does not enable us to step through variable declarations line by line.

Click the Step Into icon a few times to get the gist of the flow of execution of the function. In fact, stepping through code line by line can get a little tedious. So let's imagine you're happy with this function and want to run the rest of it. Start single-stepping from the next line after the function was called. To do this, click the Step Out icon.

Now the function has been fully executed, and you're back out of it and at the next line, `document.write("</P>")`, as you can see from Figure 10-18.

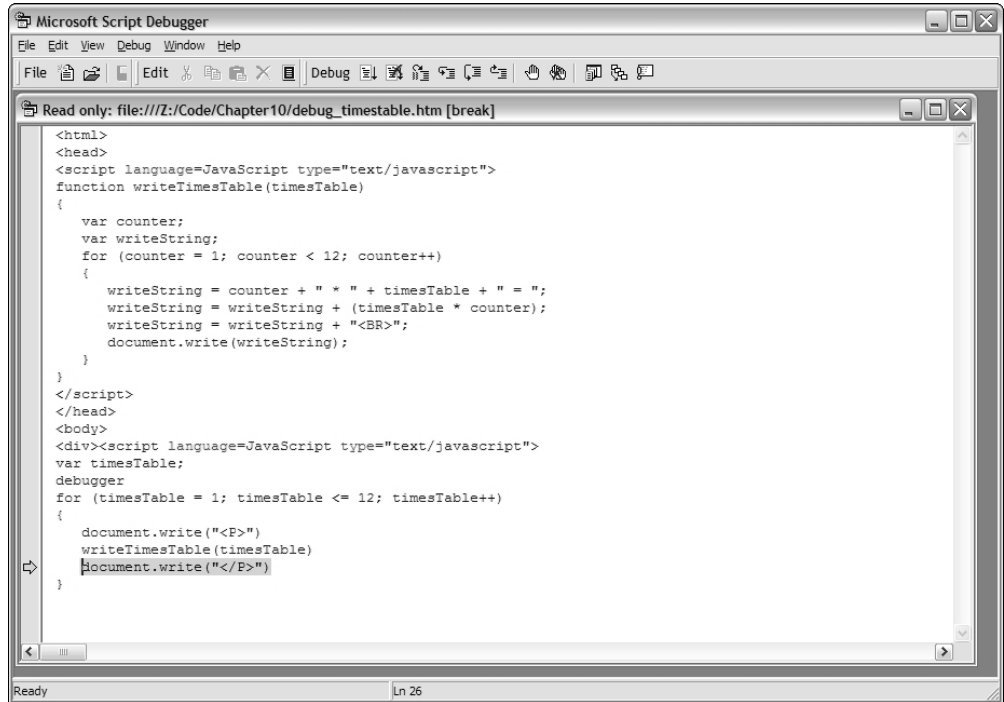



Figure 10-18

Click the Step Into icon to see that the `document.write()` line will be executed, and the next statement in the flow of execution is the increment part of your `for` loop. Click Step Into again and execution will continue to the condition part of the `for` loop. Clicking Step Into twice more brings you back to the calling of the `writeTimesTable()` function. You've already seen this in action, so really you want to step over it and go to the next line. Well, no prizes for guessing that the Step Over icon is what you need to click to do this.

Click the Step Over icon and the function will be executed, but without your having to step through it statement by statement. You should find yourself back at the `document.write("</P>")` line.

 If you've finished debugging, you can run the rest of the code without stepping through each line by clicking the Run icon on the toolbar, shown after this paragraph. Let's do that; then you can return to the browser and see the results of the code you have executed. You should see a page of times tables from $1 \times 1 = 1$ to $11 \times 12 = 132$ in the browser.

Instead of clicking the icon you can also press the F5 key.

Breakpoints

Breakpoints are markers you can set in the debugger that force code execution to stop at that point and start single-stepping through the code.

Load your `debug_timestable2.htm` page into the browser. This will open the debugger and stop execution at the line with your `debugger` statement. Now imagine that you want to stop in your `writeTimesTable()` function on the line that writes the results of the times table to the page, namely `document.write(writeString)`, as shown in Figure 10-19. This is the last statement in the `for` loop. However, we're busy people and don't want to manually step through every line before that. What you can do is set a breakpoint on that line and then click the Run icon, which will restart the code execution in the normal fashion (that is, without single-stepping). Then, when the breakpoint is reached, code execution will stop and you can start single-stepping if you want.



To set the breakpoint, you need to scroll up the code window in the debugger until you can see the line on which you want to put the breakpoint. Click that line; then click the Toggle Breakpoint icon on the toolbar, shown here.

Any line with a breakpoint on it is indicated by the reddish-brown dot on the left of the code window and by the line itself being set to a reddish brown, although the line may not always be colored. You can set as many or as few breakpoints at one time as you wish, so if you want to break on other lines you can add breakpoints there too.



To unset a breakpoint you just click the relevant line of code and click the Toggle Breakpoint icon again, and that toggles it off. To clear all breakpoints at once you can click the Clear All Breakpoints icon shown here (see Figure 10-19).

Okay, now let's start the code running again by clicking the Run icon in the toolbar.

You'll find that the code resumes executing without single-stepping until it reaches your breakpoint, at which point it stops. You can either single-step using the Step Into icon or click the Run icon again, in which case execution continues unless another breakpoint is reached.

Leave the debugger open with code execution halted on your breakpoint; you'll be using it in a moment.

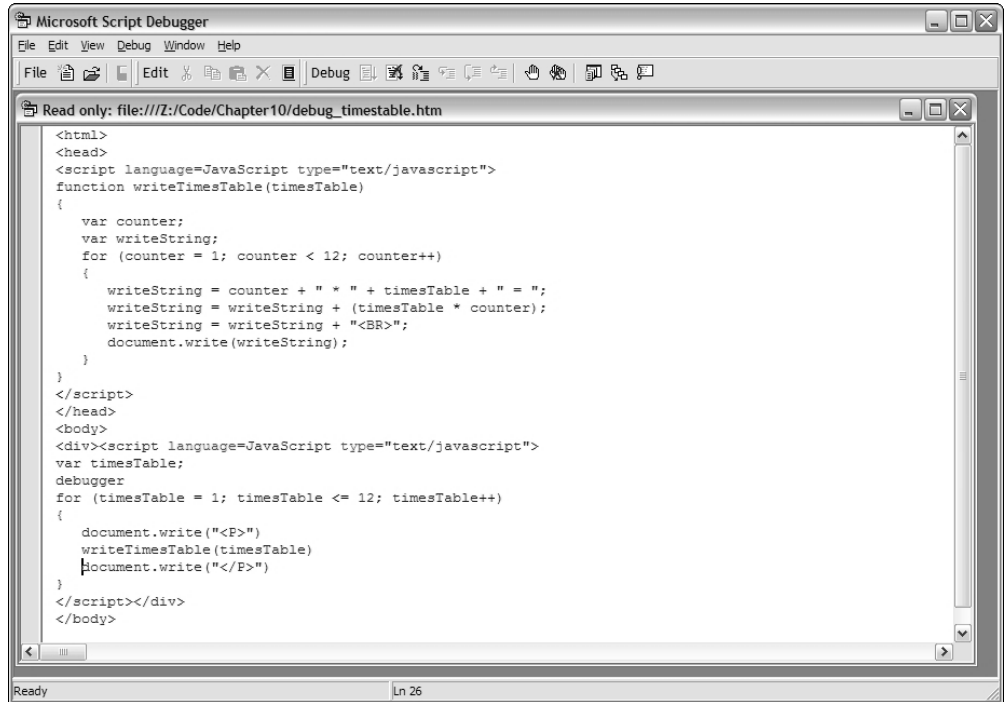


Figure 10-19

The Command Window

While you're stepping through code and checking its flow of execution, what would be really useful is the ability to check the values contained inside variables, to evaluate conditions, and even to change things on the fly. You can do all of these things using the debugger's command window.

Hopefully you still have the debugger open with execution halted at the breakpoint you set previously. The line stopped on in Figure 10-19 is repeated here:

```
document.write(writeString);
```

Let's see how you can find out the value currently contained in the variable `writeString`.



First you need to open the command window from within the debugger. You do this by clicking the Command Window icon, illustrated here, or by selecting Command Window from the View menu.

In the command window, type the name of the variable you want to examine, in this case `writeString`; then click Enter. This will cause the value contained in the variable to be printed below your command in the command window, as shown in Figure 10-20.

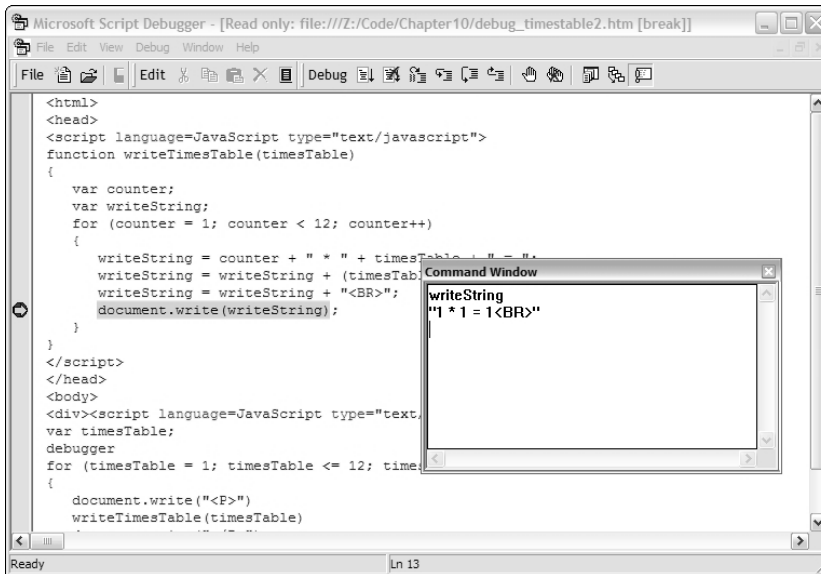


Figure 10-20

If you want to change a variable, you can write a line of JavaScript into the command window and press Enter. Try it with the following code:

```
writeString = "Changed on the Fly<BR>"
```

Now remove the breakpoint (see the previous section for instructions) and click the Run icon. If you switch to the browser, you see the results of your actions: Where the 1*1 times table result should be, the text you changed on the fly has been inserted. Note that this alteration does not change your actual HTML source file, just the page currently loaded in the browser.

The command window can also evaluate conditions. Refresh the browser to reset the debugger and leave execution stopped at your debugger statement. Click the Step Into icon twice and execution will stop on the condition in the `for` statement.

Type the following into the command window and press Enter:

```
timesTable <= 12
```

Because this is the first time the loop has been run, as shown in Figure 10-21, `timesTable` is equal to 1 and so the condition `timesTable <= 12` evaluates to `true`. Note that the debugger sometimes represents `true` by the value `-1` and `false` by the value `0`, so you may see the value `-1` instead of `true`.

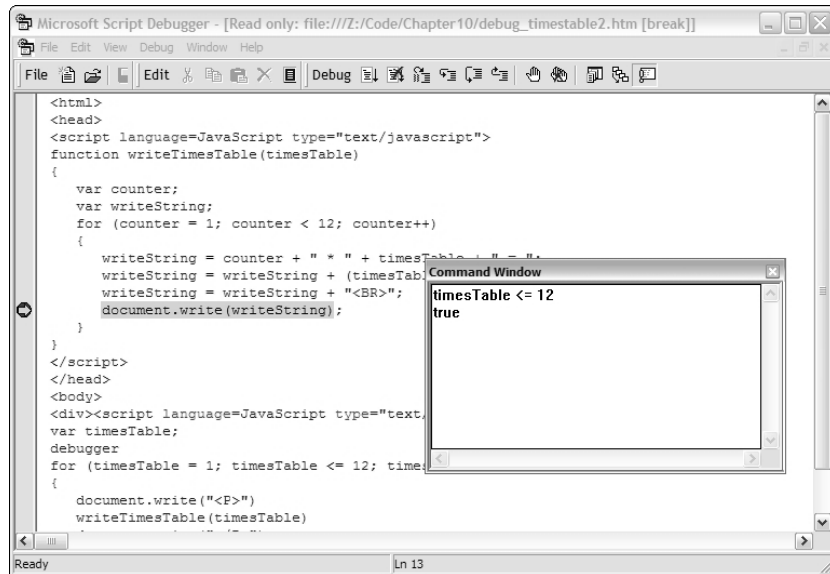


Figure 10-21

You can also use the command window to access properties of the browser's Browser Object Model (BOM). For example, if you type `window.location.href` into the command window and press Enter, it will tell you where the current page is stored.

In fact, the command window can execute any single line of JavaScript including functions.

Call Stack Window

When you are single-stepping through the code, the call stack window keeps a running list of which functions have been called to get to the current point of execution in the code.

Let's create an example web page that demonstrates the call stack very nicely.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<script language=JavaScript type="text/javascript">
function firstCall()
{
    secondCall();
}
function secondCall()
{
    thirdCall();
}
function thirdCall()
{
```

Chapter 10: Common Mistakes, Debugging, and Error Handling

```
//
}
function button1_onclick()
{
    debugger
    firstCall();
}
</script>
</head>
<body>
<input type="button" value="Button" name=button1
    onclick="return button1_onclick()">
</body>
</html>
```

Save this page as `debug_callstack.htm`, and then load it into IE. After you've done this, all you'll see is a blank web page with a button. Click the button and the debugger will be opened at the `debugger` statement in the `button1_onclick()` function, which is connected to the button's `onclick` event handler.



To open the call stack window, click the Call Stack icon in the toolbar, shown here, or choose Call Stack from the View menu.

Your debugger now looks like what is shown in Figure 10-22.

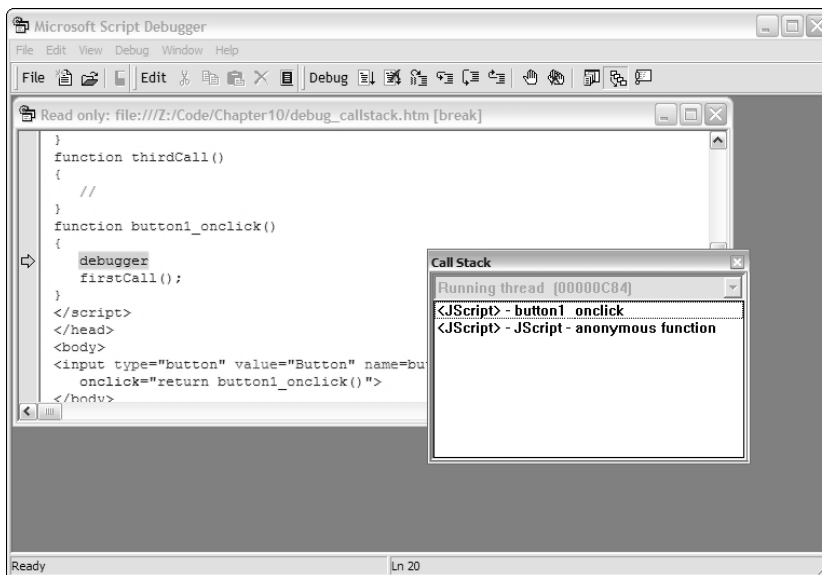


Figure 10-22

Chapter 10: Common Mistakes, Debugging, and Error Handling

Every time a function is called, the debugger adds the function to the top of the call stack. You can already see that the first function called was actually the code attached to the `onclick` event handler of your button. The anonymous function is the event handler code that calls your `onclick` function. Next, added to the call stack is the function called by the `onclick` event handler, which is the function `button1_onclick()` shown at the top of the call stack.

If you want to see where each function was first entered, you need only double-click the function name in the call stack window. Double-click `<Jscript> - Jscript - anonymous function` and the calling line—that is, the code connected to the `onclick` attribute of the `<input>` tag—will be shown. Now double-click the top line—`<Jscript> - button1_onclick`—and that will take you back to the current execution point.

Now single-step twice, using the Step Into icon. The first step is to the line that calls the `firstCall()` function. The second step takes you into that function itself. The function is immediately added to the call stack, as shown in Figure 10-23.

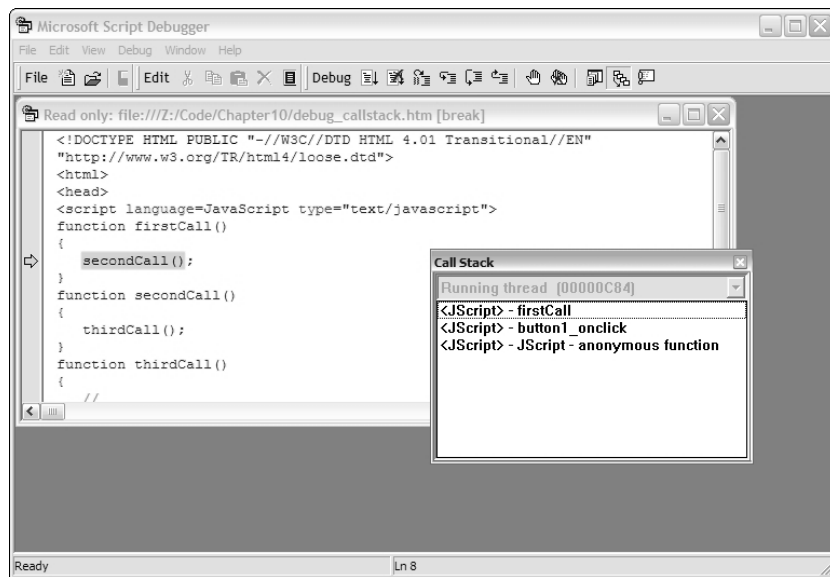


Figure 10-23

Click the Step Into icon again and you'll step into the second function, `secondCall()`. Again this is added to the call stack. One more click takes you into the third function, `thirdCall()`, again with its name being added to the top of the call stack.

Now click Step Into again and as you leave the function `thirdCall()` you will see that its name is removed from the top of the call stack. Another click takes you out of the second function `secondCall()`, whose name is also now removed from the stack. Each additional click takes you out of a function, and removes its name from the call stack, until eventually all the code has been executed and you're back to the browser again.

Chapter 10: Common Mistakes, Debugging, and Error Handling

Your demo page was very simple to follow, but with complex pages, especially multi-frame pages, the call stack can prove very useful for tracking where you are, where you have been, and how you got there.

Running Documents Window

The final window you'll look at is the running documents window. This window lists each instance of Internet Explorer running and shows you which pages, or documents, are currently loaded in that instance.

Let's create some example pages demonstrating the use of the running documents window.

The running documents window proves most useful with frame-based pages, so let's create a page with two frames inside it.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
"http://www.w3.org/TR/html4/frameset.dtd">
<html>
<frameset rows="50%,*">
  <frame name="topFrame" src="debug_topFrame.htm">
  <frame name="bottomFrame" src="debug_bottomFrame.htm">
</frameset>
</html>
```

This first page defines the frameset. Save it as `debug_frameset.htm`.

Next is the page for the top window.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>example</title>
<script language=JavaScript type="text/javascript">
function button1_onclick()
{
  var x;
  x = 1 + 1;
  alert(x)
}
</script>
</head>
<body>
<h2>Top Frame</h2>
<input type="button" value="Button" name="button1" onclick="return
button1_onclick()">
</body>
</html>
```

Save this page as `debug_topFrame.htm`.

Finally, enter the third page.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<script language=JavaScript type="text/javascript">
function button1_onclick()
{
    var x;
    x = 2 * 2;
    alert(x);
}
</script>
</head>
<body>
<h2>Bottom Frame</h2>
<input type="button" value="Button" name="button1"
    onclick="return button1_onclick()">
</body>
</html>
```

Save this page as `debug_bottomFrame.htm`.

Now load `debug_frameset.htm` into the browser. You will see two frames, each containing a button.



You can view all the pages currently loaded in the frames by first opening the debugger by choosing Script debugger ⇄ Open from the View menu. Now with the debugger open you need to view the running documents window by clicking the Running Documents icon in the toolbar, shown here, or selecting Running Documents from the View menu.

This will initially show each instance of Internet Explorer running on the machine. Click the plus sign to open a window that shows which pages are currently loaded, or running, in that instance of Internet Explorer. When pages are in framesets, as ours are, then pages contained within the window frameset page are included, indented underneath the frameset page (see Figure 10-24). You'll need to click the plus sign again to open them.

The easiest way to debug the code within the top frame is to right-click `debug_topFrame.htm` in the Running Documents window and select Break at Next Statement. Then click the button in the top frame and the debugger will open, with execution stopped at the first statement due to be executed (`onclick="return button1_onclick()"`).

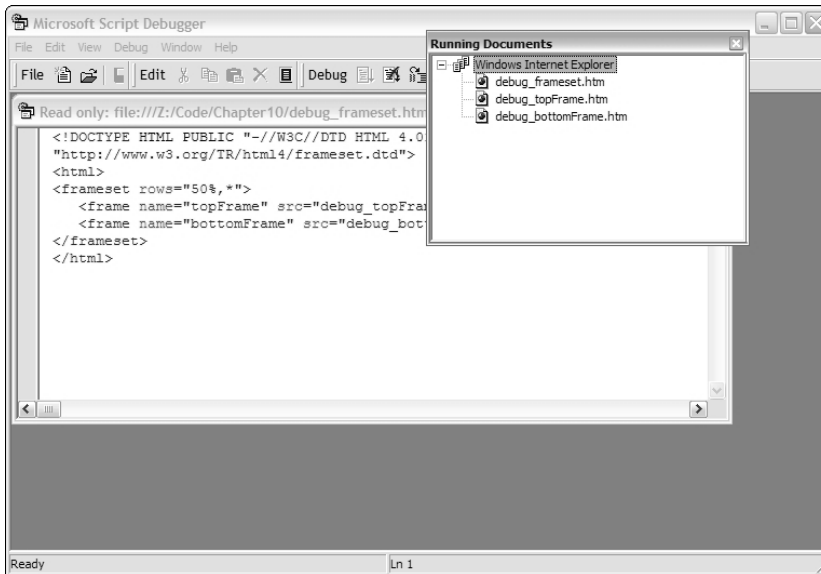


Figure 10-24

That concludes your brief tour of the Microsoft script debugger. This debugger is excellent for debugging pages so that they will work in IE. However, it won't help us spot errors caused by cross-browser incompatibility. For example, what works in Firefox might throw an error in IE, and a page that you've debugged successfully in IE could easily throw an error in Firefox. Aside from this problem, the debugger is a great help for spotting logic errors in your code.

Firefox Debugging with Venkman

Like IE, Firefox has a debugger of its own, called Venkman. It provides all the features and debugging power of the IE debugger: For example, you can step into, over, and out of your code, work with breakpoints, and access a call stack window.

You can download Venkman from

<https://addons.mozilla.org/firefox/216/>

Unfortunately, at the time of this writing, September 2006, Venkman and Firefox have bugs that make their use together difficult. After the initial install Venkman works, but later attempts to open it fail. Firefox must be restarted for Venkman to work again. Hopefully by the time you read this Venkman will be working again!

To install Venkman, open Firefox and go to the [mozilla.org](https://addons.mozilla.org/firefox/216/) URL mentioned earlier; then follow the instructions. After Venkman is installed you'll need to close down and restart Firefox. A brief guide to Venkman can be found at:

www.hacksrus.com/~ginda/venkman/

Chapter 10: Common Mistakes, Debugging, and Error Handling

When Firefox is reopened you can launch the debugger. If you now look under the Tools → JavaScript Debugger menu you will see that an extra menu option has appeared (see Figure 10-25).

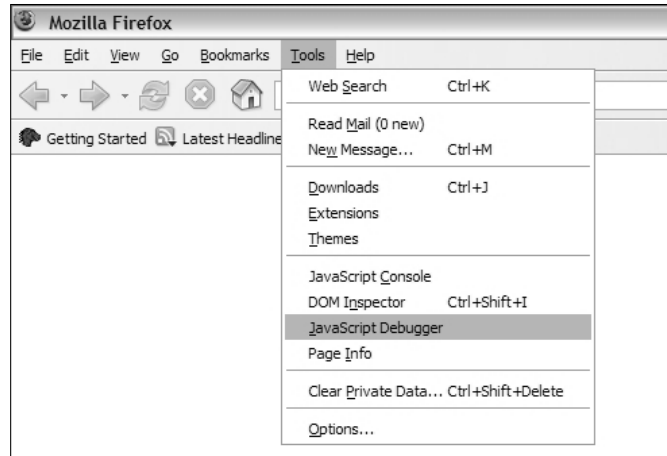


Figure 10-25

Choosing JavaScript Debugger opens Venkman, as shown in Figure 10-26.

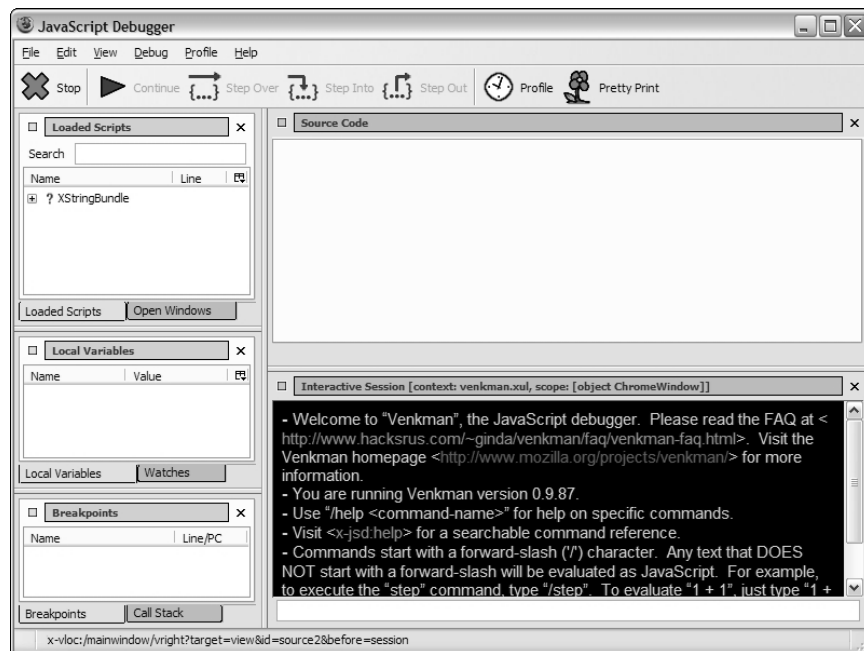


Figure 10-26

Chapter 10: Common Mistakes, Debugging, and Error Handling

The important first step is to launch Venkman. Then you can load the page or frameset that you wish to debug into Firefox. Leave Venkman open, switch to Firefox, and load the `debug_timestable2.htm` file you created earlier. As soon as it's loaded the page runs and comes to a stop on the debugger command and highlights where it has stopped, just as in the IE debugger (see Figure 10-27).

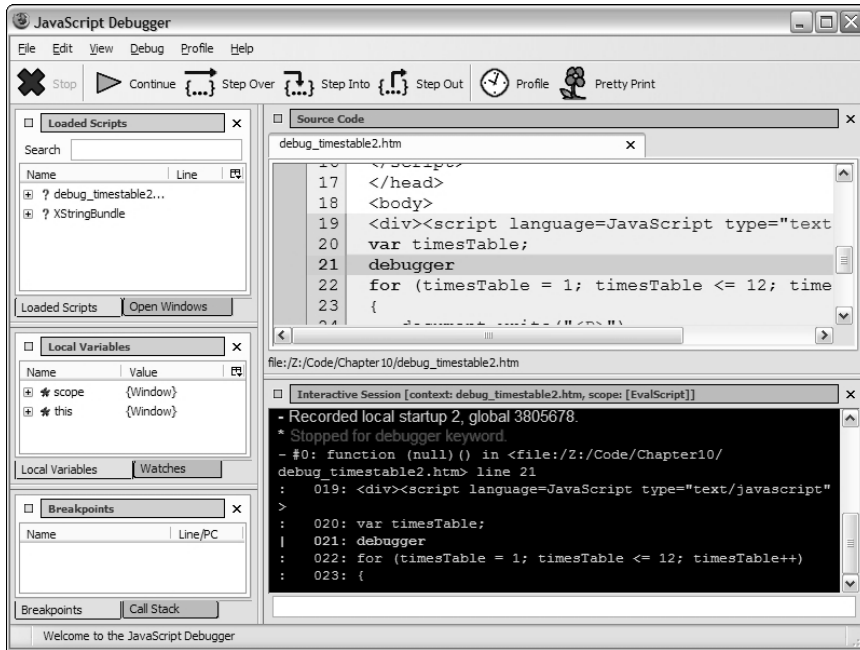


Figure 10-27

Although the layout and icons are different from those in the IE debugger, the concepts are the same, as are the sorts of things you can do. As Figure 10-27 shows, the black background of the interactive window at the bottom-right corner is basically the same as in the IE debugger's command window. In the long text box below it you can enter JavaScript code, just as with the IE debugger. If you enter `timesTable` into the text box and press Enter, it will tell you the value contained in the `timesTable` variable, which, because it hasn't yet been given, is currently void.

Click the Step Into icon on the debugger's toolbar and the code goes to the next line. The step over, step into, and step out commands work here very much as they do in the IE debugger, but there are some differences. For example, the `for` loop head is just one step with Venkman, whereas the MS debugger steps to the initialization first and then to the condition. Also, while the MS debugger runs over variable declarations, Venkman doesn't.

Keep stepping until you are in the `writeTimesTable()` function, as shown in Figure 10-28.

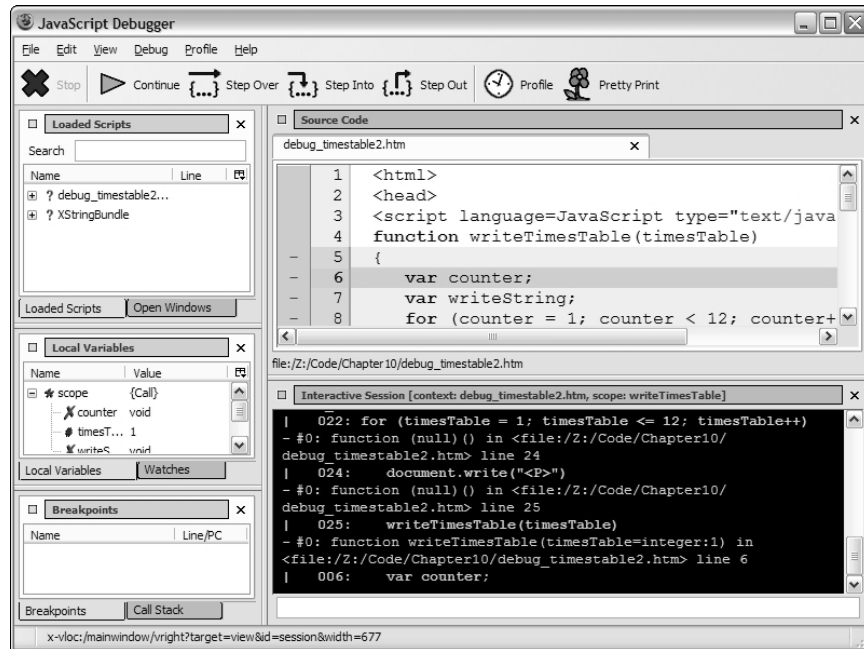


Figure 10-28

Notice that the window in the middle on the left-hand side, named Local Variables, has changed. It now shows under the heading `scope` the variables within the scope of this function. In this function these are the `counter`, `timesTable`, and `writeString` variables. The current values of each variable are also shown, `void` in the case of variables yet to be assigned a value.

In the bottom left-hand corner are the Breakpoints and Call Stack windows. Currently you see only the Breakpoints window, but by clicking the Call Stack tab under the window you can switch to that window. (You can switch back with the Breakpoints tab.) Click the Call Stack tab to show the current state of the call stack, as shown in Figure 10-29.

As Figure 10-29 shows, there are two items in the call stack: the `writeTimesTable()` function call and the code that called it.

To set a breakpoint, you click the gray area in the source code window on the line where you want to set a breakpoint (see Figure 10-30).

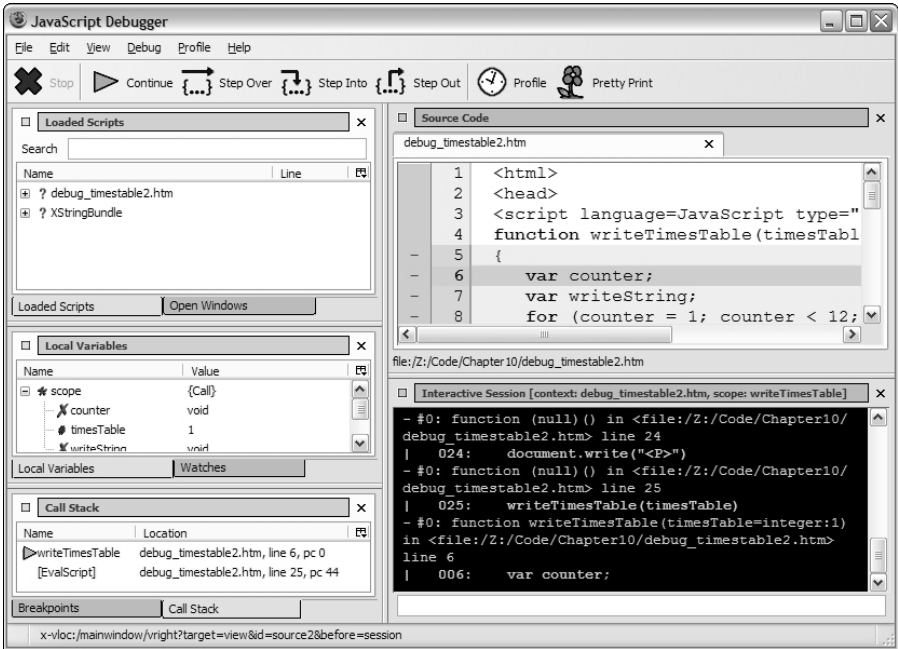


Figure 10-29

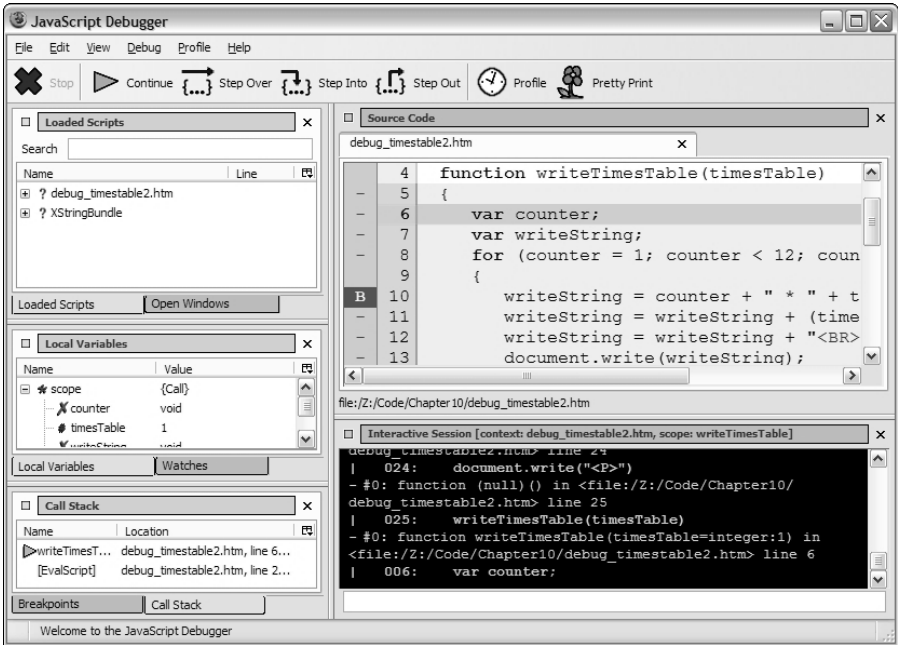


Figure 10-30

In Figure 10-30, we clicked the gray area on the `writeString = counter...` line, and the gray area has turned red and been marked with the letter **B** to show it's a breakpoint. If you click the breakpoint again it will turn orange and be marked with the letter **F** to indicate that this is a future breakpoint. Clicking a third time removes the breakpoint. Alternatively, you could right-click the breakpoint listed in the breakpoint window and clear it there — or, indeed, clear all breakpoints. Note that a future breakpoint is a breakpoint that is not set for this execution of the code but that will be set next time the page is executed, for example if you reload the page.

The final window is the Loaded Scripts window, situated in Figure 10-30 in the top left-hand corner. This shows currently loaded files. In your case there is just one document, but there can be more than one if, for example, a frameset is loaded. This is the equivalent of the IE debugger's Running Documents window.

The Venkman debugger is a little more sophisticated than the basic IE debugger, and it's worth reading the full and quite extensive documentation on the Venkman home page for details.

Error Handling

When writing your programs, you want to be informed of every error. However, when you've finally deployed the code to a web server for the whole world to access, the last thing you want the user to see is a lot of error message dialog boxes. Of course, writing bug-free code would be a good start, but keep the following points in mind:

- ❑ Occasions arise when conditions beyond your control lead to errors. A good example of this is when you are relying on something, such as a Java applet, that isn't on the user's computer and that you have no way of checking for.
- ❑ Murphy's Law states that anything that can go wrong will go wrong!

Preventing Errors

The best way to handle errors is to stop them from occurring in the first place. That seems like stating the obvious, but there are a number of things you should do if you want error-free pages.

1. Check pages thoroughly on as many different platforms and browsers as possible. Of course this is easier said than done, given the number of possible variations, and it's easier with a professional web site created by a team of people with the hardware, software, and time to check platform and browser compatibility. The alternative is for you to decide which browsers and platforms are supported and state your requirements on your first web page. Then verify that your page works with the specified combinations. Use the browser- and platform-checking code you saw earlier in the book to send unsupported users to a nice, safe, and probably boring web page with reduced functionality, or maybe just supply them with a message that their browser/platform is not supported.
2. Validate your data. If there is a way for users to enter dud data that will cause your program to fall over, then they will. If your code will fall over if a text box is empty, you must check that it has something in it. If you need a whole number, you must check that that's what the user has entered. Is the date the user just entered valid? Is the e-mail address `mind your own business` the user just entered likely to be valid? No, so you must check that it is in the format `something@something.something`.

Chapter 10: Common Mistakes, Debugging, and Error Handling

Okay, so let's say you've carefully checked your pages and there is not a syntax or logic error in sight. You've added data validation that confirms that everything the user enters is in a valid format. Things can still go wrong, and problems may arise that you can do nothing about. Here's a real-world example of something that can still go wrong.

Yours truly, Paul, created an online message board that used something called remote scripting, a Microsoft technology that enables us to transfer data from a server to a web page. It relies on a small Java applet to enable the transfer of data. Paul checked the code and everything was fine. After launching the board, it worked just fine, except that in about 5 percent of cases the Java applet initialized, but then caused an error. To cut a long story short, remote scripting worked fine, except in a small number of cases where the user was behind a particular type of firewall (a firewall is a means of stopping hackers from getting into a local computer network). Now, there is no way of determining whether a user is behind a certain type of firewall, so there is nothing that can be done in that sort of exceptional circumstance. Or is there?

In fact IE 5+ and Firefox include something called the `try...catch` statement. It's also part of the ECMAScript 3 standard. This enables you to try to run your code; if it fails, the error is caught by the `catch` clause and can be dealt with as you wish. For the message board, Paul used a `try...catch` clause to catch the Java applet's failure and redirected the user to a more basic page that still displayed messages, but without using the applet.

Let's now look at the `try...catch` statements.

The try...catch Statements

The `try...catch` statements work as a pair; you can't have one without the other.

You use the `try` statement to define a block of code that you want to try to execute.

You use the `catch` statement to define a block of code that will execute if an exception to the normal running of the code occurs in the block of code defined by the `try` statement. The term *exception* is key here: It means a circumstance that is extraordinary and unpredictable. Compare that with an *error*, which is something in the code that has been written incorrectly. If no exception occurs, the code inside the `catch` statement is never executed. The `catch` statement also enables you to get the contents of the exception message that would have been shown to the user had you not caught it first.

Let's create a simple example of a `try...catch` clause.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<body>
<script language="JavaScript" type="text/javascript">
try
{
    alert('This is code inside the try clause');
    alert('No Errors so catch code will not execute');
}
catch(exception)
{
```

```
if (exception.description == null)
{
    alert("Firefox says the error is " + exception.message)
}
else
{
    alert("Internet Explorer says the error is " + exception.description);
}
}
</script>
</body>
</html>
```

Save this as TryCatch.htm.

First you define the `try` statement; you mark out which block of code is within the `try` statement by enclosing it in curly braces.

Next comes the `catch` statement. You've included `exception` in parentheses right after the `catch` statement. This `exception` is simply a variable name. It will store an `Exception` object containing information about any exception thrown during execution of code inside the `try` code block. Although you've used `exception`, you could use any valid variable name. For example, `catch(exceptionObject)` would be fine and certainly more descriptive.

The `Exception` object contains two properties that provide information about the exception that occurred. The bad news is that while both IE and Firefox support the `Exception` object and both have two properties, the names of these properties differ.

The IE version of the `Exception` object has the `number` and `description` properties. The `number` property is a unique number for that error type. The `description` property is the error message the user would normally see.

With Firefox, the properties of the `Exception` object are `name` and `message`. The `name` property is a unique name for that type of error and the `message` property is much like the IE `description` property in that it gives a more detailed explanation of what went wrong. These properties are also part of the ECMAScript 3 standard and IE and Opera 6+ support them.

Within the curly braces after the `catch` statement is the code block that will execute if and only if an error occurs. In this case, the code within the `try` code block is fine, and so the `alert()` method inside the `catch` block won't execute.

Insert a deliberate error.

```
try
{
    alert('This is code inside the try clause');
    abler ( 'Exception will be thrown by this code');
}
catch(exception)
{
    if (exception.description == null)
```

Chapter 10: Common Mistakes, Debugging, and Error Handling

```
{
    alert("Firefox says the error is " + exception.message)
}
else
{
    alert("Internet Explorer says the error is " + exception.description);
}
}
```

Now when you load the page, the first `alert()` method, the `try` block of code, will execute fine and the alert box will be displayed to the user. However, the second `ablert()` statement will cause an error and code execution will start at the first statement in the `catch` block.

Because the IE and Firefox `Exception` objects support different properties, you need different code for each. How do you tell whether the browser is IE or Firefox?

By checking to see whether the `exception.description` property is `null`, you can tell whether the `description` property is supported, and therefore whether the browser is one of those supporting the property, such as IE. If the property is equal to `null`, it is not supported by the browser, so you need to display the `message` property of the `Exception` object instead. If the property is not `null`, it has a value and therefore does exist and can be used.

If you're using Internet Explorer, the error description displayed will be `Object expected`. If you're using Firefox, the same error is interpreted differently and reported as `ablert is not defined`.

If you change the code again, so it has a different error, you'll see something important.

```
try
{
    alert('This is code inside the try clause');
    alert('This code won't work');
}
catch(exception)
{
    if (exception.description == null)
    {
        alert("Firefox says the error is " + exception.message)
    }
    else
    {
        alert("Internet Explorer says the error is " + exception.description);
    }
}
```

If you load this code into an IE or Firefox browser, instead of the error being handled by your `catch` clause, you get the normal browser error message telling you `Expected ')'`.

The reason for this is that this code contains a syntax error: The functions and methods are valid, but you have an invalid character. The single quote in the word `won't` has ended the string parameter being passed to the `alert()` method. At that point JavaScript syntax, or language rules, specifies that a closing parenthesis should appear, which is not the case here. Before executing any code, JavaScript goes through all the code and checks for syntax errors, or code that breaches JavaScript's rules. If a syntax

error is found, the browser deals with it as usual; your `try` clause never runs and therefore cannot handle syntax errors.

Throwing Errors

The `throw` statement can be used within a `try` block of code to create your own run-time errors. Why create a statement to generate errors, when a bit of bad coding will do the same?

Throwing errors can be very useful for indicating problems such as invalid user input. Rather than using lots of `if...else` statements, you can check the validity of user input, then use `throw` to stop code execution in its tracks and cause the error-catching code in the `catch` block of code to take over. In the `catch` clause, you can determine whether the error is based on user input, in which case you can notify the user what went wrong and how to correct it. Alternatively, if it's an unexpected error, you can handle it more gracefully than with lots of JavaScript errors.

To use `throw`, you type `throw` and include the error message after it.

```
throw "This is my error message";
```

Remember that when you catch the `Exception` object in the `catch` statement, you can get hold of the error message that you have thrown. Although there's a string in this example `throw` statement, you can actually throw any type of data, including numbers and objects.

Try It Out `try...catch` and Throwing Errors

In this example you'll be creating a simple factorial calculator. The important parts of this example are the `try...catch` clause and the `throw` statements. It's a frameset page to enable us to demonstrate that things can go wrong that you can't do anything about. In this case, the page relies on a function defined within a frameset page, so if the page is loaded on its own, a problem will occur.

First let's create the page that will define the frameset and that also contains an important function.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Example</title>
<script language="JavaScript" type="text/javascript">
function calcFactorial(factorialNumber)
{
    var factorialResult = 1;
    for (; factorialNumber > 0; factorialNumber--)
    {
        factorialResult = factorialResult * factorialNumber;
    }
    return factorialResult;
}
</script>
</head>
<frameset COLS="100%,*">
    <frame name="fraCalcFactorial" src="CalcFactorial.htm">
</frameset>
</html>
```

Chapter 10: Common Mistakes, Debugging, and Error Handling

Save this page as CalcFactorialTopFrame.htm.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Example</title>
<script language="JavaScript" type="text/javascript">
function butCalculate_onclick()
{
    try
    {
        if (window.top.calcFactorial == null)
            throw "This page is not loaded within the correct frameset";
        if (document.form1.txtNum1.value == "")
            throw "!Please enter a value before you calculate its factorial";
        if (isNaN(document.form1.txtNum1.value))
            throw "!Please enter a valid number";
        if (document.form1.txtNum1.value < 0)
            throw "!Please enter a positive number";
        document.form1.txtResult.value =
            window.parent.calcFactorial(document.form1.txtNum1.value);
    }
    catch(exception)
    {
        if (typeof(exception) == "string")
        {
            if (exception.charAt(0) == "!")
            {
                alert(exception.substr(1));
                document.form1.txtNum1.focus();
                document.form1.txtNum1.select();
            }
            else
            {
                alert(exception);
            }
        }
        else
        {
            if (exception.description == null)
            {
                alert("The following error occurred " + exception.message)
            }
            else
            {
                alert("The following error occurred " + exception.description);
            }
        }
    }
}
</script>
</head>
<body>
```

```
<form name="form1">
  <input type="text" name=txtNum1 size=3> factorial is
  <input type="text" name=txtResult size=25><br>
  <input type="button" value="Calculate Factorial"
    name=butCalculate onclick="butCalculate_onclick()" >
</form>
</body>
</html>
```

Save this page as `CalcFactorial.htm`. Then load the first page, `CalcFactorialTopFrame.htm`, into your browser.

The page consists of a simple form with two text boxes and a button. Enter the number 4 into the first box and click the Calculate Factorial button. The factorial of 4, which is 24, will be calculated and put in the second text box. (See Figure 10-31.)

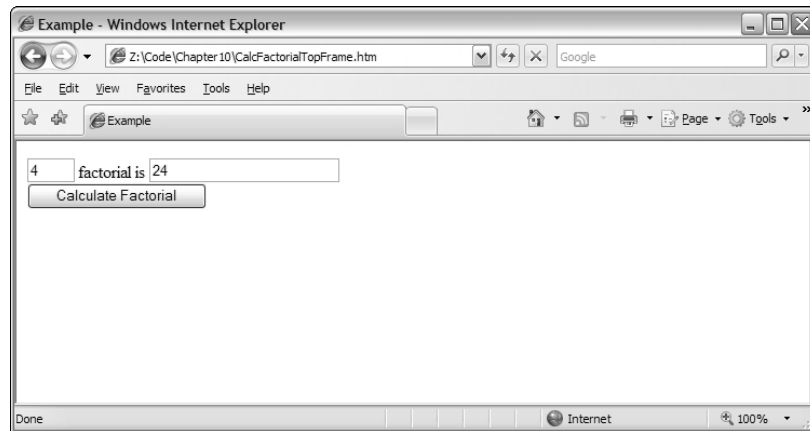


Figure 10-31

The factorial of a number is the product of all the positive integers less than or equal to that number. For example, the factorial of 4 (written $4!$) is $1 * 2 * 3 * 4 = 24$. Factorials are used in various branches of mathematics, including statistics. Here, you want only to create a function that does something complex enough to be worthy of a function, but not so complex as to distract you from the main purpose of this example: the `try...catch` and `throw` statements.

If you clear the first text box and click Calculate Factorial, you'll be told that a value needs to be entered. If you enter an invalid non-numeric value into the first text box, you'll be told to enter a valid value. If you enter a negative value, you'll be told to enter a positive value.

Also, if you try loading the page `CalcFactorial.htm` into your browser and enter a value in the text box and click Calculate Factorial, you'll be told that the page is not loaded into the correct frameset.

As you'll see, all of these error messages are created using the `try...catch` and `throw` statements.

How It Works

Because this example is all about `try...catch` and `throw`, you'll concentrate just on the `CalcFactorial.htm` page, in particular the `butCalculate_onclick()` function, which is connected to the `onclick` event handler of the form's only button.

You'll start by looking at the `try` clause and the code inside it. The code consists of four `if` statements and another line of code that puts the calculated factorial into the second text box. Each of the `if` statements is checking for a condition that, if true, would cause problems for your code.

The first `if` statement checks that the `calcFactorial()` function, in the top frameset window, actually exists. If not, it throws an error, which will be caught by the `catch` block. If the user loads the `CalcFactorial.htm` page rather than the frameset page `CalcFactorialTopFrame.htm`, then without this `throw` statement your code will fail.

```
try
{
    if (window.top.calcFactorial == null)
        throw "This page is not loaded within the correct frameset";
```

The next three `if` statements check the validity of the data entered into the text box by the user. First you check that something has actually been entered in the box, then that what has been entered is a number, and then finally you check that the value is not negative. Again if any of the `if` conditions is true, you throw an error, which will be caught by the `catch` block. Each of the error messages you define starts with an exclamation mark, the purpose of which is to mark the error as a user input error, rather than an error such as not being in a frameset.

```
    if (document.form1.txtNum1.value == "")
        throw "!Please enter a value before you calculate its factorial";
    if (isNaN(document.form1.txtNum1.value))
        throw "!Please enter a valid number";
    if (document.form1.txtNum1.value < 0)
        throw "!Please enter a positive number";
```

If everything is fine, the `calcFactorial()` function will be executed and the results text box will be filled with the factorial of the number entered by the user.

```
        document.form1.txtResult.value =
            window.parent.calcFactorial(document.form1.txtNum1.value);
    }
```

Finally, turn your attention to the `catch` part of the `try...catch` statement. First, any message thrown by the `try` code will be caught by the exception variable.

```
catch(exception)
{
```

The type of data contained in `exception` will depend on how the error was thrown. If it was thrown by the browser and not by your code, `exception` will be an object, the `Exception` object. If it's thrown by your code, then in this instance you've thrown only primitive strings. So the first thing you need to do is decide what type of data `exception` contains. If it's a string, you know it was thrown by your code and

can deal with it accordingly. If it's an object, and given that you know none of your code throws objects, you assume it must be the browser that has generated this exception and that `exception` is an `Exception` object.

```
if (typeof(exception) == "string")
{
```

If it was code that generated the exception using a `throw` (and so `exception` is a string), you now need to determine whether the error is a user input error, such as the text box not containing a value to calculate, or whether it was another type of error, such as the page not being loaded in your frameset. All the user input exception messages had an exclamation mark at the beginning, so you use an `if` statement to check the first character. If it is a `!`, you notify the user of the error and then return focus to your control. If it's not, you just display an error message.

```
    if (exception.charAt(0) == "!")
    {
        alert(exception.substr(1));
        document.form1.txtNum1.focus();
        document.form1.txtNum1.select();
    }
    else
    {
        alert(exception);
    }
}
```

If `exception` was not a string, you know you have an exception object and need to display either the `message` property if it's Firefox or the `description` property if it's IE. You use `if e.description == null` check to see which property is supported.

```
    else
    {
        if (exception.description == null)
        {
            alert("The following error occurred " + exception.message)
        }
        else
        {
            alert("The following error occurred " + exception.description);
        }
    }
}
```

Nested try...catch Statements

So far you've been using just one `try...catch` statement, but it's possible to include a `try...catch` statement inside another `try` statement. Indeed, you can go further and have a `try...catch` inside the `try` statement of this inner `try...catch`, or even another inside that, the limit being what it's actually sensible to do.

So why would you use nested `try...catch` statements? Well, you can deal with certain errors inside the inner `try...catch` statement. If, however, you're dealing with a more serious error, the inner `catch` clause could pass that error to the outer `catch` clause by throwing the error to it.

Chapter 10: Common Mistakes, Debugging, and Error Handling

Here's an example.

```
try
{
    try
    {
        ablurt ("This code has an error");
    }
    catch(exception)
    {
        var eMessage
        if (exception.description == null)
        {
            eMessage = exception.name;
        }
        else
        {
            eMessage = exception.description;
        }
        if (eMessage == "Object expected" || eMessage == "ReferenceError")
        {
            alert("Inner try...catch can deal with this error");
        }
        else
        {
            throw exception;
        }
    }
}
catch(exception)
{
    alert("Error the inner try...catch could not handle occurred");
}
```

In this code you have two `try...catch` pairs, one nested inside the other.

The inner `try` statement contains a line of code that contains an error. The `catch` statement of the inner `try...catch` checks the value of the error message caused by this error. If the exception message is either `Object expected` or `ReferenceError`, the inner `try...catch` deals with it by way of an alert box.

In fact, both the exception messages checked for are the same thing, but reported differently by IE and Firefox. Note that these examples use the Firefox `Exception` object's `name` property rather than the `message` for comparison, because `name` is a much shorter one-word description of the exception than `message`, which is a sentence describing the exception.

However, if the error caught by the inner `catch` statement is any other type of error, it is thrown up in the air again for the `catch` statement of the outer `try...catch` to deal with.

Let's change the `butCalculate_onclick()` function from the previous example, `CalcFactorial.htm`, so that it has both an inner and an outer `try...catch`.

```
function butCalculate_onclick()
{
    try
    {
        try
        {
            if (window.top.calcFactorial == null)
                throw ("This page is not loaded within the correct frameset");
            if (document.form1.txtNum1.value == "")
                throw ("Please enter a value before you calculate its factorial");
            if (isNaN(document.form1.txtNum1.value))
                throw ("Please enter a valid number");
            if (document.form1.txtNum1.value < 0)
                throw ("Please enter a positive number");
            document.form1.txtResult.value =
                window.parent.calcFactorial(document.form1.txtNum1.value);
        }
        catch(exception)
        {
            if (typeof(exception) == "string" && exception.charAt(0) == "!")
            {
                alert(exception.substr(1));
                document.form1.txtNum1.focus();
                document.form1.txtNum1.select();
            }
            else
            {
                throw exception;
            }
        }
    }
    catch(exception)
    {
        switch (exception)
        {
            case "This page is not loaded within the correct frameset":
                alert(exception);
                break;
            default :
                alert("The following critical error has occurred \n" + exception);
        }
    }
}
```

The inner `try...catch` deals with user input errors. However, if the error is not a user input error thrown by us, it is thrown for the outer `catch` statement to deal with. The outer `catch` statement has a `switch` statement that checks the value of the error message thrown. If it's the error message thrown by us because the `calcFactorialTopFrame.htm` is not loaded, the `switch` statement deals with it in the first `case` statement. Any other error is dealt with in the `default` statement. However, there may well be occasions when there are lots of different errors you want to deal with in `case` statements.

finally Clauses

The `try...catch` statement has a `finally` clause that defines a block of script that will execute whether or not an exception was thrown. The `finally` clause can't appear on its own; it must be after a `try` block, which the following code demonstrates.

```
try
{
    ablurt("An exception will occur");
}
catch(exception)
{
    alert("Exception occurred");
}
finally
{
    alert("Whatever happens this line will execute");
}
```

The `finally` part is a good place to put any cleanup code that needs to be executed regardless of any errors that occurred previously.

Summary

In this chapter you looked at the less exciting part of coding, namely bugs. In an ideal world you'd get things right the first time, every time, but in reality any code more than a few lines long is likely to suffer from bugs.

- ❑ You first looked at some of the more common errors, those made not just by JavaScript beginners, but also by experts with lots of experience.
- ❑ You installed the script debugger, which works with Internet Explorer. Without the script debugger any errors throw up messages, but nothing else. With the script debugger you get to see exactly where the error might have been and to examine the current state of variables when the error occurred. Also, you can use the script debugger to analyze code as it's being run, which enables you to see its flow step by step, and to check variables and conditions.
- ❑ You looked at Venkman, the script debugger for Firefox. You saw that it works much like the IE debugger and in some ways is superior to it. Although these debuggers have different interfaces, their principles are the same.
- ❑ Some errors are not necessarily bugs in your code, but in fact exceptions to the normal circumstances that cause your code to fail. (For example, a Java applet might fail because a user is behind a firewall.) You saw that the `try...catch` statements are good for dealing with this sort of error, and that you can use the `catch` clause with the `throw` statement to deal with likely errors, such as those caused by user input. Finally, you saw that if you want a block of code to execute regardless of any error, you can use the `finally` clause.

In the next chapter you'll be looking at a way to store information on the user's computer using something called a *cookie*. Although they may not be powerful enough to hold a user's life history, they are certainly powerful enough for you to keep track of a user's visits to your web site and of what pages he views when he visits. With that information you can provide a more customized experience for the user.

Exercise Questions

Suggested solutions to these questions can be found in Appendix A.

Question 1

The example `debug_timestable2.htm` has a deliberate bug. For each times table it creates only multipliers with values from 1 to 11.

Use the script debugger to work out why this is happening, and then correct the bug.

Question 2

The following code contains a number of common errors. See if you can spot them:

```
<html>
<head>
</head>
<body>
<script language=JavaScript>
function checkForm(theForm)
{
    var formValid = true;
    var elementCount = 0;
    while(elementCount =< theForm.length)
    {
        if (theForm.elements[elementCount].type == "text")
        {
            if (theForm.elements[elementCount].value() = "")
            alert("Please complete all form elements")
            theForm.elements[elementCount].focus;
            formValid = false;
            break;
        }
    }
    return formValid;
}
</script>
<form name=form1 onsubmit="return checkForm(document.form1)">
    <input type="text" ID=text1 name=text1>
    <br>
    CheckBox 1<input type="checkbox" ID=checkbox1 name=checkbox1>
    <br>
    CheckBox 2<input type="checkbox" ID=checkbox2 name=checkbox2>
    <br>
    <input type="text" ID=text2 name=text2>
    <p>
    <input type="submit" value="Submit" ID=submit1 name=submit1>
    </p>
</form>
</body>
</html>
```


11

Storing Information: Cookies

Our goal as web site programmers should be to make the web site experience as easy and pleasant for the user as possible. Clearly, well-designed pages with easily navigable layouts are central to this, but they're not the whole story. You can go one step further by learning about your users and using information gained about them to personalize the web site.

For example, imagine a user, whose name you asked on the first visit, returns to your web site. You could welcome her back to the web site by greeting her by name. Another good example is given by a web site, such as Amazon's, that incorporates the one-click purchasing system. By already knowing the user's purchasing details, such as credit-card number and delivery address, you can allow the user to go from viewing a book to buying it in just one click, making the likelihood of the user purchasing it that much greater. Also, based on information, such as the previous purchases and browsing patterns of the user, it's possible to make book suggestions.

Such personalization requires that information about users be stored somewhere in between their visits to the web site. Previous chapters have mentioned that accessing the user's local file system from a web application is pretty much off limits because of security restrictions included in browsers. However, you, as a web site developer, can store small amounts of information in a special place on the user's local disk, using what is called a *cookie*. There may be a logical reason why they are named cookies, but it also provides authors with the opportunity to make a lot of second-rate, food-related jokes!

Baking Your First Cookie

The key to cookies is the `document` object's `cookie` property. Using this property, you can create and retrieve cookie data from within your JavaScript code.

You can set a cookie by setting `document.cookie` to a *cookie string*. You'll be looking in detail at how this cookie string is made up later in the chapter, but let's first create a simple example of a cookie and see where the information is stored on the user's computer.

A Fresh-Baked Cookie

The following code will set a cookie with the `UserName` set as `Paul`, and an expiration date of 28 December, 2020.

```
<html>
<head>
<script language="JavaScript" type="text/javascript">
    document.cookie = "UserName=Paul;expires=Tue, 28 Dec 2020 00:00:00;";
</script>
</head>
<body>
<p>This page just created a cookie</p>
</body>
</html>
```

Save the page as `FreshBakedCookie.htm`. You'll see how the code works as you learn the parts of a cookie string, but first let's see what happens when a cookie is created.

How you view cookies without using code varies with the browser you are using. You'll see how to do it first in IE, and then in Firefox.

Viewing Cookies in IE

In this section you'll see how to look at the cookies that are already stored by IE on your computer. You'll then load the cookie-creating page you just created with the preceding code to see what effect this has.

First you need to open IE. The examples in this chapter use IE 7, so if you're using IE 6 you may find the screenshots and menus in slightly different places.

Before you view the cookies, you'll first clear the temporary Internet file folder for the browser, because this will make it easier to view the cookies that your browser has stored. In IE, select Internet Options from the Tools menu, which is shown in Figure 11-1.

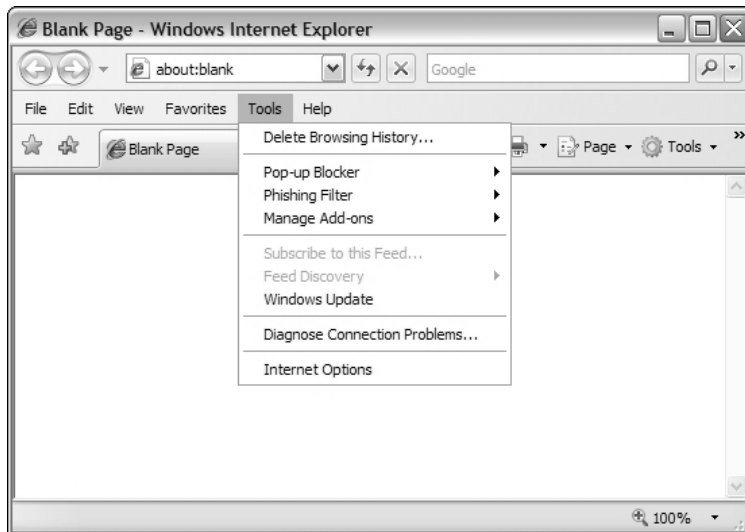


Figure 11-1

Having selected this option, you'll be presented with the Internet Options dialog box shown in Figure 11-2.



Figure 11-2

Click the Delete button under Browsing History. Another dialog box appears, as shown in Figure 11-3.



Figure 11-3

Chapter 11: Storing Information: Cookies

Click the Delete files button next to Temporary Internet Files, and select Yes when it asks if you're sure.

You now have a nice clean cache, which makes it easy to see when you create a cookie. You can now close the dialog box and return to the main Internet Options dialog box.

Let's have a look at the cookies you have currently residing on your machine. From the Internet Options dialog box, click the Settings button next to the Delete button grouped under Browsing History. You should see the dialog box shown in Figure 11-4.



Figure 11-4

Now click the View Files button, and a list of all the temporary pages and cookie files on your computer will be displayed. If you followed the previous instructions and deleted all temporary Internet files, there should be nothing listed except any cookies from web sites you've visited, as shown in Figure 11-5.

The actual cookies, their names, and their values, may look slightly different depending on your computer's operating system.

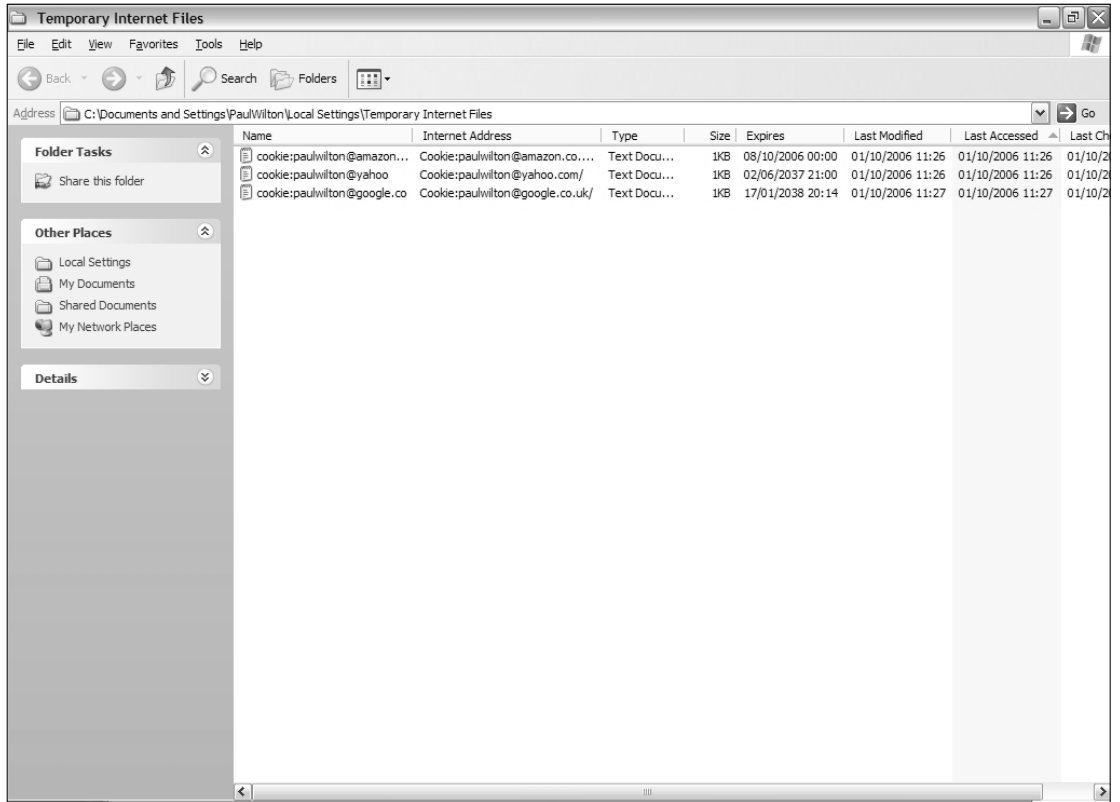


Figure 11-5

You can examine the contents of the cookies by double-clicking them. Note that you may get a warning about the potential security risk of opening a text file, although you are fairly safe with cookies because they are simply text files. In Figure 11-6 you can see the contents of the cookie file named `google` set by the search engine Google.

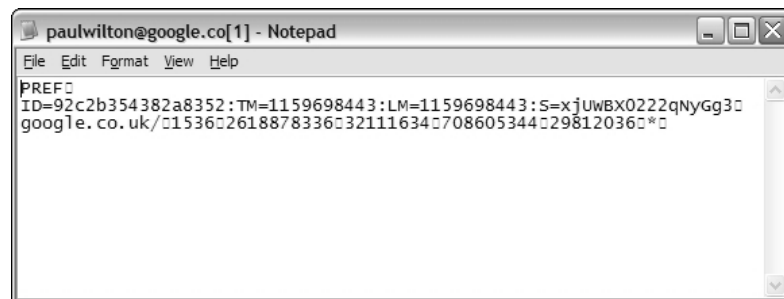


Figure 11-6

Chapter 11: Storing Information: Cookies

As you can see, a cookie is just a plain old text file. Each web site, or *domain name*, has its own text file where all the cookies for that web site are stored. In this case there's just one cookie currently stored for `google.co.uk`. Domains like `amazon.com` will almost certainly have many cookies set.

In Figure 11-6, you can see the cookie's details. Here, the name of the cookie is `PREF`; its value is a series of characters, which although indecipherable to you make sense to the Google web site. It was set by the domain `google.co.uk`, and it relates to the root directory `/`. The contents probably look like a mess of characters, but don't worry: When you learn how to program cookies, you'll see that you don't need to worry about setting the details in this format.

After you have finished, close the cookie and click OK on the dialog boxes to return to the browser.

Now let's load the `FreshBakedCookie.htm` page into your IE browser. This will set a cookie. Let's see how it has changed things by returning to the Internet Options dialog box (by choosing Internet Options from the Tools menu). Click the Settings button, and then click View Files. Your computer now shows something like the information in Figure 11-7.

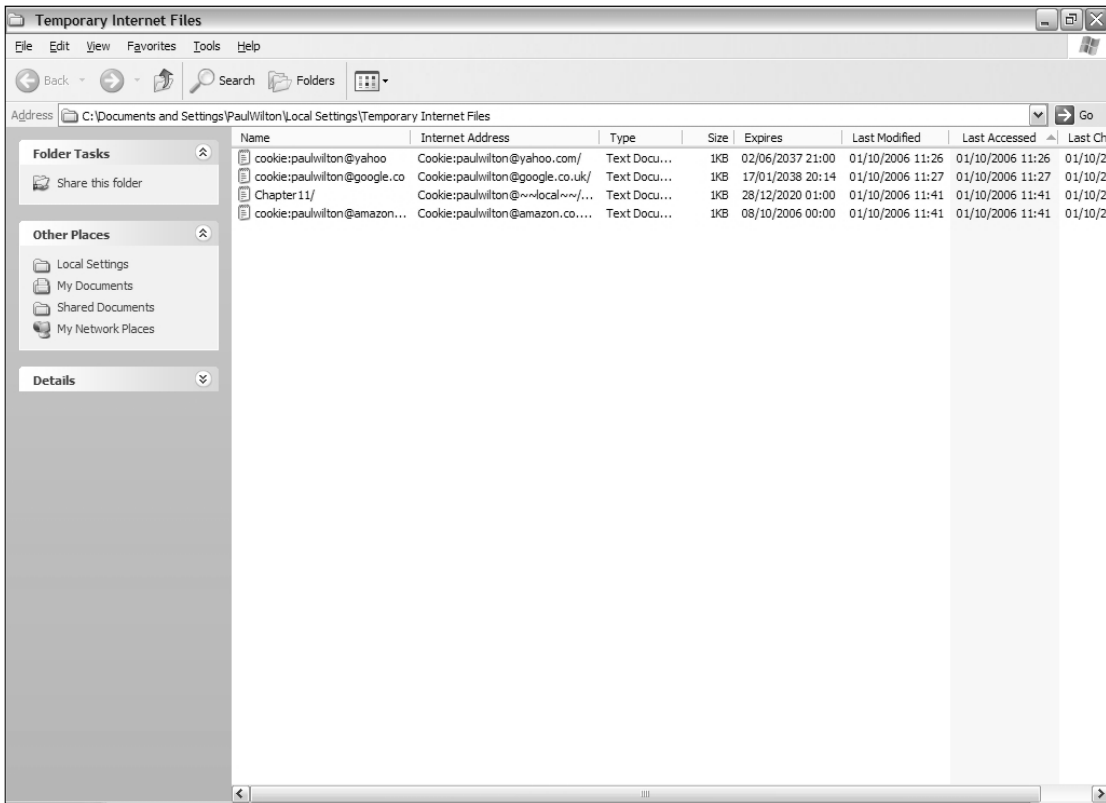


Figure 11-7

Because you are creating a cookie from a web page that is stored on the local hard drive rather than a server, its domain name has been set to the name of the directory the web page is stored in. Obviously, this is a little artificial. In reality, people will be loading your web pages from your web site on the Internet and not off your local hard drive. The Internet address is based on the directory the `FreshBakedCookie.htm` file was in. You can also see that it expires on December 28, 2020, as you specified when you created the cookie. Double-click the cookie to view its contents, which look like those in Figure 11-8.

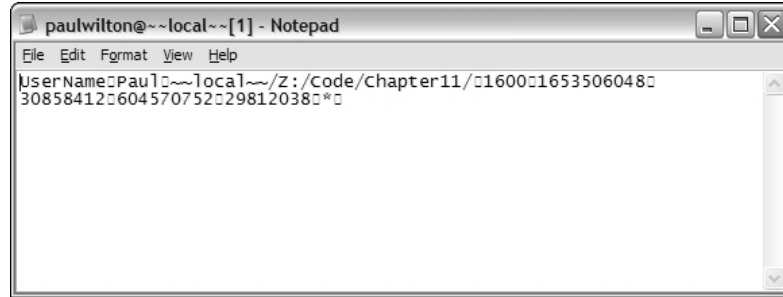


Figure 11-8

You can see the name you gave to the cookie at the left, `UserName`, its value, `Paul`, and also the directory it's applicable to. The expiration date is there as well; it's just not in an easily recognizable form. Note that you may sometimes need to close the browser and reopen it before you see the cookie file.

Viewing Cookies in Firefox

There is no sharing of cookies between browsers, so the cookies stored when you visited web sites using an IE browser won't be available to Firefox and vice versa.

FF keeps its cookies in a totally different place from IE, and the contents are viewed by a different means. To view cookies in Firefox 1.5, choose Options from the Tools menu. Then select the Privacy tab and finally select the Cookies tab and you should see the dialog box shown in Figure 11-9.

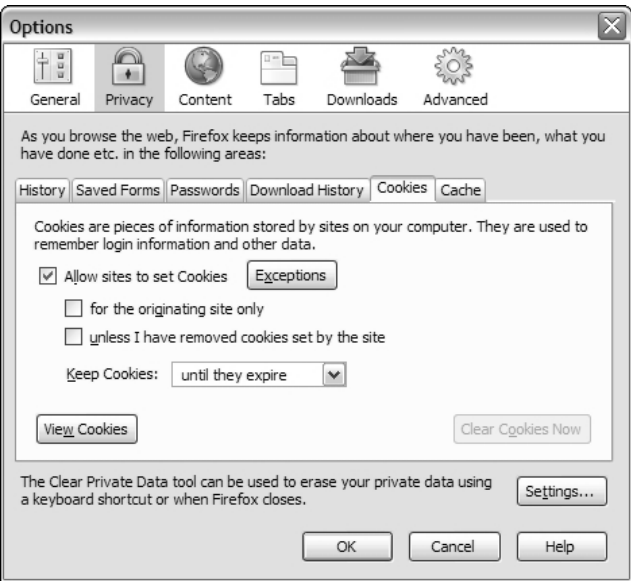


Figure 11-9

Click View Cookies and his will open another dialog box listing all the cookies currently stored on your PC. It should look something like what is shown in Figure 11-10.



Figure 11-10

Click OK to get back to the browser, and load `FreshBakedCookie.htm`. Then repeat the process you followed previously to get to the Cookie Manager, and you should find that the `UserName` cookie has been added to the box. Because it's loaded from a file on your PC and not the Internet, the cookie has a blank web address. The expanded cookie details are shown in Figure 11-11.



Figure 11-11

Note that buttons are provided at the bottom of the Cookie Manager to remove the cookie selected or all of the cookies that are stored.

Now that you've seen how to view cookies manually, let's look at how you create them and read them using code. You'll start by looking at each of the parts making up a cookie string.

Note that cookie management has changed in Firefox 2.0. You'll still find it under the Tools ⇄ Options menu but now there's no Cookie tab. Instead there is a Show Cookies button. Clicking the Show Cookies button will open a new dialog box that lets you view and delete cookies.

The Cookie String

When you are creating a cookie there are six parts you can set: name, value, expires, path, domain, and *secure*, although the latter four of these are optional. You'll now look at each of these in turn.

name and value

The first part of the cookie string consists of the name and value of the cookie. The name is used so that you can reference the cookie later, and the value is the information part of the cookie.

This name/value part of the cookie string is compulsory; it sort of defeats the point of the cookie if you don't store a name or value, because storing information is what cookies are all about. You should make sure that this part comes first in the cookie string.

The value for the cookie is a primitive string, although of course the string can hold number characters if it is numerical data that you want to store. If you are storing text, certain characters, such as semicolons, cannot be used inside the value, unless you use a special encoding, which you'll see later. In the case of semicolons, this is because they are used to separate the different parts of the cookie within the cookie string.

Chapter 11: Storing Information: Cookies

In the following line of code, you set a cookie with the name `UserName` and the value `Paul`.

```
document.cookie = "UserName=Paul;";
```

This cookie has a very limited *lifespan*, which is the length of time the information will continue to exist. If you don't set an expiration date, a cookie will expire when the user closes the browser. The next time the user opens the browser the cookie will be gone. This is fine if you just want to store information for the life of a user *session*, which is a single visit by the user to your web site. However, if you want to ensure that your cookie is available for longer, you must set its expiration date, which you'll look at next.

expires

If you want a cookie to exist for longer than just a single user session, you need to set an expiration date using the second part of the cookie string, `expires`, as follows:

```
document.cookie = "UserName=Paul;expires=Tue, 28 Dec 2020 00:00:00 GMT; ";
```

The cookie set by the previous line of code will remain available for future use right up until December 28, 2020. Note that the format of the expiration date is very important, especially for IE browsers. It should be the same format the cookie is given by the `toGMTString()` method. This method is similar to the `toUTCString()` method that you saw in Chapter 9.

In practice, you'll probably use the `Date` object to get the current date, and then set a cookie to expire three or six months after this date. Otherwise, you're going to need to rewrite your pages on December 28, 2020.

For example, you could write the following:

```
var expireDate = new Date();
expireDate.setMonth(expireDate.getMonth() + 6);
document.cookie = "UserName=Paul;expires=" + expireDate.toGMTString() + ";;";
```

This will create a new cookie called `UserName` with the value of `Paul`, which will expire six months from the current date. Note that other factors can cause a cookie to expire before its expiration date, such as the user deleting the cookie or the upper cookie limit being reached.

path

You'll find that 99 percent of the time you will only need to set the `name`, `value`, and `expires` parts of a cookie. However, at times the other three parts, such as the `path` part that you are looking at in this section, need to be set. The final two parts, `domain` and `secure`, are for advanced use beyond the scope of a beginners' book, but you'll look at them briefly just for completeness.

You're probably used to the idea of there being directories on your hard drive. Rather than storing everything on your computer in one place on the hard drive, you divide it into these directories. For example, you might keep your word-processing files in `My Documents`, your image files in `My Images`, and so on. You probably also subdivide your directories, so under `My Images` you might have subdirectories called `My Family` and `My Holiday`.

Well, web servers use the same principle. Rather than putting the whole web site into one web directory, it's common and indeed sensible to divide it into various different directories. For example, if you visit

the Wrox web site at `www.wrox.com`, and then click one of the book categories, you'll find that the path to the page navigated to is now `www.wrox.com/Books/`.

This is all very interesting, but why is it relevant to cookies?

The problem is that cookies are specific not only to a particular web domain, such as `www.wrox.com`, but also to a particular path on that domain. For example, if a page in `www.wrox.com/Books/` sets a cookie, then only pages in that directory or its subdirectories will be able to read and change the cookie. If a page in `www.wrox.com/academic/` tried to read the cookie, it would fail. Why are cookies restricted like this?

Take the common example of free web space. A lot of companies on the Web enable you to sign up for free web space. Usually everyone who signs up for this web space has a site at the same domain. For example, Bob's web site might be at `www.freespace.com/members/bob/`. Belinda might have hers at `www.freespace.com/members/belinda`. If cookies could be retrieved and changed regardless of the path, then any cookies set on Bob's web site could be viewed by Belinda and vice versa. This is clearly something neither of them would be happy about. Not only is there a security problem, but if, unknown to each other, they both have a cookie named `MyHotCookie`, there would be problems with each of them setting and retrieving the same cookie. When you think how many users a free web space provider often has, you can see that there is potential for chaos.

Okay, so now you know that cookies are specific to a certain path, but what if you want to view your cookies from two different paths on your server? Say for example you have an online store at `www.mywebsite.com/mystore/` but you subdivide the store into subdirectories, such as `/Books` and `/Games`. Now let's imagine that your checkout is in the directory `www.mywebsite.com/mystore/Checkout`. Any cookies set in the `/Books` and `/Games` directories won't be visible to each other or pages in the `/Checkout` directory. To get around this you can either set cookies only in the `/mystore` directory, since these can be read by that directory and any of its subdirectories, or you can use the path part of the cookie string to specify that the path of the cookie is `/mystore` even if it's being set in the `/Games` or `/Books` or `/Checkout` subdirectories.

For example, you could do this like so:

```
document.cookie = "UserName=Paul;expires=Tue, 28 Dec 2020 00:00:00" +  
";path=/mystore;"
```

Now, even if the cookie is set by a page in the directory `/Books`, it will still be accessible to files in the `/mystore` directory and its subdirectories, such as `/Checkout` and `/Games`.

If you want to specify that the cookie is available to all subdirectories of the domain it is set in, you can specify a path of the root directory using the `/` character.

```
document.cookie = "UserName=Paul;expires=Tue, 28 Dec 2020 00:00:00;path=/;"
```

Now, the cookie will be available to all directories on the domain it is set from. If the web site is just one of many at that domain, it's best not to do this because everyone else will also have access to your cookie information.

It's important to note that although Windows computers don't have case-sensitive directory names, many other operating systems do. For example, if your web site is on a Unix- or Linux-based server, the path property will be case-sensitive.

Chapter 11: Storing Information: Cookies

domain

The fourth part of the cookie string is the domain. An example of a domain is `wrox.com` or `pawilton.com`. Like the path part of the cookie string, the domain part is optional and it's unlikely that you'll find yourself using it very often.

By default, cookies are available only to pages in the domain they were set in. For example, if you have your first web site running on a server with the domain `MyPersonalWebSite.MyDomain.Com` and you have a second web site running under `MyBusinessWebSite.MyDomain.Com`, a cookie set in one web site will not be available to pages accessed under the other domain name, and vice versa. Most of the time this is exactly what you want, but if it is not, you can use the domain part of the cookie string to specify that a cookie is available to all subdomains of the specified domain. For example, the following sets a cookie that can be shared across both subdomains:

```
document.cookie = "UserName=Paul;expires=Tue, 28 Dec 2020 00:00:00;path="/" +
";domain=MyDomain.Com;"
```

Note that the domain must be the same: You can't share `www.SomeoneElsesDomain.com` with `www.MyDomain.com`.

secure

The final part of the cookie string is the secure part. This is simply a Boolean value; if it's set to `true` the cookie will be sent only to a web sever that tries to retrieve it using a secure channel. The default value, which is `false`, means the cookie will always be sent, regardless of the security. This is only applicable where you have set up a server with SSL (Secure Sockets Layer).

Creating a Cookie

To make life easier for yourself, you'll write a function that enables you to create a new cookie and set certain of its attributes with more ease. You'll look at the code first and create an example using it shortly.

```
function setCookie(cookieName, cookieValue, cookiePath, cookieExpires)
{
    cookieValue = escape(cookieValue);
    if (cookieExpires == "")
    {
        var nowDate = new Date();
        nowDate.setMonth(nowDate.getMonth() + 6);
        cookieExpires = nowDate.toGMTString();
    }
    if (cookiePath != "")
    {
        cookiePath = ";Path=" + cookiePath;
    }
    document.cookie = cookieName + "=" + cookieValue +
        ";expires=" + cookieExpires + cookiePath;
}
```

The `secure` and `domain` parts of the cookie string are unlikely to be needed, so you allow just the `name`, `value`, `expires`, and `path` parts of a cookie to be set by the function. If you don't want to set a path or expiration date, you just pass empty strings for those parameters. If no path is specified, the current directory and its subdirectories will be the path. If no expiration date is set, you just assume a date six months from now.

The first line of the function introduces the `escape()` function, which you've not seen before.

```
cookieValue = escape(cookieValue);
```

When we talked about setting the value of a cookie, we mentioned that certain characters cannot be used directly, such as a semicolon. (This also applies to the name of the cookie.) To get around this problem, you can use the built-in `escape()` and `unescape()` functions. The `escape()` function converts characters that are not text or numbers into the hexadecimal equivalent of their character in the Latin-1 character set, preceded by a `%` character.

For example, a space has the hexadecimal value `20`, and the semicolon the value `3B`. So the following code produces the output shown in Figure 11-12:

```
alert(escape("2001 a space odyssey;"));
```

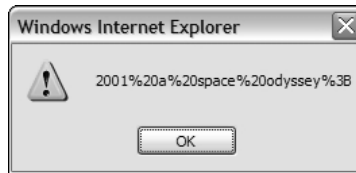


Figure 11-12

You can see that each space has been converted to `%20`, the `%` indicating that it represents an escape or special character rather than an actual character, and that `20` is the ASCII value of the actual character. The semicolon has been converted to `%3B`, as you'd expect.

As you'll see later, when retrieving cookie values you can use the `unescape()` function to convert from the encoded version to plain text.

Back to your function; next you have an `if` statement.

```
if (cookieExpires == "")
{
    var nowDate = new Date();
    nowDate.setMonth(nowDate.getMonth() + 6);
    cookieExpires = nowDate.toGMTString();
}
```

This deals with the situation in which an empty string `""` has been passed for the `cookieExpires` parameter of the function. Because most of the time you want a cookie to last longer than the session it's created in, you set a default value for `expires` that is six months after the current date.

Chapter 11: Storing Information: Cookies

Next, if a value other than an empty string ("") has been passed to the function for the `cookiePath` parameter, you need to add that value when you create the cookie. You simply put "path=" in front of any value that has been passed in the `cookiePath` parameter.

```
if (cookiePath != "")
{
    cookiePath = ";Path=" + cookiePath;
}
```

Finally, on the last line you actually create the cookie, putting together the `cookieName`, `cookieValue`, `cookieExpires`, and `cookiePath` parts of the string.

```
document.cookie = cookieName + "=" + cookieValue +
    ";expires=" + cookieExpires + cookiePath;
```

You'll be using the `setCookie()` function whenever you want to create a new cookie because it makes setting a cookie slightly easier than having to remember all the parts you want to set. More importantly, it can be used to set the expiration date to a date six months ahead of the current date.

For example, to use the function and set a cookie with default values for `expires` and `path`, you just type the following:

```
setCookie("cookieName", "cookieValue", "", "")
```

Try It Out Using `setCookie()`

You'll now put all this together in a simple example in which you use your `setCookie()` function to set three cookies named `Name`, `Age`, and `FirstVisit`. You then display what is in the `document.cookie` property to see how it has been affected.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<script language="JavaScript" type="text/JavaScript">
function setCookie (cookieName, cookieValue, cookiePath, cookieExpires)
{
    cookieValue = escape(cookieValue);
    if (cookieExpires == "")
    {
        var nowDate = new Date();
        nowDate.setMonth(nowDate.getMonth() + 6);
        cookieExpires = nowDate.toGMTString();
    }
    if (cookiePath != "")
    {
        cookiePath = ";Path=" + cookiePath;
    }
    document.cookie = cookieName + "=" + cookieValue +
        ";expires=" + cookieExpires + cookiePath;
}
setCookie("Name", "Bob", "", "");
```



```
setCookie("Age", "101", "", "");  
setCookie("FirstVisit", "10 May 2007", "", "");  
alert(document.cookie);  
</script>  
</head>  
<body>  
</body>  
</html>
```

Save the example as `CreateCookie.htm` and load it into a web browser.

You'll see the alert box shown in Figure 11-13. Note that all three cookies are displayed as name/value pairs separated from the others by semicolons, and also that the expiration date is not displayed. If you had set the path parameter, this also would not have been displayed. The `UserName` cookie from a previous example is also displayed.



Figure 11-13

How It Works

You've already seen how the `setCookie()` function works, so let's look at the three lines that use the function to create three new cookies.

```
setCookie("Name", "Bob", "", "");  
setCookie("Age", "101", "", "");  
setCookie("FirstVisit", "10 May 2007", "", "");
```

It is all fairly simple. The first parameter is the name that you'll give the cookie. (You'll see shortly how you can retrieve a value of a cookie based on the name you gave it.) It's important that the names you use be only alphanumeric characters, with no spaces, punctuation, or special characters. Although you can use cookie names with these characters, doing so is more complex and best avoided. Next you have the value you want to give the cookie. The third parameter is the path, and the fourth parameter is the date you want the cookie to expire on.

For example, take the first line where you use the `setCookie()` function. Here you are setting a cookie that will be named `Name` and have the value `Bob`. You don't want to set the `path` or `expires` parts, so you just pass an empty string (`" "`). Note that you must pass the empty string. You can't pass nothing at all.

The remaining two lines in the previous code snippet set the cookies named `Age` and `FirstVisit` and set their values to `101` and `10 May 2007`, respectively.

If you did want to set the path and the expiration date, how might you change your code?

Chapter 11: Storing Information: Cookies

Well, imagine that you want the path to be `/MyStore` and the expiration date to be one year in the future. Then you can use the `setCookie()` function in the following way:

```
var expireDate = new Date();
expireDate.setMonth(expireDate.getMonth() + 12);
setCookie("Name", "Bob", "/MyStore", expireDate.toGMTString());
```

First you create a new `Date` object, and by passing no parameter to its constructor, you let it initialize itself to the current date. In the next line, you add 12 months to that date. When setting the cookie using `setCookie()` you pass `/MyStore` as the path and `expireDate.toGMTString()` as the expires parameter.

What about the situation in which you've created your cookie, say one named `Name` with a value of `Bob`, and you want to change its value? To do this you can simply set the same cookie again, but with the new value. To change the cookie named `Name` from a value of `Bob` to a value of `Bobby` you'd need the following code:

```
setCookie("Name", "Bobby", "", "");
```

What if you want to delete an existing cookie? Well that's easy. Just make it expire by changing its value and setting its expiration date to a date in the past, as in the following example:

```
setCookie("Name", "", "", "Mon, 1 Jan 1990 00:00:00");
```

Getting a Cookie's Value

In the preceding example you used `document.cookie` to retrieve a string containing information about the cookies that have been set. However, this string has two limitations.

First, the cookies are retrieved in name/value pairs, with each individual cookie separated by a semi-colon. The `expires`, `path`, `domain`, and `secure` parts of the cookie are not available to you and cannot be retrieved.

Second, the `cookie` property enables you to retrieve only *all* the cookies set for a particular path and, when they are hosted on a web server, that web server. So, for example, there's no simple way of just getting the value of a cookie with the name `Age`. To do this you'll have to use the string manipulation techniques you learned in previous chapters to cut the information you want out of the returned string.

A lot of different ways exist to get the value of an individual cookie, but the way you'll use has the advantage of working with all cookie-enabled browsers. You use the following function:

```
function getCookieValue(cookieName)
{
    var cookieValue = document.cookie;
    var cookieStartsAt = cookieValue.indexOf(" " + cookieName + "=");
    if (cookieStartsAt == -1)
    {
        cookieStartsAt = cookieValue.indexOf(cookieName + "=");
    }
}
```

```
if (cookieStartsAt == -1)
{
    cookieValue = null;
}
else
{
    cookieStartsAt = cookieValue.indexOf("=", cookieStartsAt) + 1;
    var cookieEndsAt = cookieValue.indexOf(";", cookieStartsAt);
    if (cookieEndsAt == -1)
    {
        cookieEndsAt = cookieValue.length;
    }
    cookieValue = unescape(cookieValue.substring(cookieStartsAt,
        cookieEndsAt));
}
return cookieValue;
}
```

The first task of the function is to get the `document.cookie` string and store it in the variable `cookieValue`.

```
var cookieValue = document.cookie;
```

Next you need to find out where the cookie with the name passed as a parameter to the function is within the `cookieValue` string. You use the `indexOf()` method of the `String` object to find this information, as shown in the following line:

```
var cookieStartsAt = cookieValue.indexOf(" " + cookieName + "=");
```

The method will return either the character position where the individual cookie is found or `-1` if no such name, and therefore no such cookie, exists. You search on `" " + cookieName + "="` so that you don't inadvertently find cookie names or values containing the name that you require. For example, if you have `xFoo`, `Foo`, and `yFoo` as cookie names, a search for `Foo` without a space in front would match `xFoo` first, which is not what you want!

If `cookieStartsAt` is `-1`, the cookie either does not exist or it's at the very beginning of the cookie string so there is no space in front of its name. To see which of these is true, you do another search, this time with no space.

```
if (cookieStartsAt == -1)
{
    cookieStartsAt = cookieValue.indexOf(cookieName + "=");
}
```

In the next `if` statement you check to see whether the cookie has been found. If it hasn't, you set the `cookieValue` variable to `null`.

```
if (cookieStartsAt == -1)
{
    cookieValue = null;
}
```

Chapter 11: Storing Information: Cookies

If the cookie has been found, you get the value of the cookie you want from the `document.cookie` string in an `else` statement. You do this by finding the start and the end of the value part of that cookie. The start will be immediately after the equals sign following the name. So in the following line, you find the equals sign following the name of the cookie in the string by starting the `indexOf()` search for an equals sign from the character at which the cookie name/value pair starts.

```
else
{
    cookieStartsAt = cookieValue.indexOf("=", cookieStartsAt) + 1;
```

You then add one to this value to move past the equals sign.

The end of the cookie value will either be at the next semicolon or at the end of the string, whichever comes first. You do a search for a semicolon, starting from the `cookieStartsAt` index, in the next line.

```
var cookieEndsAt = cookieValue.indexOf(";", cookieStartsAt);
```

If the cookie you are after is the last one in the string, there will be no semicolon and the `cookieEndsAt` variable will be `-1` for no match. In this case you know the end of the cookie value must be the end of the string, so you set the variable `cookieEndsAt` to the length of the string.

```
if (cookieEndsAt == -1)
{
    cookieEndsAt = cookieValue.length;
}
```

You then get the cookie's value using the `substring()` method to cut the value that you want out of the main string. Because you have encoded the string with the `escape()` function, you need to `unescape()` it to get the real value, hence the use of the `unescape()` function.

```
cookieValue = unescape(cookieValue.substring(cookieStartsAt,
    cookieEndsAt));
}
```

Finally you return the value of the cookie to the calling function.

```
return cookieValue;
```

Try It Out What's New?

Now you know how to create and retrieve cookies. Let's use this knowledge in an example in which you check to see if any changes have been made to a web site since the user last visited it.

You'll be creating two pages for this example. The first is the main page for a web site; the second is the page with details of new additions and changes to the web site. A link to the second page will appear on the first page only if the user has visited the page before (that is, if a cookie exists) but has not visited since the page was last updated.

Let's create the first page.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD html 4.01 Transitional//EN">
<html>
<head>
<title>Cookie Example</title>
<script language="JavaScript" type="text/javascript">
var lastUpdated = new Date("Tue, 28 Dec 2020");
function getCookieValue(cookieName)
{
    var cookieValue = document.cookie;
    var cookieStartsAt = cookieValue.indexOf(" " + cookieName + "=");
    if (cookieStartsAt == -1)
    {
        cookieStartsAt = cookieValue.indexOf(cookieName + "=");
    }
    if (cookieStartsAt == -1)
    {
        cookieValue = null;
    }
    else
    {
        cookieStartsAt = cookieValue.indexOf("=", cookieStartsAt) + 1;
        var cookieEndsAt = cookieValue.indexOf(";", cookieStartsAt);
        if (cookieEndsAt == -1)
        {
            cookieEndsAt = cookieValue.length;
        }
        cookieValue = unescape(cookieValue.substring(cookieStartsAt,
            cookieEndsAt));
    }
    return cookieValue;
}
function setCookie(cookieName, cookieValue, cookiePath, cookieExpires)
{
    cookieValue = escape(cookieValue);
    if (cookieExpires == "")
    {
        var nowDate = new Date();
        nowDate.setMonth(nowDate.getMonth() + 6);
        cookieExpires = nowDate.toGMTString();
    }
    if (cookiePath != "")
    {
        cookiePath = ";Path=" + cookiePath;
    }
    document.cookie = cookieName + "=" + cookieValue +
        ";expires=" + cookieExpires + cookiePath;
}
</script>
</head>
<body>
<h2 align=center>Welcome to my website</h2>
<br><br>
<center>
```

Chapter 11: Storing Information: Cookies

```
<script>
var lastVisit = getCookieValue("LastVisit");
if (lastVisit != null)
{
    lastVisit = new Date(lastVisit);
    if (lastVisit < lastUpdated)
    {
        document.write("<A href=\"WhatsNew.htm\">");
        document.write("<img src=\"WhatsNew.gif\" border=0></A>");
    }
}
var nowDate = new Date();
setCookie("LastVisit", nowDate.toGMTString(), "", "")
</script>
</center>
</body>
</html>
```

This page needs to be saved as `MainPage.htm`. Note that it contains the two functions, `setCookie()` and `getCookieValue()`, that you created earlier. Also note that the image `WhatsNew.gif` is referenced by this page; either create such an image, or retrieve the image from the code download.

Next, you'll just create a simple page to link to for the What's New details.

```
<html>
<body>
<h2 align=center>Here's what's new on this website</h2>
</body>
</html>
```

Save this page as `WhatsNew.htm`.

Load `MainPage.htm` into a browser. The first time you go to the main page, there will be nothing but a heading saying *Welcome to my website*. Obviously, if this were a real web site, it would have a bit more than that, but it suffices for this example. However, refresh the page and suddenly you'll see the page shown in Figure 11-14.

If you click the image you're taken to the `WhatsNew.htm` page detailing all the things added to the web site since you last visited. Obviously nothing has actually changed in your example web site between you loading the page and then refreshing it. You got around this for testing purposes by setting the date when the web site last changed, stored in variable `lastUpdated`, to a date in the future (here, December 28, 2020).



Figure 11-14

How It Works

The `WhatsNew.htm` page is just a simple HTML page with no script, so you will confine your attention to `MainPage.htm`. In the head of the page in the first script block, you declare the variable `lastUpdated`.

```
var lastUpdated = new Date("Tue, 28 Dec 2020");
```

Whenever you make a change to the web site, this variable needs to be changed. It's currently set to `Tue, 28 Dec 2020`, just to make sure you see a `What's New` image when you refresh the page. A better alternative for live pages would be the `document.lastModified` property, which returns the date on which the page was last changed.

The rest of the first script block contains the two functions `getCookieValue()` and `setCookie()` that you looked at earlier. These haven't changed, so we won't discuss them in detail here.

The interesting material is in the second script block within the body of the page. First you get the date of the user's last visit from the `LastVisit` cookie using the `getCookieValue()` function.

```
var lastVisit = getCookieValue("LastVisit");
```

If it's `null`, the user has either never been here before, or it has been six or more months since the last visit and the cookie has expired. Either way, you won't put a `What's New` image up because everything is new if the user is a first-time visitor, and a lot has probably changed in the last six months — more than what your `What's New` page will detail.

Chapter 11: Storing Information: Cookies

If `lastVisit` is not `null`, you need to check whether the user visited the site before it was last updated, and if so to direct the user to a page that shows what is new. You do this within the `if` statement.

```
if (lastVisit != null)
{
    lastVisit = new Date(lastVisit);
    if (lastVisit < lastUpdated)
    {
        document.write("<A href=\"WhatsNew.htm\">");
        document.write("<img src=\"WhatsNew.gif\" border=0></A>");
    }
}
```

You first create a new `Date` object based on the value of `lastVisit` and store that back into the `lastVisit` variable. Then, in the condition of the inner `if` statement, you compare the date of the user's last visit with the date on which you last updated the web site. If things have changed since the user's last visit, you write the What's New image to the page, so the user can click it and find out what's new. Note that you use the escape character `\` for the double-quote characters inside the strings that are written to the page; otherwise, JavaScript will think they indicate the end of the string.

Finally, at the end of the script block, you reset the `LastVisit` cookie to today's date and time using the `setCookie()` function.

```
var nowDate = new Date();
setCookie("LastVisit", nowDate.toGMTString(), "", "")
```

Cookie Limitations

You should be aware of a number of limitations when using cookies.

The first is that although all modern browsers support cookies, the user may have disabled them. In Firefox you can do this by selecting the Options menu, followed by the privacy tab and the cookies tab. In IE you select Internet Options on the Tools menu. Select the Privacy tab and you can change the level with the scroll control. Most users have session cookies enabled by default. Session cookies are cookies that last for as long as the user is browsing your web site. After he's closed the browser the cookie will be cleared. More permanent cookies are also normally enabled by default. However, third-party cookies, those from a third-party site, are usually disabled. These are the cookies used for tracking people from site to site and hence the ones that raise the most privacy concerns.

Both the functions that you've made for creating and getting cookies will cause no errors when cookies are disabled, but of course the value of any cookie set will be `null` and you need to make sure your code can cope with this.

You could set a default action for when cookies are disabled. In the previous example, if cookies are disabled, the What's New image will never appear.

Alternatively, you can let the user know that your web site needs cookies to function by putting a message to that effect in the web page.

Another tactic is to actively check to see whether cookies are enabled and, if not, to take some action to cope with this, such as by directing the user to a page with less functionality that does not need cookies. How do you check to see if cookies are enabled?

In the following script you set a test cookie and then read back its value. If the value is `null`, you know cookies are disabled.

```
setCookie("TestCookie","Yes","", "");
if (getCookieValue("TestCookie") == null)
{
    alert("This website requires cookies to function");
}
```

A second limitation is on the number of cookies you can set on the user's computer for your web site and how much information can be stored in each. For each domain you can store up to 20 cookies, and each *cookie pair* — that is, the name and value of the cookie combined — must not be more than 4,096 characters in size. It's also important to be aware that all browsers do set some upper limit for the number of cookies stored. When that limit is reached, older cookies, regardless of expiration date, are often deleted. Modern browsers have a 300-cookie limit, though this may vary between browsers.

To get around the 20-cookie limit, you can store more than one piece of information per cookie. This example uses multiple cookies:

```
setCookie("Name", "Karen", "", "")
setCookie("Age", "44", "", "")
setCookie("LastVisit", "10 Jan 2001", "", "")
```

You could combine this information into one cookie, with each detail separated by a semicolon.

```
setCookie("UserDetails", "Karen;44;10 Jan 2001", "", "")
```

Because the `setCookie()` function escapes the value of the cookie, there is no confusion between the semicolons separating pieces of data in the value of the cookie, and the semicolons separating the parts of the cookie. When you get the cookie value back using `getCookieValue()`, you just split it into its constituent parts; however, you must remember the order you stored it in.

```
var cookieValues = getCookieValue("UserDetails");
cookieValues = cookieValues.split(";");
alert("Name = " + cookieValues[0]);
alert("Age = " + cookieValues[1]);
alert("Last Visit = " + cookieValues[2]);
```

Now you have acquired three pieces of information and still have 19 cookies left in the jar.

Cookie Security and IE6 and IE7

IE6 has introduced a new security policy for cookies based on the P3P recommendations made by the World Wide Web Consortium (W3C), a web standards body that deals with not only cookies but HTML, XML, and various other browser standards. (You'll learn more about W3C in Chapter 13. Its web site is

Chapter 11: Storing Information: Cookies

at www.w3.org and contains a host of information, though it's far from being an easy read.) The general aim of P3P is to reassure users who are worried that cookies are being used to obtain personal information about their browsing habits. In IE 6 and 7 you can select Tools ⇨ Internet Options and click the Privacy tab to see where you can set the level of privacy with regards to cookies (see Figure 11-15). You have to strike a balance between setting it so high that no web site will work and so low that your browsing habits and potentially personal data may be recorded.

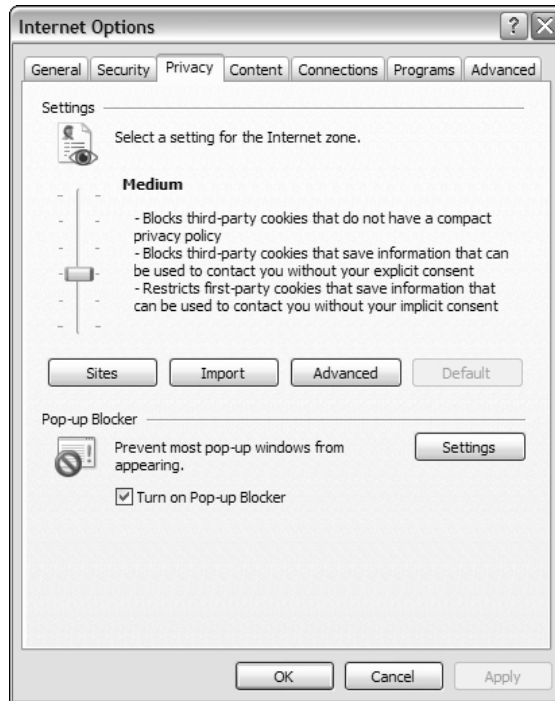


Figure 11-15

Generally, by default session cookies—cookies that last for only as long as the user is browsing your web site—are allowed. As soon as the user closes the browser, the session ends. However, if you want cookies to outlast the user's visit to your web site, you need to create a privacy policy in line with the P3P recommendations. This sounds a little complex, and certainly the fine details of the policy can be. However, IBM has created software that makes creating the XML for the policy fairly easy. It's not cheap, but there is a 90-day free trial. It can be downloaded from www.alphaworks.ibm.com/tech/p3peditor.

Plenty of other policy creation software is available; this just happens to be quite easy to use. P3PEdit is available for much lower cost from <http://policyeditor.com/>.

Try It Out Storing Previous Quiz Results

Let's return to the trivia quiz one last time and, using your knowledge of cookies, add the functionality to keep track of previous quiz results. You're going to calculate the user's average score for all the completed quizzes. You'll also allow the user to reset the statistics.

You need to alter only one page of the trivia quiz: the `GlobalFunctions.htm` page. First you need to add the two cookie functions, `getCookieValue()` and `setCookie()`, that you introduced earlier in the chapter. You can add these anywhere within the script block—here they’ve been added after all the other functions.

```
function getCookieValue(cookieName)
{
    var cookieValue = document.cookie;
    var cookieStartsAt = cookieValue.indexOf(" " + cookieName + "=");
    if (cookieStartsAt == -1)
    {
        cookieStartsAt = cookieValue.indexOf(cookieName + "=");
    }
    if (cookieStartsAt == -1)
    {
        cookieValue = null;
    }
    else
    {
        cookieStartsAt = cookieValue.indexOf("=", cookieStartsAt) + 1;
        var cookieEndsAt = cookieValue.indexOf(";", cookieStartsAt);
        if (cookieEndsAt == -1)
        {
            cookieEndsAt = cookieValue.length;
        }
        cookieValue = unescape(cookieValue.substring(cookieStartsAt,
            cookieEndsAt));
    }
    return cookieValue;
}

function setCookie(cookieName, cookieValue, cookiePath, cookieExpires)
{
    cookieValue = escape(cookieValue);
    if (cookieExpires == "")
    {
        var nowDate = new Date();
        nowDate.setMonth(nowDate.getMonth() + 6);
        cookieExpires = nowDate.toGMTString();
    }
    if (cookiePath != "")
    {
        cookiePath = ";Path=" + cookiePath;
    }
    document.cookie = cookieName + "=" + cookieValue +
        ";expires=" + cookieExpires + cookiePath;
}
```

The final change you need to make is to the `getQuestion()` function. It’s in this function that either a new question is written to the page or, if all the questions have been asked, the final results are displayed. Currently you just write the number of questions that the user got right and rate the result. Now you’re going to keep a running average of previous results and display this information as well. The addition is toward the end of the function, after the script that writes a rating to the page.

Chapter 11: Storing Information: Cookies

```
        default:
            questionHTML = questionHTML + "Excellent"
        }
    }
    var previousNoCorrect = Math.floor(getCookieValue("previousNoCorrect"));
    var previousNoAsked = Math.floor(getCookieValue("previousNoAsked"));
    var currentAvgScore = Math.round(numberOfQuestionsCorrect /
        numberOfQuestionsAsked * 100);
    questionHTML = questionHTML + "<br>The percentage you've " +
        " answered correctly in this quiz is " + currentAvgScore + "%";
    if (previousNoAsked == 0)
    {
        previousNoCorrect = 0;
    }

    previousNoCorrect = previousNoCorrect + numberOfQuestionsCorrect;
    previousNoAsked = previousNoAsked + numberOfQuestionsAsked;

    currentAvgScore = Math.round(previousNoCorrect / previousNoAsked * 100);
    setCookie("previousNoAsked", previousNoAsked, "", "");
    setCookie("previousNoCorrect", previousNoCorrect, "", "");
    questionHTML = questionHTML + "<br>This brings your average todate to " +
        currentAvgScore + "%"
    questionHTML = questionHTML + "<p><input type=button " +
        "value='Reset Stats' " +
        "onclick=\"window.top.fraTopFrame.fraGlobalFunctions.setCookie\" +
        \"('previousNoAsked', 0, '', '1 Jan 1970')\" \" +
        \"name=buttonReset>\"
    questionHTML = questionHTML + "<p><input type=button " +
        "value='Restart Quiz' " +
        "onclick=\"window.location.replace('quizpage.htm')\" \" +
        \"name=buttonRestart>\"
    }
    return questionHTML;
}
```

How It Works

So how does this new code work?

First you use cookies to retrieve the number of questions previously answered correctly and the number of questions previously asked in total.

```
var previousNoCorrect = Math.floor(getCookieValue("previousNoCorrect"));
var previousNoAsked = Math.floor(getCookieValue("previousNoAsked"));
```

Note that if no cookie is set, `CookieValue()` will return `null`, which `Math.floor()` converts to 0.

You then work out the average of the current score, which is simply the number of questions the user got correct in this quiz divided by the number the user answered and multiplied by 100.

```
var currentAvgScore = Math.round(numberOfQuestionsCorrect /
    numberOfQuestionsAsked * 100);
```

You then add the average result to `questionHTML`, the string of HTML you'll be writing to the page.

```
questionHTML = questionHTML + "<br>The percentage you've " +  
    " answered correctly in this quiz is " + currentAvgScore + "%";
```

Next you check to see if the number of questions asked was previously 0. If it was, the user has reset the stats, or this is the first time she has taken the quiz. You reset the number correct to 0, because it would hardly make sense if out of no questions answered your user got 10 right!

```
if (previousNoAsked == 0)  
{  
    previousNoCorrect = 0;  
}
```

Then you update the number of questions that the user previously answered correctly and the total number of previously answered questions, before using this information to calculate the running average of all the quizzes the user has taken since playing your quiz or resetting the stats.

```
previousNoCorrect = previousNoCorrect + numberOfQuestionsCorrect;  
previousNoAsked = previousNoAsked + numberOfQuestionsAsked;  
  
currentAvgScore = Math.round(previousNoCorrect / previousNoAsked * 100);
```

You update your cookies with the new values in the next lines, using the `setCookie()` function you created earlier.

```
setCookie("previousNoAsked", previousNoAsked, "", "");  
setCookie("previousNoCorrect", previousNoCorrect, "", "");
```

In the final new lines, you complete the construction of the results string, adding the running average and then creating two buttons. The first button will reset the number of questions asked by setting the `previousNoAsked` variable to 0, and the second is a replacement for the link that restarts the quiz.

```
questionHTML = questionHTML + "<br>This brings your average to date to " +  
    currentAvgScore + "%"  
questionHTML = questionHTML + "<p><input type=button " +  
    "value='Reset Stats' " +  
    "onclick=\"window.top.fraTopFrame.fraGlobalFunctions.setCookie\" +  
    \"('previousNoAsked', 0, '', '1 Jan 1970')\" \" " +  
    "name=buttonReset>"  
questionHTML = questionHTML + "<p><input type=button " +  
    "value='Restart Quiz' " +  
    "onclick=\"window.location.replace('quizpage.htm')\" \" " +  
    "name=buttonRestart>"
```

On the button creation lines, you need to include single quotes inside double quotes inside more double quotes. This will confuse JavaScript because it'll think the string has ended before it actually has. This is the reason that you use the special escape character `\`, which indicates that the double quotes are not delimiting a string, but are part of the string itself.

Save the page and load `TriviaQuiz.htm`.

Chapter 11: Storing Information: Cookies

Now when you complete the quiz, you'll see a summary something like the one in Figure 11-16.

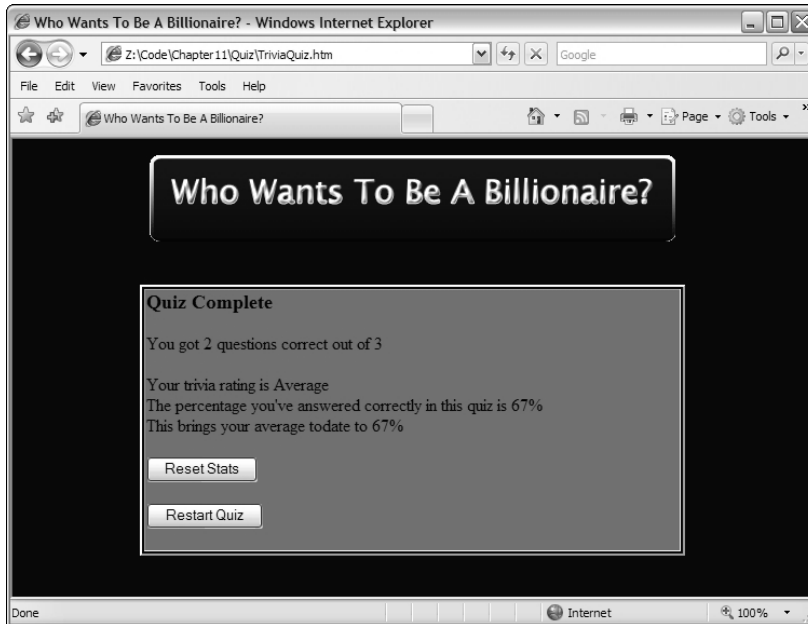


Figure 11-16

That completes the changes to the trivia quiz for this chapter.

Summary

In this chapter, you looked at how you can store information on the user's computer and use this information to personalize the web site. In particular you found the following:

- ❑ The key to cookies is the `document.cookie` property.
- ❑ Creating a cookie simply involves setting the `document.cookie` property. Cookies have six different parts you can set. These are the name, the value, when it expires, the path it is available on, the domain it's available on, and finally whether it should be sent only over secure connections.
- ❑ Although setting a new cookie is fairly easy, you found that retrieving its value actually gets all the cookies for that domain and path, and that you need to split up the cookie name/value pairs to get a specific cookie using `String` object methods.
- ❑ Cookies have a number of limitations. First, the user can set the browser to disable cookies, and second, you are limited to 20 cookies per domain and a maximum of 4,096 characters per cookie name/value pair.
- ❑ Finally, you added the display of a running average for the trivia quiz.

Exercise Questions

Suggested solutions to these questions can be found in Appendix A.

Question 1

Create a page that keeps track of how many times the page has been visited by the user in the last month.

Question 2

Use cookies to load a different advertisement every time a user visits a web page.

12

Introduction to Dynamic HTML

Dynamic HTML (DHTML) is a term with various meanings, but it can be boiled down to one basic concept: dynamically changing a web page after it is loaded into the browser. For example, it can be used to change the color and style of text when the mouse pointer moves over it. It is also widely used to enhance the experience a visitor has when visiting a web page, making the page more interactive.

Before the advent of DHTML, web pages were static. That is, when an HTML document was downloaded and displayed in the user's browser, no changes could be made to the page's content or appearance. With DHTML, however, you can change the appearance of the existing content and even move content around on the screen. Want to change that heading color to red? No problemo. Want to make text scroll across the screen? You got it.

JavaScript provides you with a means of writing DHTML pages. You've already seen this in action with the example in Chapter 5 in which you enabled the user to change an image loaded into the page with the click of a button. Today's browsers go beyond this, and you can change virtually anything in a web page.

In this chapter, you'll take an introductory look at DHTML. You'll look at how you can change things and move things around on the page. At the end of the chapter, you'll build a scrolling marquee to display whatever text you desire.

Cross-Browser Issues

In mid-1997, Netscape released the fourth version of its Navigator browser. Not only was this an iteration from Navigator 3, but it was the first browser to support DHTML. This support, however, was rudimentary at best. Netscape chose to implement its own proprietary version of DHTML. This implementation, while unique, extremely limited what users could accomplish with the browser.

Chapter 12: Introduction to Dynamic HTML

Later that year, Microsoft released Internet Explorer 4 (IE4), and it, too, supported DHTML. Microsoft's offering, however, far exceeded that of Netscape's. It was still far from perfect, but DHTML in IE4 enabled you to do so much more than Netscape allowed. The entrance of this browser quickly ushered DHTML to the forefront and IE4 became the browser of choice for developers.

Despite the favor IE4 held at the time, it was advantageous for developers to build DHTML scripts that worked in both browsers, as Netscape still held a good portion of the browser market. Some called this obligation to code for both browsers "DHTML hell," and rightly so. The amount of JavaScript needed for one DHTML script could easily double due to *code branching* (writing code specifically for a particular browser by using browser detection) and getting said script to work correctly (especially in Netscape) was very time-consuming.

Fast-forward to today, and the number of browsers has doubled: We now have Internet Explorer 7, Firefox 1.5, Safari 2, and Opera 9. However, the DHTML hell of the late '90s is nothing more than a bad memory. Today's modern browsers are far more consistent in their support for DHTML, thankfully. But a few gotchas still exist — the most prominent being event handling.

Events

Events are an important part of DHTML. Most of the time, DHTML code reacts to something that the user did: the content's color changed when he moved his mouse over certain content, a hidden element showed itself when he clicked a specific link, or a list of words popped up that match the keystrokes the user made.

In Chapter 5, you learned about events and how to connect code to them, and every example you saw and worked through works in every browser. However, as you move on to more advanced event handling concepts, you'll start to notice differences between browsers. These browsers can easily be divided into two groups: Internet Explorer (IE) and the other modern-day browsers (non-IE).

Event Handlers as Attributes (Revisited)

You learned in Chapter 5 that the most common way to add an event is to add the event handler's name and the code you want to execute to the HTML tag's attributes. This is an easy way to add an event handler to a specific HTML tag:

```
<html>
<body>
  <a href="somepage.htm" name="linkSomePage" onmouseover="alert('You Moved?') "
    onclick="alert('You Clicked?') ">
    Click Me
  </a>
</body>
</html>
```

This HTML page contains one link. When you move your mouse pointer over it, an alert box displays the text *You Moved?*; it also displays the message *You Clicked?* when you click it. Simple, right? This works in all the major browsers. So what's the problem with it?

What if you wanted to do this on several links within a page? Sure, you could copy and paste the event attributes as many times as you want, but if you ever wanted to change the messages in the alert boxes,

you'd have to edit every one that you used. Also, what if you wanted to use only one function to handle both the `onmouseover` and `onclick` events and display the appropriate message according to which event was fired?

This is where things get a little complicated, and you have to adjust your code to accommodate the differences between IE and the other browsers.

Events in Internet Explorer

When an event fires in IE, the browser populates a global object called `event` with the data associated with the fired event. You can access this object through the `window` object like this:

```
var myEvent = window.event;
```

`event` has a variety of properties, each containing a specific piece of information that you can use. The first you'll cover is the `type` property, which you can use to retrieve the type of event that fired.

The Event Type

The `type` property returns a string containing the name of the event without the `on` prefix. So the `onclick` event is returned as `click`, and the `onmouseover` event is returned as `mouseover`.

Try It Out Using the type Property

Let's look at how you can use this useful property in an example.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
    <title>Using the type Property</title>

    <script type="text/javascript">
        function paragraph_eventHandler() {
            if (window.event.type == "click") { //The click event was fired.
                alert("You Clicked Me!");
            } else if (window.event.type == "mouseover") { //The mouseover event.
                alert("You Tickled Me!");
            }
        }
    </script>
</head>
<body>
    <p onmouseover="paragraph_eventHandler()" >
        Move your mouse over me.
    </p>
    <p onclick="paragraph_eventHandler()" >
        Click me!
    </p>
</body>
</html>
```

Chapter 12: Introduction to Dynamic HTML

Save this as `IE_type_property.htm`. When you load this page into IE, you'll see some plain, ordinary text in two paragraphs. When you move your mouse over the first paragraph, you're greeted with an alert box stating `You Tickled Me!` (see Figure 12-1).

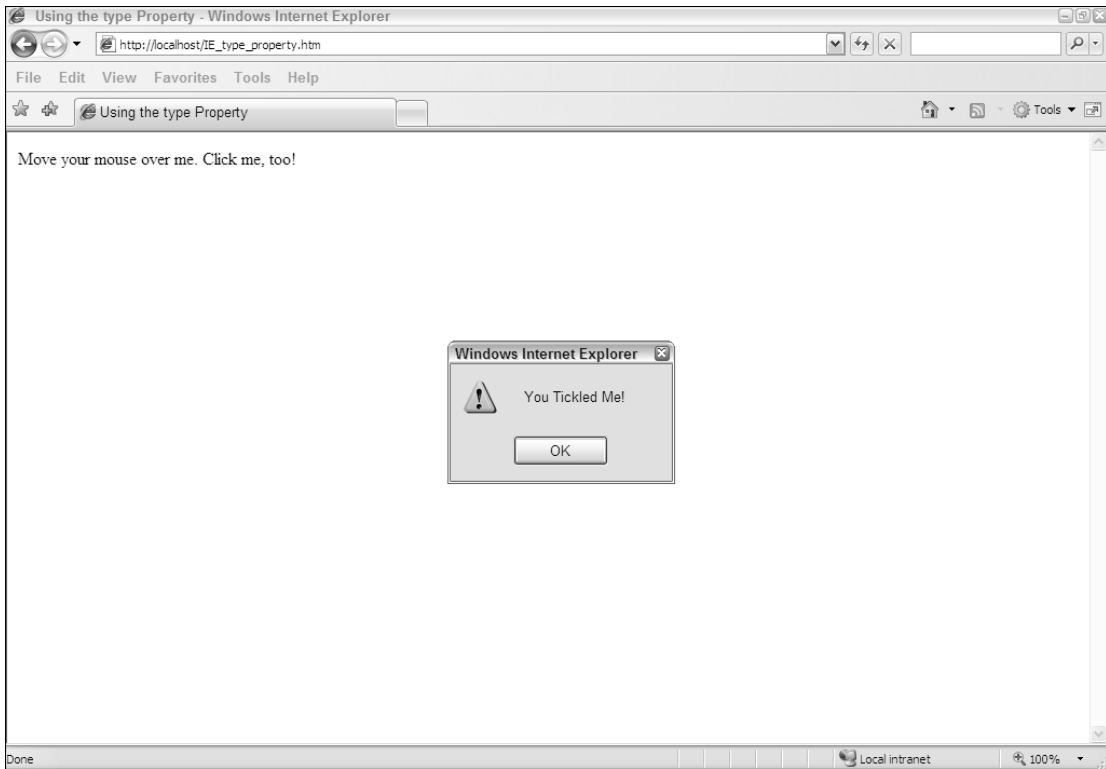


Figure 12-1

If you click the text in the second paragraph, you'll see an alert box saying `You Clicked Me!`, as shown in Figure 12-2.

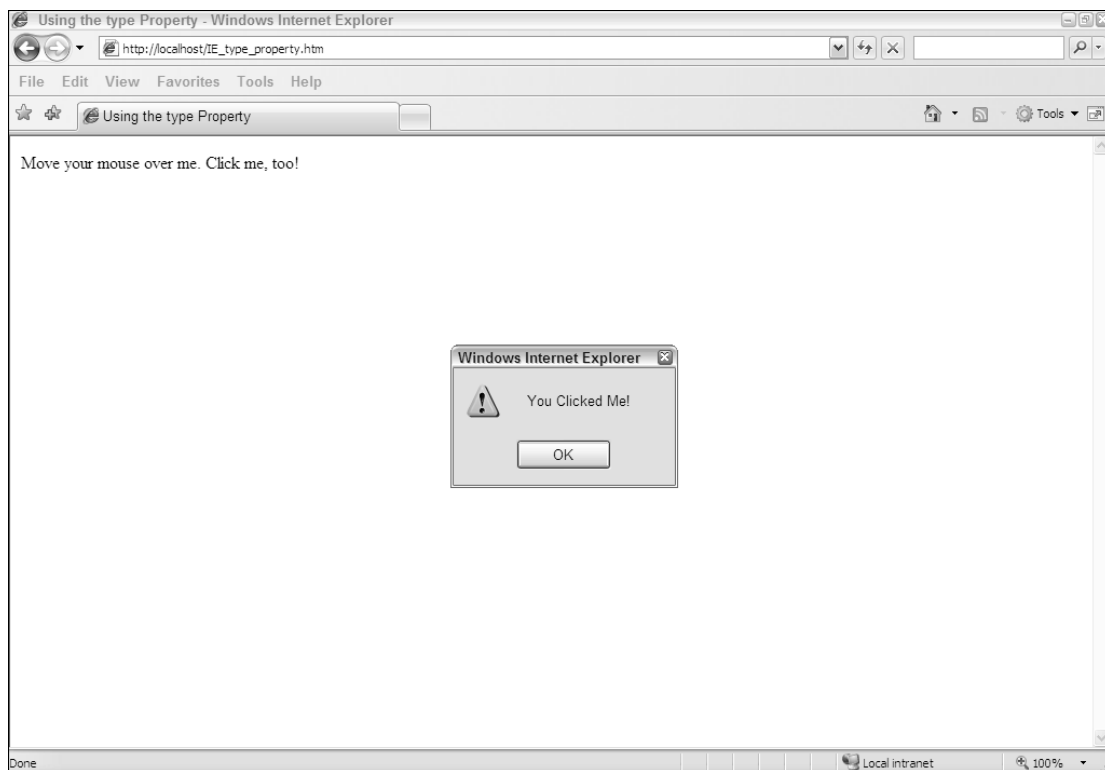


Figure 12-2

How It Works

The body of the page contains two `<p/>` elements, both of which contain text. The first paragraph assigns the `paragraph_eventHandler()` function to handle the `mouseover` event. So when this paragraph is clicked, the browser knows to call the event handler.

```
<p onmouseover="paragraph_eventHandler()">  
    Move your mouse over me.  
</p>
```

The second paragraph follows the same concept. However, instead of handling the `mouseover` event, `paragraph_eventHandler()` handles the `click` event.

```
<p onclick="paragraph_eventHandler()">  
    Click me!  
</p>
```

Chapter 12: Introduction to Dynamic HTML

A script block resides in the head of the page. The JavaScript in this code block consists of only one function, called `paragraph_eventHandler()`, which handles the `click` and `mouseover` events of the two paragraphs created earlier.

```
function paragraph_eventHandler() {  
    //more code here  
}
```

The function determines the type of event by using the `window.event.type` property:

```
function paragraph_eventHandler() {  
    if (window.event.type == "click") { //The click event was fired.  
        //more code here  
    } else if (window.event.type == "mouseover") { //The mouseover event.  
        //more code here  
    }  
}
```

To determine the event, compare the `type` property to the event name. Unlike the names of event handlers, the names of events do not have the `on` prefix. Therefore, the desired events for this example are `click` and `mouseover`. Then, based upon the event, the function uses the `alert()` method to display the appropriate message to the user like this:

```
function paragraph_eventHandler() {  
    if (window.event.type == "click") { //The click event was fired.  
        alert("You Clicked Me!");  
    } else if (window.event.type == "mouseover") { //The mouseover event.  
        alert("You Tickled Me!");  
    }  
}
```

Now when the user causes these events to fire by moving or clicking her mouse, an alert box displays a message corresponding to the event that was fired.

The `type` property is supported by Internet Explorer, Firefox, Safari, and Opera. The only difference is in how you access it, which is discussed later.

The `srcElement` Property

Another valuable piece of information offered by the `event` object is the `srcElement` property. This property retrieves the HTML element that receives the event. For example, consider the following HTML:

```
<a href="somePage.htm" onclick="alert('Hello! You clicked me!')">Click Me</a>
```

When you click this link, an alert box displays the message `Hello! You clicked me! That's obvious, right?` What may not be so obvious is what actually receives the click event: the `<a/>` element. If you were to check the `srcElement` property during this event, it would point to this particular `<a/>` element. This information is particularly useful when you need to manipulate the element that received the event.

Try It Out The `srcElement` Property

The following code demonstrates an image rollover:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
    <title>Using the srcElement Property</title>

    <script type="text/javascript">
function image_eventHandler() {
    var sourceElement = window.event.srcElement; //Get the element.
    var eventType = window.event.type; //Get the type of event.

    if (eventType == "mouseover") { //The mouse rolled over the image.
        sourceElement.src = "o.gif"; //So change the image's src property.
    }

    if (eventType == "mouseout") { //The mouse moved out.
        sourceElement.src = "x.gif"; //So change it back to the original.
    }
}
    </script>
</head>
<body>
    
</body>
</html>
```

Save this file as `IE_srcElement_property.htm`. Figure 12-3 shows what the page looks like when loaded into IE.

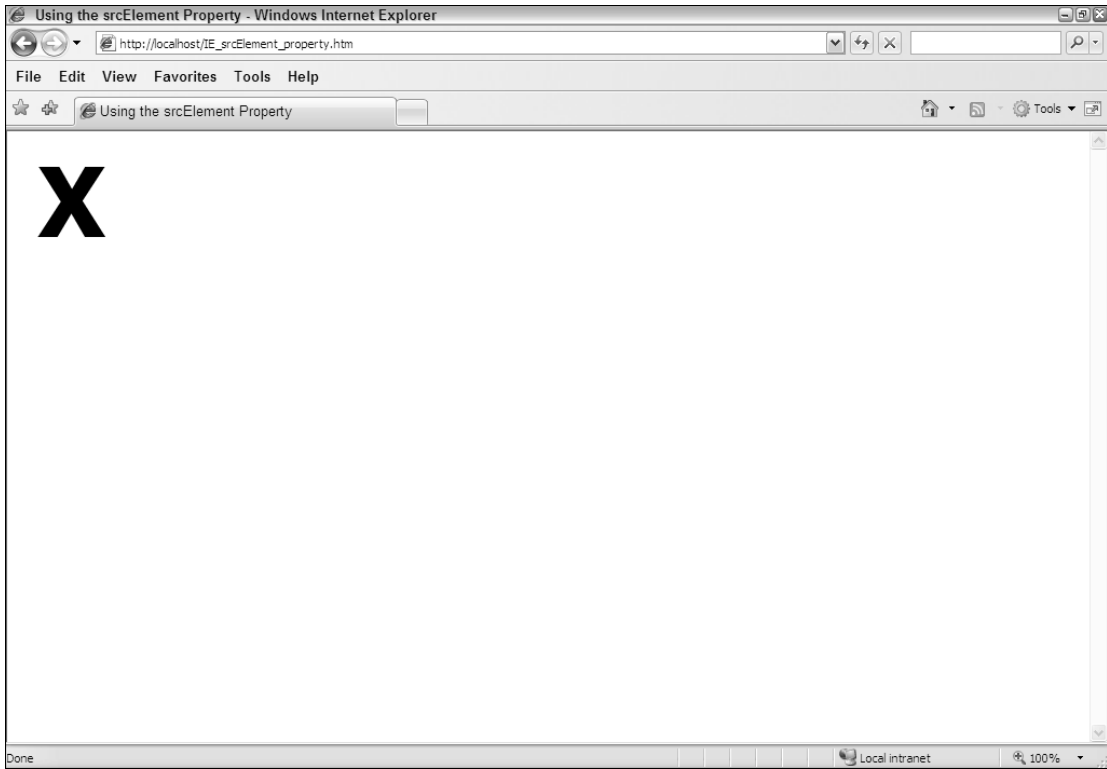


Figure 12-3

Move your mouse over the picture and it will change from an X to an O (see Figure 12-4). Moving the mouse pointer off the image makes it revert back to X.

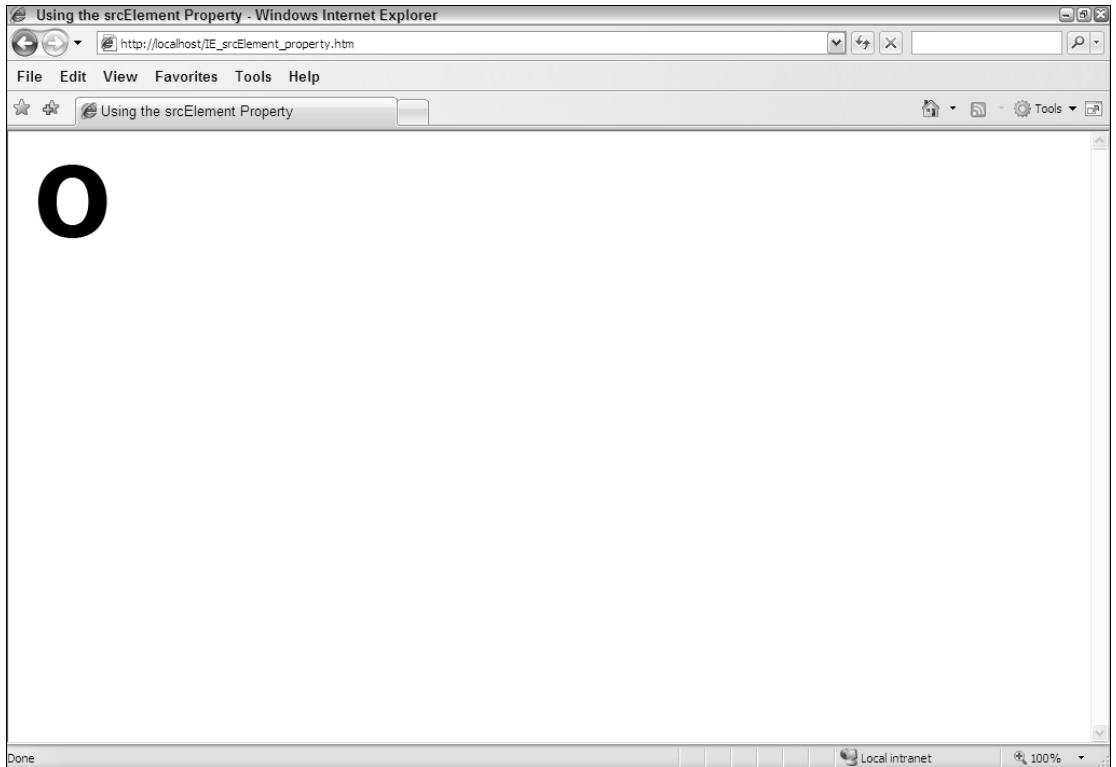


Figure 12-4

How It Works

The script block contains one function, called `image_EventHandler()`, which handles the `mouseover` and `mouseout` events for the `` element in the page's body. The first two lines of this function retrieve `window.event.srcElement` and `window.event.type`. These values are assigned to the `sourceElement` and `eventType` variables respectively.

```
function image_eventHandler() {  
    var sourceElement = window.event.srcElement;  
    var eventType = window.event.type;  
  
    //more code here  
}
```

Next, check the event's type to see if it was the `mouseover` event. To do this, compare `eventType` to the string `mouseover`.

```
function image_eventHandler() {  
    var sourceElement = window.event.srcElement;
```

Chapter 12: Introduction to Dynamic HTML

```
var eventType = window.event.type;

if (eventType == "mouseover") { //The mouse rolled over the image
    sourceElement.src = "o.gif"; //So change the image's src property.
}

//more code here
}
```

If it is the `mouseover` event, then change the image's `src` property to `o.gif`. This changes the picture displayed by the `` element to show an O. Now check for the `mouseout` event and change the image's `src` property back to `x.gif`.

```
function image_eventHandler() {
    var sourceElement = window.event.srcElement;
    var eventType = window.event.type;

    if (eventType == "mouseover") { //The mouse rolled over the image
        sourceElement.src = "o.gif"; //So change the image's src property.
    }

    if (eventType == "mouseout") { //The mouse moved out.
        sourceElement.src = "x.gif"; //So change it back to the original.
    }
}
```

Because you did this, the image now displays an X again, and it will stay that way until the mouse pointer moves over it again.

Events in Other Browsers

Even though the implementation between the IE and the non-IE browsers differs, the basic principles are the same. Unlike IE, however, non-IE browsers do not have a global `event` object that keeps track of the events. Instead, it is in the hands of developers (meaning you) to access the events. Consider the following HTML:

```
<p onmouseover="paragraph_eventHandler(event)">
    Move your mouse over me.
</p>
<p onclick="paragraph_eventHandler(event)">
    Click me!
</p>
```

This HTML may look familiar. In fact, it was used in the Try It Out section for the `window.event.type` property. But there's a small change: The event, represented by `event`, is explicitly passed to `paragraph_eventHandler()`. The strange thing about this variable is that it is not defined anywhere; instead, it is an argument used only with event handlers connected through HTML attributes. It contains a reference to the current event object.

It is important not to confuse this event object with IE's `event` object. They may share the same name, but the similarities almost end there. Non-IE browsers do not have a global event object like `window.event`; their event object is a predefined attribute that is passed to the event handlers when the event fires.

Because the event object is now being passed to the `paragraph_eventHandler()` function, the function definition must accommodate this functionality.

```
function paragraph_eventHandler(evt) {  
  
}
```

This code simply adds an argument, called `evt`, to the function, which gives you easy access to the event object passed to the function.

Try It Out The type Property for the Other Browsers

Although the non-IE event object differs from the IE event object, they both expose a few of the same properties. One such property is the `type` property, which performs the same function in IE and non-IE browsers alike. This example shows how to use it.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"  
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  
  
<html xmlns="http://www.w3.org/1999/xhtml" >  
<head>  
  <title>Using the type Property in Other Browsers</title>  
  
  <script type="text/javascript">  
    function paragraph_eventHandler(evt) {  
      if (evt.type == "click") { //The click event was fired.  
        alert("You Clicked Me!");  
      } else if (evt.type == "mouseover") { //The mouseover event was fired.  
        alert("You Tickled Me!");  
      }  
    }  
  </script>  
</head>  
<body>  
  <p onmouseover="paragraph_eventHandler(event)">  
    Move your mouse over me.  
  </p>  
  <p onclick="paragraph_eventHandler(event)">  
    Click me!  
  </p>  
</body>  
</html>
```

Save this as `OB_type_property.htm` (OB means “other browsers”). Load this page into Firefox, Safari, or Opera, and you’ll see two paragraphs that contain text. Move the mouse pointer over the first paragraph, and an alert box displays the text `You Tickled Me!`. Clicking on the second paragraph shows another alert box displaying the text `You Clicked Me!`

How It Works

Two paragraphs are defined in the body of the page (déjà vu!). The first paragraph handles the mouseover event by assigning the `onmouseover` attribute to `paragraph_eventHandler()` and passing the event object to the function. The second paragraph’s `onclick` attribute is set to the same function, and handles the `click` event.

The event object passed to the event handlers must be named `event`; otherwise, the code will not work.

In the script block, the `paragraph_eventHandler()` function is defined, and it accepts one argument, called `evt`. This argument provides easy access to the event object.

```
function paragraph_eventHandler(evt) {  
    //more code here  
}
```

Next, the code checks to see which event called the event handler. It does this by using the `type` property and comparing it to the event names (again, without the `on` prefix).

```
function paragraph_eventHandler(evt) {  
    if (evt.type == "click") { //The click event was fired.  
        alert("You Clicked Me!");  
    } else if (evt.type == "mouseover") { //The mouseover event was fired.  
        alert("You Tickled Me!");  
    }  
}
```

Not many differences separate this code from the code in the IE-specific example. The only difference is in how the `type` property is accessed. IE uses the `window.event` object, and the non-IE browsers use the implicit event object, which is passed to the event handler.

The target Property

When an event fires, it is useful to know what element it fired on. IE grants developers access to this information with the `srcElement` property. The other browsers' event object does not support this specific property, but it does have the `target` property. This property retrieves the HTML element that received the event.

Try It Out The target Property

Use of the `target` property closely resembles that of IE's `srcElement` property. The main difference is in how you access the property, and this example shows you how.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"  
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  
  
<html xmlns="http://www.w3.org/1999/xhtml" >  
<head>  
    <title>Using the target Property</title>  
  
    <script type="text/javascript">  
        function image_eventHandler(evt) {  
            var eventTarget = evt.target; //Get the element.  
            var eventType = evt.type; //Get the type of event.  
  
            if (eventType == "mouseover") { //The mouse rolled over the image.
```

```
        eventTarget.src = "o.gif"; //So change the image's src property.
    }

    if (eventType == "mouseout") { //The mouse moved out.
        eventTarget.src = "x.gif"; //So change it back to the original picture.
    }
}
</script>
</head>
<body>
    
</body>
</html>
```

Save this file as `OB_target_property.htm`. When it's loaded into Firefox, Safari, or Opera, you'll see a picture of an X. When the mouse pointer moves over the `` element, the picture changes to an O. The picture reverts back to an X when the mouse pointer exits the `` element.

How It Works

The only JavaScript code in the script block is one function: `image_eventHandler()`, which handles the `mouseover` and `mouseout` events of the `` element in the page's body. It accepts one argument, called `evt`, which provides access to the event object. Following is the function definition.

```
function image_eventHandler(evt) {
    //more code here
}
```

In order for the rollover to work, the function needs two pieces of information: the event type, and the HTML element where the event fired.

```
function image_eventHandler(evt) {
    var eventTarget = evt.target; //Get the element.
    var eventType = evt.type; //Get the type of event.

    //more code here
}
```

The first line of this code creates a variable called `eventTarget`, which contains the value of the `target` property of the event object (the HTML element that received the event). Next, create the `eventType` variable and assign it the value of the fired event type.

Now use this information to check the type of event that called the function. Do this by comparing `eventType` to that of the names of event types. The following code checks to see if the `mouseover` event fired:

```
function image_eventHandler(evt) {
    var eventTarget = evt.target; //Get the element.
```

Chapter 12: Introduction to Dynamic HTML

```
var eventType = evt.type; //Get the type of event.

if (eventType == "mouseover") { //The mouse rolled over the image.
    eventTarget.src = "o.gif"; //So change the image's src property.
}

//more code here
}
```

If so, change the target's `src` property (the target is the `` element). The next step is to check to see if the event is `mouseout`. Follow the same step for the `mouseover` event, but this time compare `eventType` to the string `mouseout`.

```
function image_eventHandler(evt) {
    var eventTarget = evt.target; //Get the element.
    var eventType = evt.type; //Get the type of event.

    if (eventType == "mouseover") { //The mouse rolled over the image.
        eventTarget.src = "o.gif"; //So change the image's src property.
    }

    if (eventType == "mouseout") { //The mouse moved out.
        eventTarget.src = "x.gif"; //So change it back to the original picture.
    }
}
```

This code changes the image's picture back to `x.gif`, thereby reverting it to its original state.

Now Play Together, Kids

The challenge DHTML developers face is getting their code to work in every major browser. In the DHTML hell of yesteryear, getting code to work in the major browsers was a headache at best. Times change, thankfully, and getting DHTML code to play nicely with today's major browsers is quite simple. In fact, this is one of the easiest cross-browser problems developers face.

The first problem to tackle is how to get the browsers to retrieve the event object similarly. As a quick recap, IE has a global object called `event` (or `window.event`). Non-IE browsers require an event object, also called `event`, to be passed to the event handler. The two objects are similarly named, so use that to your advantage. Essentially, when you're assigning the events with the HTML attributes, use the method for non-IE browsers like this:

```

```

This HTML does two things. First, it allows non-IE browsers to handle the events; this is the same HTML used in the `target` property example. The event object is passed to `image_eventHandler()`. Second, and this is only for IE, the HTML passes the global `event` object to `image_eventHandler()`. Even though the `event` object is global, it is possible to pass it to other functions. Therefore, no matter if the user is viewing the web page in IE, Firefox, Safari, or Opera, the appropriate event object is passed to `image_eventHandler()`.

The second problem is how to handle the event. The image rollover DHTML script must retrieve the element that the event fired upon. In IE, this element is retrieved with the `srcElement` property. In non-IE browsers, the `target` property is the desired property. Thankfully, this problem has a straightforward solution: branch the code according to which browser is displaying the page. In Chapter 5 you learned how to use object detection to determine which browser the user is using. Do the same thing here. Consider the following code:

```
var elementTarget; //contains either the srcElement or the target property

if (evt.srcElement) { //The browser is IE
    elementTarget = evt.srcElement;
}

//more code here
```

The first line of this code creates a variable called `elementTarget`, which stores either the `srcElement` or `target` property, depending upon the browser that loaded the page. The next line uses object detection to determine the browser. In this case, the `srcElement` is checked. If it exists, then the browser is IE, and you can safely use the property.

Next, write code for the other browsers. If the `srcElement` property doesn't exist, the browser obviously isn't IE, and you can use the `target` property.

```
var elementTarget; //contains either the srcElement or the target property

if (evt.srcElement) { //The browser is IE
    elementTarget = evt.srcElement;
} else { //The browser is non-IE
    elementTarget = evt.target;
}
```

Now you can use the `elementTarget` variable to access the HTML element that the event was fired upon. The following example shows the only time the code branches to accommodate the differing browsers.

Try It Out Cross-Browser Image Rollover

This example modifies the image rollover you previously wrote to work in all modern browsers.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
    <title>Cross-browser Image Rollover</title>

    <script type="text/javascript">
        function image_eventHandler(evt) {
            var elementTarget;

            if (evt.srcElement) { //The browser is IE
                elementTarget = evt.srcElement;
```

Chapter 12: Introduction to Dynamic HTML

```
    } else { //The browser is non-IE
        elementTarget = evt.target;
    }

    if (evt.type == "mouseover") { //The mouse rolled over the image.
        elementTarget.src = "o.gif"; //So change the image's src property.
    }

    if (evt.type == "mouseout") { //The mouse moved out.
        elementTarget.src = "x.gif"; //So change it back to the original.
    }
}
</script>
</head>
<body>
    
</body>
</html>
```

Save this file as `CB_image_rollover.htm` (CB means cross-browser). Open this page with any modern browser (IE, Firefox, Opera, or Safari), and you'll see something similar to Figure 12-5.

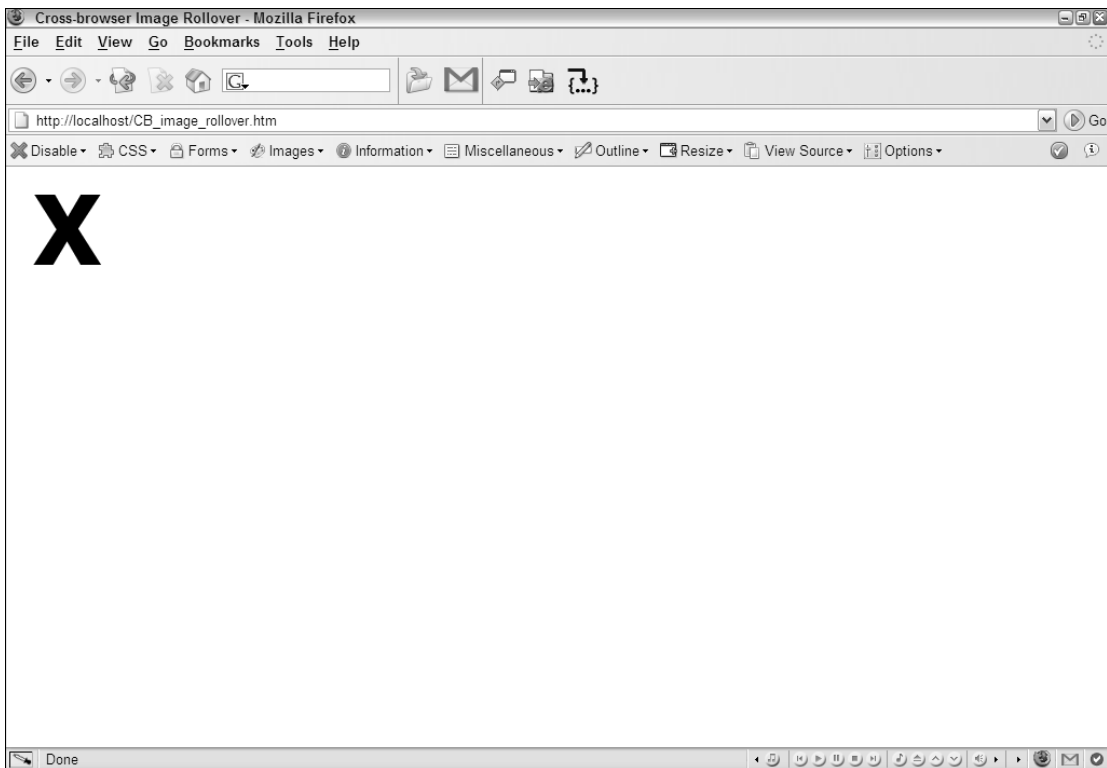


Figure 12-5

Move the mouse pointer over the image, and it changes to an O (see Figure 12-6).

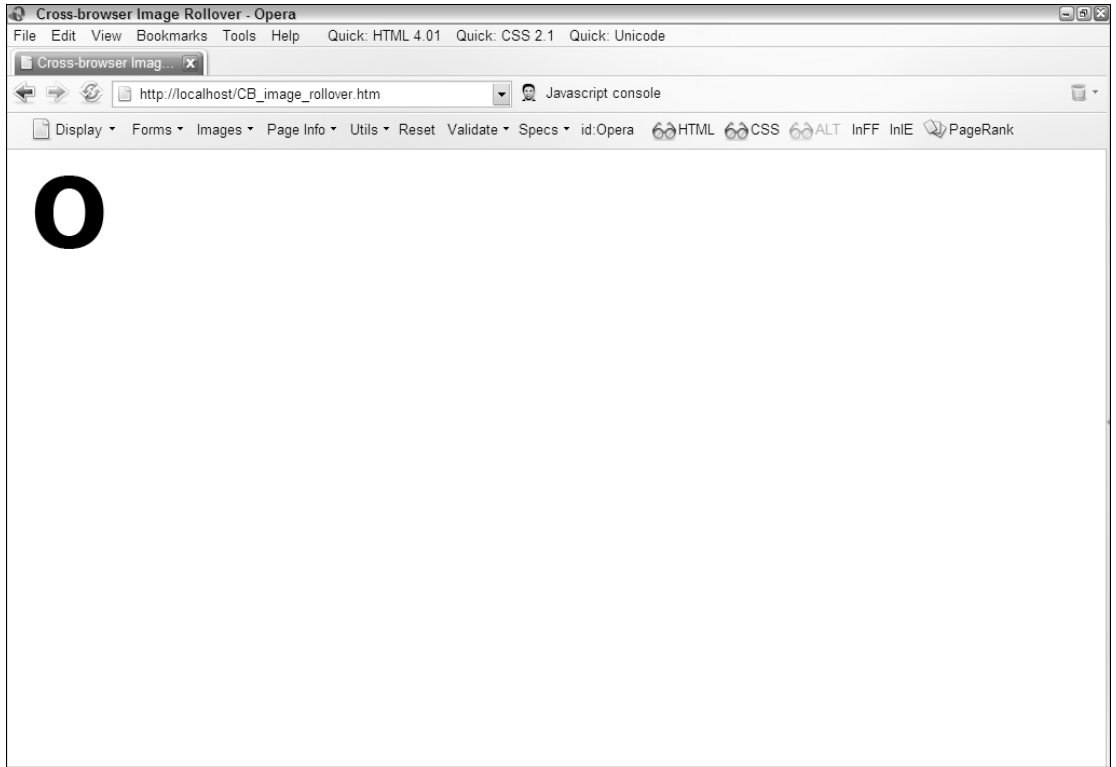


Figure 12-6

How It Works

A single `` element resides in the page's body. The `src` attribute is set to `x.gif`. Also, `onmouseover` and `onmouseout` event handlers are set to the value of `image_eventHandler(event)`.

```

```

This not only suits the needs of non-IE browsers, but also passes the IE-specific global `event` object to the `image_eventHandler()` function. This enables the function to use both event objects, depending on which browser displays the web page.

In the script block, the `image_eventHandler()` function is defined.

```
function image_eventHandler(evt) {  
    var elementTarget;  
  
    //more code here  
}
```

Chapter 12: Introduction to Dynamic HTML

It accepts an event object as an argument, and the first line creates a variable called `elementTarget`. Next, the code branches to accommodate the different browsers:

```
function image_eventHandler(evt) {
    var elementTarget;

    if (evt.srcElement) { //The browser is IE
        elementTarget = evt.srcElement;
    } else { //The browser is non-IE
        elementTarget = evt.target;
    }

    //more code here
}
```

This code consolidates the differing event object implementations into one usable variable: `elementTarget`. To do this, use object detection to determine the browser and assign the appropriate property to the `elementTarget` variable.

Since this function handles two events, it needs to determine which event called the function. As you already know, the `type` property contains this information. First check to see if the event was caused by the mouse pointer moving over the image element:

```
function image_eventHandler(evt) {
    var elementTarget;

    if (evt.srcElement) { //The browser is IE
        elementTarget = evt.srcElement;
    } else { //The browser is non-IE
        elementTarget = evt.target;
    }

    if (evt.type == "mouseover") { //The mouse rolled over the image.
        elementTarget.src = "o.gif"; //So change the image's src property.
    }

    //more code here
}
```

If this is the case, change the `src` property of the `` element to `o.gif`, which changes the picture that is displayed to the user. Notice that this is where the `elementTarget` variable is used.

Next, check to see if the event was caused by the user moving the mouse pointer off of the image:

```
function image_eventHandler(evt) {
    var elementTarget;

    if (evt.srcElement) { //The browser is IE
        elementTarget = evt.srcElement;
    } else { //The browser is non-IE
        elementTarget = evt.target;
    }

    if (evt.type == "mouseover") { //The mouse rolled over the image.
```

```
        elementTarget.src = "o.gif"; //So change the image's src property.
    }

    if (evt.type == "mouseout") { //The mouse moved out.
        elementTarget.src = "x.gif"; //So change it back to the original.
    }
}
```

This code compares the type property to the string `mouseout` and changes the image's `src` property back to `x.gif` if this event was fired. And with these final few lines of code, your first cross-browser DHTML script is complete!

These examples have focused on using the event handler attributes of HTML elements. The next section focuses on using event handlers that are assigned through JavaScript.

Event Handlers as Properties Revisited

There is one glaring difference between assigning event handlers with HTML attributes and assigning them with JavaScript. With the HTML attributes, you were able to pass arguments to the function handling the event (for example, the event object). When using properties, though, you cannot pass any arguments to the function when assigning the event handler:

```
document.images[0].onmouseover = image_eventHandler(event); //This is wrong!

document.images[0].onmouseover = image_eventHandler; //This is correct.
```

This isn't really a problem for IE, as the event object is a part of the window object, which is global. For non-IE browsers, the browser automatically passes the event object to the event handler when the event fires. So handling events in this fashion is *almost* identical to handling them as you did in the previous section.

Try It Out Image Rollover with Property Event Handlers

The following HTML is yet another implementation of the image rollover script:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
    <title>Cross-browser Image Rollover</title>
</head>
<body>
    

    <script type="text/javascript">
        function image_eventHandler(evt) {
            var elementTarget;
            var eventType;

            if (window.event) { //The browser is IE
                elementTarget = window.event.srcElement;
                eventType = window.event.type;
            } else { //The browser is non-IE
```

```
        elementTarget = evt.target;
        eventType = evt.type;
    }

    if (eventType == "mouseover") { //The mouse rolled over the image.
        elementTarget.src = "o.gif"; //So change the image's src property.
    }

    if (eventType == "mouseout") { //The mouse moved out.
        elementTarget.src = "x.gif"; //So change it back to the original.
    }
}

document.images[0].onmouseover = image_eventHandler;
document.images[0].onmouseout = image_eventHandler;
</script>
</body>
</html>
```

Save this file as `CB_image_rollover_property_events.htm`. When loaded into a browser, a picture of an X is visible.

How It Works

In the body of the page, a single `` element is defined. This is a simple element: Only the `src` attribute is used.

```

```

Here's where things start to change. Instead of putting the `<script/>` element in the head of the page, you put it in the body. This is done for one reason: The image needs to be loaded into the document before it can be accessed with JavaScript. Attempting to access an HTML element before it's loaded into the document results in an error.

You can't access an HTML element in JavaScript if it hasn't been loaded by the browser.

The first thing in the script block is the `image_eventHandler()` function. It is essentially the same as in the prior sections, but this iteration contains a few changes. The first is the addition of the `eventType` variable. This variable will contain the value of the `type` property of the event object.

```
function image_eventHandler(evt) {
    var elementTarget;
    var eventType;

    //more code here
}
```

You do this because the function now has to handle two different event objects. The `evt` parameter is used only in non-IE browsers; the following line generates an error if you run it in IE:

```
alert(evt.type); //throws an error in IE
```

This happens because IE does not pass a value to event handlers when the event fires. So when branching the code to assign `elementTarget`, you must also assign the `eventType` variable its value, like this:

```
function image_eventHandler(evt) {
    var elementTarget;
    var eventType;

    if (window.event) { //The browser is IE
        elementTarget = window.event.srcElement;
        eventType = window.event.type;
    }

    //more code here
}
```

This code uses the global `window.event` object to retrieve the event's type and assign it to `eventType`.

Next, get the same information for the non-IE browsers:

```
function image_eventHandler(evt) {
    var elementTarget;
    var eventType;

    if (window.event) { //The browser is IE
        elementTarget = window.event.srcElement;
        eventType = window.event.type;
    } else { //The browser is non-IE
        elementTarget = evt.target;
        eventType = evt.type;
    }

    //more code here
}
```

This code uses the `evt` parameter to retrieve the `target` and `type` properties. At this point you have the information you need; now it's time to use it in changing the image element's `src` property.

```
function image_eventHandler(evt) {
    var elementTarget;
    var eventType;

    if (window.event) { //The browser is IE
        elementTarget = window.event.srcElement;
        eventType = window.event.type;
    } else { //The browser is non-IE
        elementTarget = evt.target;
        eventType = evt.type;
    }

    if (eventType == "mouseover") { //The mouse rolled over the image.
        elementTarget.src = "o.gif"; //So change the image's src property.
    }
}
```

```
    if (eventType == "mouseout") { //The mouse moved out.  
        elementTarget.src = "x.gif"; //So change it back to the original.  
    }  
}
```

This code hasn't changed from the previous examples: It determines what event called the function and changes the `src` property accordingly. Now all that is left is to wire up the events. Do so by using the `onmouseover` and `onmouseout` properties of the image element.

```
document.images[0].onmouseover = image_eventHandler;  
document.images[0].onmouseout = image_eventHandler;
```

This code retrieves the image element by using the `images` collection. There is only one image in the document. Therefore, the index of 0 is used to retrieve the element.

These sections have walked you through many rollover examples. The next section departs from JavaScript and introduces you to Cascading Style Sheets.

CSS: A Primer

Cascading style sheets (CSS) is an important technology that developers use to build many DHTML scripts. Cascading Style Sheets is a language that enables you to apply a set of styles to various HTML tags. For example, you may want all `<h1/>` elements in the web page to be colored blue and underlined. Before CSS you would do this with various formatting tags, such as `` and `<u>`. This approach worked; however, it caused a few problems.

The first problem with this approach is that you need to wrap the content with these formatting tags, and styling different pieces of content the same way requires repeated tag use. This results in extra work for web page authors. Also, changing an existing style (say blue underlined text) to a new style (red boldface text), requires the change of every `` and `<u>` tag throughout the document.

To add to this problem, you may want to set a very specific size for the text of the `<h1>` tags. The sizes offered by the font are very limited, and those few sizes enable you to specify only whether the font should be larger or smaller than normal. The font sizes are relative to the font size the user has set in her browser. This makes it possible for text to be rendered in an unreadable size.

HTML also does not give you control over where elements appear in the page. The browser displays HTML elements according to where the tags appear in the source code. Therefore, the following HTML will always display `Hello` on the first line and `World` on the second line:

```
<p>Hello</p>  
<p>World</p>
```

This is what is referred to as the *document flow* — how the browser naturally displays the web page. Without CSS, it is not possible to change the document flow. You can use formatting tags for different types of positioning (like lists or tables), but you can't specify an exact position.

CSS solves each of these problems. It enables you to specify certain styles that can be used throughout the entire web page easily and efficiently. It also gives you more control over how the web page looks, thereby giving you more power to make the page look the way you first envisioned it.

Adding Some Style to HTML

You can add style to an HTML page by creating style sheets within the `<style>` tag or by specifying `style` attributes in an element's opening tag. There are a number of ways to define style, but this primer covers only three of them:

- ☐ Defining a style for certain HTML elements, for example a style for all paragraph elements.
- ☐ Creating a style class. You then specify in the class attribute of a tag the name of the style class to be applied.
- ☐ Specifying style for just one element.

The `<style>` Tag

You can use the first two ways of adding style mentioned in the previous list—defining a style for a type of tag and defining a style class—by creating a `<style/>` element inside the `<head/>` element.

To define the style for a particular element, use the following format:

```
elementName {  
    style-property-name: style property value;  
    another-style-property: another value;  
}
```

This CSS code is referred to as a CSS *rule*. A rule consists first of a *selector*. A selector selects the HTML element to which the style rule will be applied. In the previous example, `elementName` is the selector. Any HTML element name can be used as a selector; just remember to take out the angle brackets (`<` and `>`). For example, to define a selector for all `<p>` tags, you would use an `elementName` of `p`.

After the selector comes a set of *style declarations* inside the curly braces. A declaration consists of a property name, a colon (instead of an equals sign), and then the property's value. Style declarations are separated by semicolons. For example, let's say you want all `<p>` tags to be blue, in the Arial font, and 10 points in size. First create a rule to select the `<p>` tags. Then add the declarations by specifying the `font-family`, `font-size`, and `color` properties. The following code does this:

```
<html>  
<head>  
    <style>  
        p {  
            font-family: arial;  
            font-size: 10pt;  
            color: blue;  
        }  
    </style>  
</head>
```

```
<body>
  <p>Some blue arial 10 point text</p>
  <p>Also blue arial 10 point text</p>
</body>
</html>
```

The first property specified is `font-family`, which is assigned the value `arial`. This is followed by a semicolon to mark the end of that style declaration. Next the `font-size` property is defined and its value set to `10pt`. Finally you specify the `color` property and its value of `blue`. Both the paragraphs defined in the page will have the same font, font size, and color.

If you want to define similar properties for the `<td>` tags, add a rule inside the `<style>` tag.

```
<style>
  p {
    font-family: arial;
    font-size: 10pt;
    color: blue;
  }

  td {
    font-family: arial;
    font-size: 12pt;
    color: red;
  }
</style>
```

This CSS sets the `<td>` tags to have the same font family as the `<p>` tags, but their font's size is larger and their text is rendered in red. This is the basic behavior of CSS, but it doesn't help if you want some paragraphs in a larger font while others remain in the 10-point font.

CSS is not limited to this basic functionality, however. You can define what are called *classes*: CSS rules that can be applied to a variety of elements. In fact, with CSS classes, you can make any element emulate almost any other.

A class's selector is a custom name that you define. Classes are distinguished by a dot in front of their names. The following code defines a class called `heading1`:

```
<style>
  p {
    font-family: arial;
    font-size: 10pt;
    color: blue;
  }

  .heading1 {
    font-size: 24pt;
    color: orange;
  }
</style>
```


The `heading1` class's style declarations specify that the `font-size` should be 24pt and that the `color` should be orange. Applying this class to an HTML element requires the use of the `class` attribute. All elements within the `<body/>` element can be classed. The following code modifies the previous HTML example. The first of the two paragraphs is classed as `heading1`.

```
<html>
<head>
  <style>
    p {
      font-family: arial;
      font-size: 10pt;
      color: blue;
    }

    .heading1 {
      font-size: 24pt;
      color: orange;
    }
  </style>
</head>
<body>
  <p class="heading1">A Heading in orange arial 24 point text</p>
  <p>Some blue arial 10 point text</p>
</body>
</html>
```

Now the first paragraph has the `heading1` class of style applied. Therefore, it is rendered as orange 24-point text in the Arial font. The `heading1` class does not declare a font face property; it inherits the font face from the `p` rule. The style defined in the `p` rule is applied to all `<p/>` elements. So the first `<p/>` element has both the `p` rule and the `heading1` rule applied to it. If the same style declarations are defined in both rules, the declarations in the class rule take precedence. Therefore the first paragraph has the Arial font applied from the `<p/>` definition, but the `size` and `color` style declarations of the `heading1` class override those defined in the `p` rule.

The style Attribute

The third way of defining styles uses the `style` attribute, which most HTML elements support. For example, if you wanted just one paragraph to be in green italics you could change the HTML tag definition like this:

```
<p style="font-style: italic; color: green">
  Some green arial 10 point text
</p>
```

Define the style declarations inside the `style` attribute of the tag, just as you did previously when it was inside the `<style>` tag. First set the `font-style` attribute to `italic`; then, after a semicolon, set the `color` to green. As before, the setting of the `color` property here overrides the setting in the style sheet definition for all `<p/>` elements. However, the paragraph will still use the `font-family` and `font-size` properties set in the `<style/>` tag.

Cascading Style Sheets

The style declarations defined for one element may cascade down to the elements contained within that element; hence the name Cascading Style Sheets. In other words, some style declarations of a parent element cascade down to the child elements inside. For example, in the following code no rules are defined for the two paragraph elements. Instead a rule for `<div/>` elements is created, and one such element contains a sole `<p/>` element. The paragraph outside the `<div/>` has no style applied to it, yet the `<p/>` inside the `<div/>` inherits the style set for the `<div/>` tag.

```
<html>
<head>
  <style>
    div {
      font-family: arial;
      font-size: 10pt;
      color: blue;
    }
  </style>
</head>
<body>
  <p>This text has no style.</p>
  <div>
    <p>This text inherits style from its parent div element.</p>
  </div>
</body>
</html>
```

Styling Properties

In the previous sections you learned how to select elements in the HTML page to apply styles to, but you didn't cover much in the way of actual style properties. After all, what use is a style sheet where hardly any style is declared?

Properties are essentially broken up into distinct groups. Although it is outside the scope of this primer to cover all properties, there are quite a few to discuss.

Colors and Background

Probably the simplest styles you can apply to a web page are colors. You can change the color of text (foreground color), as well as the color of the background. You are not limited to using a color for your background; you may also choose to use an image.

Foreground Color

Changing the foreground color is quite simple, and you have seen examples of it in this chapter. The CSS property that controls this color is the `color` property, and you have a variety of ways to assign its value. This little section will show you the many ways you can assign the color red to your text.

The first method of assigning color is the use of a color keyword.

```
color: red;
```

Using this approach ensures that your CSS is quite readable; however, you are limiting yourself when using keywords. The CSS specification defines 17 color keywords that every browser should implement: aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive, orange, purple, red, silver, teal, white, and yellow. Some browsers implement many more colors, while others do not. You can easily avoid any problems caused by this discrepancy by using either the previously listed names or one of the other color values.

The second approach enables you to assign the specific red, green, and blue values of a color by using `rgb()` notation. This notation allows two types of values: integer values that range from 0–255, and percentage values from 0%–100%. You assign a value for red, green, and blue.

```
color: rgb(255, 0, 0);  
color: rgb(100%, 0%, 0%);
```

Whether you use plain integers or percentages, the result of the preceding CSS will be the same.

Lastly, you can use hexadecimal values to assign colors. In order to do this, you must prepend the hexadecimal number with a pound sign (#). Hexadecimal numbers in CSS typically consist of three pairs of numbers.

```
color: #FF0000;
```

The first pair is red (FF), the second is green (00), and the third is blue (00). You can also use three digits, like this:

```
color: #f00;
```

When the browser finds three-digit hexadecimal values, it converts them to a six-digit form and replicates the digits. So the previous three-digit value becomes the six-digit value you just looked at.

The concept of using colors does not end with foreground color. These same principles are used in background color (and anywhere you can use color) as well.

Backgrounds and Color

In order to assign a color to an element's background, you must use the `background-color` property:

```
background-color: gray;  
color: red;
```

The first line sets the background color of the element to gray, and the text assumes the color red. As mentioned earlier, the preceding color principles apply to background colors as well. You can use color names, hexadecimal numbers, or `rgb()` notation to assign a background color.

Sometimes all you want is a little color, but other times you want to spruce up your web site with more—like an image. Through CSS, it's possible to enable an element to display an image in its background. To do this, you use the `background-image` property:

```
background-image: url(myImage.gif);
```

Chapter 12: Introduction to Dynamic HTML

The first thing you may notice is the use of the `url()` notation. This specific example shows the use of a relative URI, meaning that `myImage.gif` is located in the same directory as this style sheet. You can also use an absolute URI like the following:

```
background-image: url(http://www.yoursite.com/myImage.gif);
```

Both ways are correct. Remember that you can use both `background-color` and `background-image`; you're not limited to using just one or the other.

It is also possible to use single quotes (') or double quotes(") in url() notation like this:

background-image: url("myImage.gif");. However, this does not work in some Mac browsers. For compatibility reasons, it is best to leave quotes out.

Fonts and Text

Text is the most important aspect of any web site. Ninety-nine percent of the time, a visitor comes to read what you have to say. Therefore, it's important to learn how to style your text. You've already learned how to set the color of text; now take a look at how you can set the type of font, as well as its size, weight, and decoration.

The `font-family` property, as its name suggests, sets the font for the text. You have a variety of fonts installed on your computer at your disposal; you can also use generic fonts in the event that a viewer's computer lacks the font (or fonts) you specify:

```
font-family: arial, verdana, sans-serif;
```

In this CSS, the `font-family` property is set to three values: `arial`, `verdana`, and `sans-serif`. If the viewer's computer does not have the Arial font installed, the browser attempts to use Verdana. If neither Arial nor Verdana is installed, the browser uses the generic `sans-serif` font. Using a generic font, even if it is the "last resort" font, is encouraged, as it tells the browser what type of font to use if none of your preferred fonts are available. Following is the list of generic fonts:

- ☐ serif
- ☐ sans-serif
- ☐ cursive
- ☐ fantasy
- ☐ monospace

There's no limit to the amount of fonts you can assign to the `font-family` property. Just know that the browser will attempt to use each font in the order you specify.

Now that you've set a font family, you should also set a size. The CSS property you use to do this is aptly named `font-size`. This property accepts a variety of units for values, and you've already seen the use of points as a measurement. Another popular unit for font size is pixels. When you specify a font size in pixels, the browser renders the text using pixels as the font measurement. For example, see the following CSS:

```
font-size: 12px;
```

When the browser renders text with this style, the font is set to 12 pixels in height. There is also no limit to the size you can use; however, you'll want to use something that is easy to read.

When you want to emphasize a certain part of the text, you have several options. First, you can bold your text with the `font-weight` property:

```
font-weight: bold;
```

This will make the text bold and allow the user to see that it is important in some way. You can also italicize your text by using the `font-style` property:

```
font-style: italic;
```

This italicizes the text. To make life a bit easier, the CSS specification gives you a quick way to set your font preferences by assigning these values to the `font` property. The `font` property's syntax looks like this:

```
font: font-style font-weight font-size font-family;
```

For example, if you wanted to set the font contained in an element to be 12-point bold text and to use the Verdana font face, your CSS would look like this:

```
font: bold 12pt verdana;
```

You don't have to specify each value of each property, only the ones you desire. However, it is important to keep them in the order listed earlier. Take the following example:

```
font: bold italic verdana 12pt; /* Wrong Format */
```

This CSS is not in the correct format, because `font-style` should come before `font-weight`, and `font-size` should come before `font-family`. Some browsers may display the text with the desired style; however, other browsers may not — so it is best to keep to the standard order.

Lastly, you can underline text by using the `text-decoration` property:

```
text-decoration: underline;
```

This property is not related to the font properties discussed earlier. Therefore, you cannot add underline to the `font` property.

Links, by default, are underlined. You can change this by setting `text-decoration` to `none`.

The presentation of your text plays an important role in the way users perceive your web site. Make sure that it is easy to read: the proper size, color, and emphasis will make your visitors happy.

Showing and Hiding Elements

All elements are shown by default; however, you may wish to hide (or show) content based upon an action the user takes. For example, you can show an element that contains information on a specific link when the user moves her mouse over the link, or you can cause an element to pop up in order to display information about a certain form element that the user is currently typing in.

Chapter 12: Introduction to Dynamic HTML

But how do you show and hide elements? With the `visibility` property:

```
visibility: hidden;
```

Setting this property to `hidden` hides the element. Using this property, however, has a side effect. Hiding an element does not remove it from the document's flow; instead you see the area in which the element is located, yet the element's contents are hidden. In other words, you end up with a blank area when an element is hidden.

To show a hidden element, set the `visibility` property to `visible`:

```
visibility: visible;
```

The element's contents (and the element itself) are visible once again to the user when you use this value.

Hiding an element with the `visibility` property causes a blank space to appear where the element used to be seen. This can sometimes cause an undesired result. To solve this issue, CSS also provides another way of hiding an element. By using the `display` property, you can remove an element completely from the document's flow, showing no evidence that the element was there. To do this, set the `display` property to `none`, like this:

```
display: none;
```

The `display` property can accept other values, which cause the element to be displayed again. For simplicity's sake, we'll cover two here. The first is the value `block`. Setting the `display` property to `block` tells the browser to render the element as a block, much as it displays paragraphs.

```
display: block;
```

The second value is `inline`. This property displays the element not as a block, but as a line. Many elements have an inline display by default, like `` elements and `<a/>` elements.

```
display: inline;
```

It is important to note that the `visibility` and `display` properties are two different properties and should be treated as such. The `display` property has the ability to remove an element visually from the HTML page, and the `visibility` property simply hides or shows an element.

It's All About Position

HTML content normally flows from the top to the bottom of a page, the position of the output being based on where the tag is defined in the page. Consider the following HTML:

```
<P>Paragraph 1</P>  
<P>Paragraph 2</P>
```

When the browser renders this HTML, the paragraphs appear one after the other.

```
Paragraph 1  
Paragraph 2
```

However, it is possible to position content with style sheets. You can do it in many ways, but this primer only covers two. With *absolute positioning* the positioned element is removed from the document's flow and can be placed anywhere in the browser's viewport. With *relative positioning* the element is calculated according to the normal flow, but it can be shifted relative to its original position. You can specify whether an element is positioned relatively or absolutely by using the `position` style attribute, which takes the values `relative` and `absolute`.

The keys to positioning an element are the CSS properties `left` and `top`. You can use a number of different units for these values, but for simplicity's sake we'll stick here with those most commonly used: pixels and percentages. Let's start with pixels.

When you set your computer screen's display resolution, you can choose values like 800×600 or 1024×768 , and so on. These values are actually pixels, so 800×600 means that your screen will be treated as being 800 pixels wide by 600 pixels high. These values are independent of your actual physical monitor size; they determine the maximum sizes of the browser window. If the user has an 800-by-600 screen and you set a `<div/>` element to appear 1,000 pixels from the top, the element will be offscreen. You need to also keep in mind that the user can resize the browser window and that screen resolution is a theoretical maximum. Also, the browser toolbars and frame take up some space.

As far as absolute positioning is concerned, the top left-hand corner of the browser window is 0, 0 and the bottom right-hand corner will be at the upper limit of the values that determine screen resolution. So for 800×600 resolutions, the bottom right-hand corner is at the coordinates 800, 600, whereas for 1024×768 resolutions, it's 1024, 768, as shown in Figure 12-7.

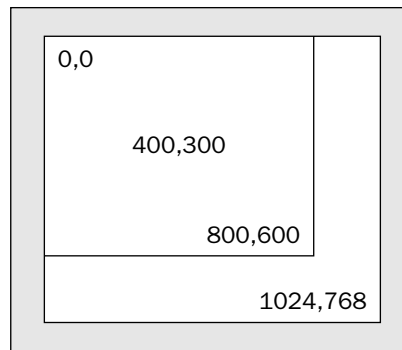


Figure 12-7

If you specified that a `<div/>` element's `left` style attribute be set to 400 pixels and its `top` style attribute to 300 pixels, the element's top left-hand corner would be near the middle of a browser window on an 800×600 resolution display. If the display were set to 1024×768 , the top left-hand corner would be about two-fifths across the browser window by about two-fifths down. This position has been marked approximately on the diagram in Figure 12-7.

Let's look at an example. Imagine you have an absolutely positioned `<div/>` element at a position `left` of 200 and `top` of 100, and with other style attributes specifying it should be 200 pixels wide by 200 pixels deep and have a background color of `blue`. Your browser window would look something like what is shown in Figure 12-8.

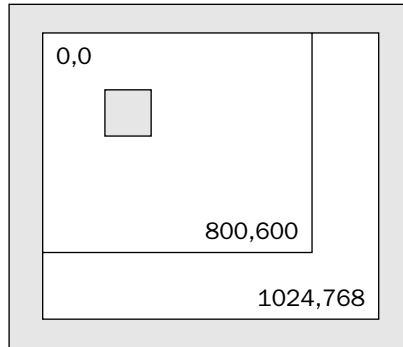


Figure 12-8

Now how can you position a paragraph contained within the `<div/>` element relative to that `<div/>` element? If you think of the `<div/>` element as a sort of screen within a screen, in this case a screen of resolution 200 * 200, relative positioning is easier to understand.

If you think of the `<div/>` element as a browser window itself, the top left-hand corner of the `<div/>` element is 0, 0 and the bottom right-hand corner is 200, 200 — that is, the `<div/>` element's width and height — in this case. If you then specify that the `<p/>` element should be at the relative position `left` of 100 and `top` of 100, that would leave the top left-hand corner of the `<p/>` element in exactly the middle of the `<div/>` element in which it is contained.

What if you relatively position a tag that is not inside another tag? Well, in that case it'll be relative to the `<body/>` element, which is the whole page itself. Essentially this is the same as absolute positioning if the whole page fills the screen.

Let's see the HTML for a page with the `<div/>` and `<p/>` elements at the positions we're discussing.

```
<html>
<head>
  <style>
    .divStyle1 {
      background-color: blue;
      position : absolute;
      width: 200px; height: 200px;
    }

    .pStyle1 {
      color: white;
      position : relative;
    }
  </style>
</head>
<body>
  <div style="left: 200px; top: 200px" class="divStyle1">
    <p class="pStyle1" style="left: 100px; top: 100px">My Paragraph</p>
  </div>
</body>
</html>
```


Type this into a text editor and save it as `relativepos.htm`.

In the `<style/>` element, you create two style classes: `divStyle1` and `pStyle1`. Within these, you specify the style declarations previously discussed. In `divStyle1` you specify a blue background, that the positioning should be absolute, and the width and height in pixels. `pStyle1` specifies only the color of the text and that it should be positioned relatively.

Then, in the `<div/>` element, the CSS class `divStyle1` is assigned by using the `class` attribute, and the `style` attribute specifies the `left` and `top` positions of the tag. Because the `divStyle1` class's specified positioning should be absolute, the values `200` and `200` will be absolute screen values.

In the `<p/>` element, you apply the `pStyle1` class and then specify the position of the element as `100px` and `100px`. This time the element is placed relative to its position in the document flow, with `0, 0` as the top left-hand corner of the `<div/>` and `200, 200` as the bottom right-hand corner.

Most modern browsers require you to specify the measuring unit when assigning values for `top`, `left`, `height`, and `width`. As with the `font-size` property we looked at earlier, you simply add `px` to the end of the numerical value.

On a screen with a resolution of `800 * 600` and the browser window maximized, the example should look like what is shown in Figure 12-9.

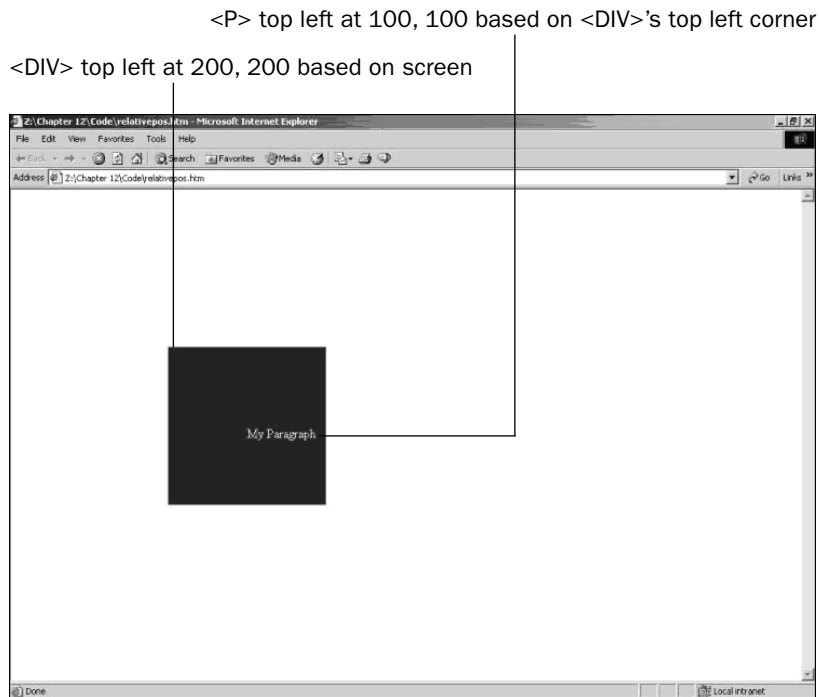


Figure 12-9

Remember that the `left` and `top` style properties apply to the left and top of the element, respectively, putting the top left of the letter `M` at position `100, 100` within the `<div/>` element.

Chapter 12: Introduction to Dynamic HTML

In addition to specifying position properties as pixels, you can use percentages. To do this you simply put % at the end of the value instead of px. For example, to position the `<div/>` and `<p/>` elements so that the top left of the `<div/>` appears in the middle of the browser window and the top left of the `<p/>` element appears in the middle of the `<div/>`, regardless of screen resolution, you'd write the following code:

```
<div style="left: 50%; top: 50%" class="divStyle1">
  <p class="pStyle1" style="left: 50%;top: 50%">My Paragraph</p>
</div>
```

Dynamic HTML

You might be wondering why you have this introduction to style sheets in a book about JavaScript. Remember, DHTML is the manipulation of an HTML document after it is loaded into the browser, and the most common way to manipulate the document is by changing the way HTML elements look. In order to do this, you need to go over a few topics. First, you'll take a look at how to find an element in the document. Second, you'll look at how you can change an element's style programmatically.

Accessing Elements

The Document Object Model (DOM) holds the ability you need to find and access HTML elements; the DOM is a hierarchical tree, and you can certainly climb it, inspect every branch and leaf, and find what you're looking for. However, the DOM provides a much easier way to find specific elements.

The DOM exposes a method called `getElementById()`, which is used to find, in a web page, specific HTML elements whose `id` attributes match that of the argument passed to the method. Consider the following HTML as an example:

```
<div id="divAdvert">Here is an advertisement</div>
```

This HTML creates a `<div/>` element, and the `id` attribute is assigned `divAdvert`. With this information, you can easily retrieve this element by using `getElementById()`.

```
var divAdvert = document.getElementById("divAdvert");
```

This JavaScript code retrieves the `<div/>` element created earlier, and you can use the `divAdvert` variable to programmatically manipulate it. There are a variety of ways to change HTML elements; the most common, though, are changing the way an element looks and changing an element's position.

Changing Appearances

Probably the most common use for DHTML is to change the way an element looks. Such a change can create an interactive experience for visitors to your web site, and can even be used to alert them to important information or that an action is required by them. Changing the way an element looks consists almost exclusively of changing CSS properties for an HTML element. You can do this two ways through JavaScript: You can change each CSS property, or you can change the value of the element's `class` attribute.

Using the style Property

In order to change specific CSS properties, you must look, once again, to the DOM. All modern browsers implement the `style` object, which maps directly to the element's `style` attribute. This object contains CSS properties, and by using it you can change any CSS property that the browser supports. Use the `style` property like this:

```
oHtmlElement.style.cssProperty = value;
```

The CSS property names generally match those used in a CSS file; therefore, changing the text color of an element requires the use of the `color` property, like this:

```
var divAdvert = document.getElementById("divAdvert"); //Get the desired element

divAdvert.style.color = "blue"; //Change the text color to blue
```

There are some cases, however, in which the property name is a little different from the one seen in a CSS file. CSS properties that contain a hyphen (-) are a perfect example of this exception. In the case of these properties, you remove the hyphen and capitalize the first letter of the word that follows the hyphen. The following code shows the incorrect and correct ways to do this:

```
divAdvert.style.background-color = "gray"; //Wrong

divAdvert.style.backgroundColor = "gray"; //Correct
```

You can also use the `style` object to retrieve styles that have previously been declared. However, if the `style` property you try to retrieve has not been set with the `style` attribute (inline styles) or with the `style` object, you will not retrieve the property's value. Consider the following style sheet which sets the style for the element with `divAdvert` as its `id`:

```
<style>
  #divAdvert {
    background-color: gray;
  }
</style>
```

First, a new concept is present in this CSS. Look at the selector, and how it is preceded by a pound sign (#). In CSS, this symbol is an ID selector, and it selects an HTML element whose ID attribute matches the selector name (without the pound sign). Therefore, this style rule is applied to the `<div>` element defined earlier in the section. In this CSS, you set the element to have a background color of `gray`. Also, the HTML element has changed to include the use of the `style` attribute:

```
<div id="divAdvert" style="color: green">I am an advertisement.</div>
```

When the browser renders this element, it will have green text on a gray background. If you had used the `style` object to retrieve the value of both the `background-color` and `color` properties, you'd get mixed results:

```
var divAdvert = document.getElementById("divAdvert"); //Get the desired element

alert(divAdvert.style.backgroundColor); //Alerts an empty string

alert(divAdvert.style.color); //Alerts green
```

Chapter 12: Introduction to Dynamic HTML

You get these results because the `style` object accesses the `style` attribute of the element. If the style declaration is set in the `<style/>` block, you cannot retrieve that property's value with the `style` object.

Try It Out Using the style Object

Let's look at a simple example of changing the appearance of some text by using the `style` object.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
    <title>Using the style Object</title>
    <style type="text/css">
        #divAdvert {
            font: 12pt arial;
        }
    </style>
    <script type="text/javascript">
function divAdvert_onMouseOver() {
    //Get the element
    var divAdvert = document.getElementById("divAdvert");

    //Italicize the text
    divAdvert.style.fontStyle = "italic";

    //Underline the text
    divAdvert.style.textDecoration = "underline";
}

function divAdvert_onMouseOut() {
    //Get the element
    var divAdvert = document.getElementById("divAdvert");

    //Set the font-style to normal
    divAdvert.style.fontStyle = "normal";

    //Remove the underline
    divAdvert.style.textDecoration = "none";
}
    </script>
</head>
<body>
    <div id="divAdvert" onmouseover="divAdvert_onMouseOver()"
        onmouseout="divAdvert_onMouseOut()">Here is an advertisement.</div>
</body>
</html>
```

Save this as `style_object.htm`. When you run this in your browser you should see a single line of text, as shown in Figure 12-10.

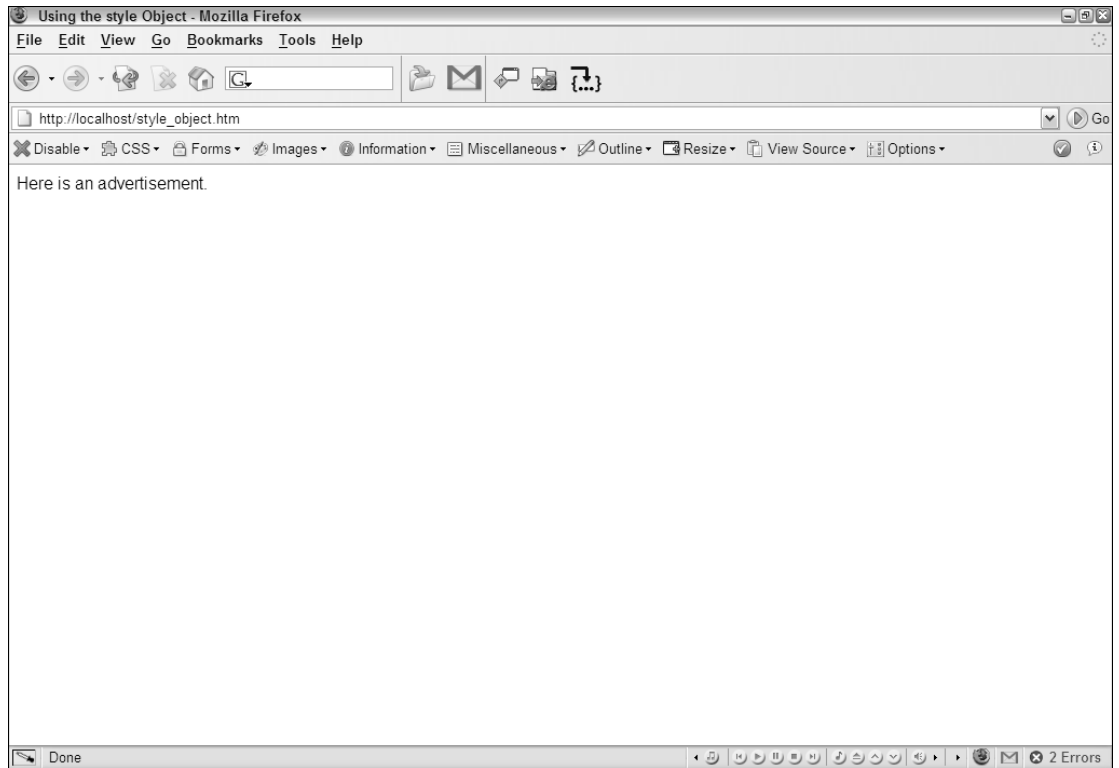


Figure 12-10

Roll your mouse over the text, and you'll see it become italicized and underlined (see Figure 12-11).

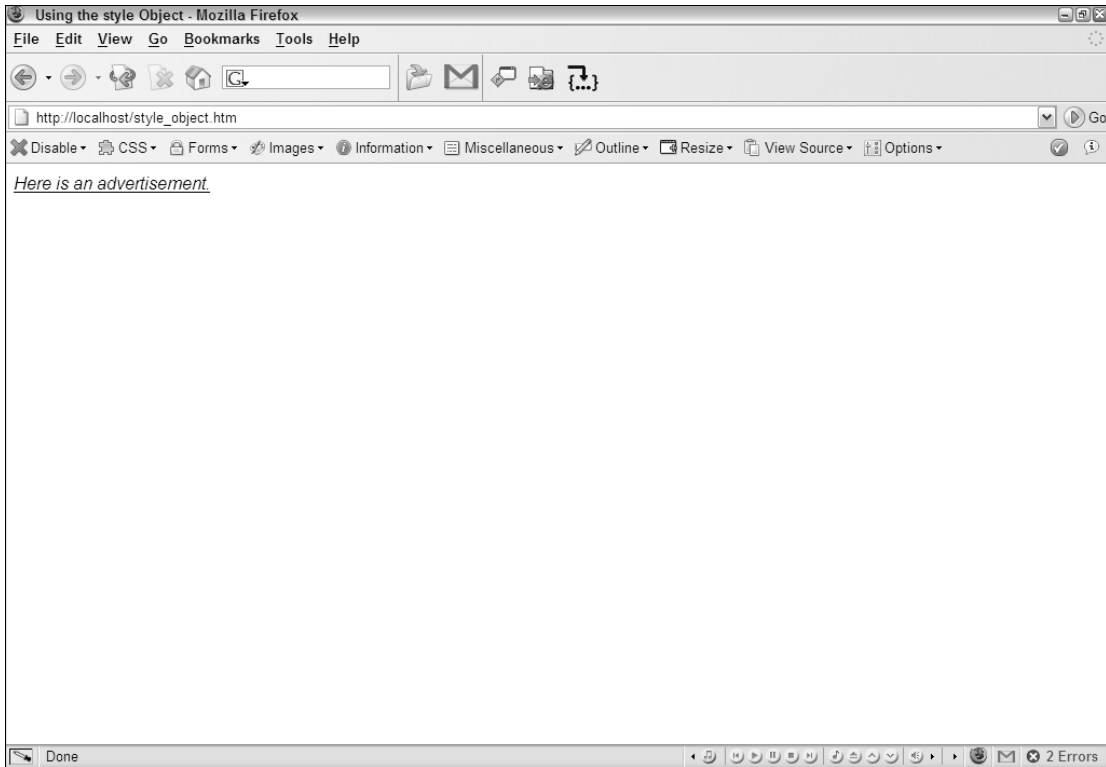


Figure 12-11

And when you move your mouse off of the text, it returns back to normal.

How It Works

In the page's body, a `<div/>` element is created and has an id of `divAdvert`. Hook up the `mouseover` and `mouseout` events to the `divAdvert_onMouseOver()` and `divAdvert_onMouseOut()` functions, respectively, which are defined in the `<script/>` block in the head of the page.

When the mouse pointer enters the `<div/>` element, the `divAdvert_onMouseOver()` function is called. The following shows the function declaration and the first line:

```
function divAdvert_onMouseOver() {  
    //Get the element  
    var divAdvert = document.getElementById("divAdvert");  
  
    //more code here  
}
```

Before you can do anything to the `<div/>` element, you must first retrieve it. You do this simply by using the `getElementById()` method. Now that you have the element, you can manipulate its style. First, make the text italic with the `fontStyle` property:

```
function divAdvert_onMouseOver() {  
    //Get the element  
    var divAdvert = document.getElementById("divAdvert");  
  
    //Italicize the text  
    divAdvert.style.fontStyle = "italic";  
  
    //more code here  
}
```

Next, underline the text by using the `textDecoration` property:

```
function divAdvert_onMouseOver() {  
    //Get the element  
    var divAdvert = document.getElementById("divAdvert");  
  
    //Italicize the text  
    divAdvert.style.fontStyle = "italic";  
  
    //Underline the text  
    divAdvert.style.textDecoration = "underline";  
}
```

Because you set the `textDecoration` property to `underline`, the text displayed in the `<div/>` element becomes underlined.

Naturally, you do not want to keep the text italicized and underlined; so use the `mouseout` event to change the text back to its original state. When this event fires, the `divAdvert_onMouseOut()` function is called.

```
function divAdvert_onMouseOut() {  
    //Get the element  
    var divAdvert = document.getElementById("divAdvert");  
  
    //Set the font-style to normal  
    divAdvert.style.fontStyle = "normal";  
  
    //Remove the underline  
    divAdvert.style.textDecoration = "none";  
}
```

The code for this function somewhat resembles that for the `divAdvert_onMouseOver()` function. First, you retrieve the `divAdvert` element; next, set the `fontStyle` property to `normal`, thus removing the italics. Lastly, set the `textDecoration` to `none`, which removes the underline from the text.

Changing the class Attribute

As you learned earlier, you can assign a CSS class to elements by using the element's `class` attribute. This attribute is exposed in the DOM by the `className` property and can be changed through JavaScript to associate a different style rule with the element.

```
oHtmlElement.className = sNewClassName;
```

Chapter 12: Introduction to Dynamic HTML

Using the `className` property to change an element's style is advantageous in two ways. First, it reduces the amount of JavaScript you have to write, which no one is likely to complain about. Second, it keeps style information out of the JavaScript file and puts it into the CSS file where it belongs. Making any type of changes to the style rules is easier because you do not have to have several files open in order to change them.

Try It Out Using the `className` Property

Let's revisit the code from `style_object.htm` from the previous section, but with a few revisions.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
  <title>Using the className Property</title>
  <style type="text/css">
    .defaultStyle {
      font: normal 12pt arial;
      text-decoration: none;
    }

    .newStyle {
      font: italic 12pt arial;
      text-decoration: underline;
    }
  </style>
  <script type="text/javascript">
    function divAdvert_onMouseOver() {
      //Get the element
      var divAdvert = document.getElementById("divAdvert");

      //Change the className
      divAdvert.className = "newStyle";
    }

    function divAdvert_onMouseOut() {
      //Get the element
      var divAdvert = document.getElementById("divAdvert");

      //Change the className
      divAdvert.className = "defaultStyle";
    }
  </script>
</head>
<body>
  <div id="divAdvert" class="defaultStyle" onmouseover="divAdvert_onMouseOver()"
    onmouseout="divAdvert_onMouseOut()">Here is an advertisement.</div>
</body>
</html>
```


Save this file as `className_property.htm`. This page behaves in the exact same manner as `style_object.htm`. When you place your mouse pointer over the text, the text becomes italicized and underlined; when you move your pointer off of the text, it changes back to normal.

How It Works

There are a few key differences between this HTML page and the one created using the `style` object. For starters, the `#divAdvert` style rule is removed and replaced with two CSS classes:

```
.defaultStyle {
    font: normal 12pt arial;
    text-decoration: none;
}

.newStyle {
    font: italic 12pt arial;
    text-decoration: underline;
}
```

The first class, called `defaultStyle`, is the rule first applied to the `<div/>` element. It declares a normal 12-point Arial font with no underlining. Next, another class called `newStyle` is created. This new rule contains style declarations to specify 12-point italic Arial that is underlined. With these changes, the `<div/>` element definition is changed to use the `defaultStyle` CSS class:

```
<div id="divAdvert" class="defaultStyle" onmouseover="divAdvert_onMouseOver()"
    onmouseout="divAdvert_onMouseOut()">Here is an advertisement.</div>
```

Notice that the `id` attribute is the same: JavaScript still needs to access the element in order to change its `className` property. The `onmouseover` and `onmouseout` event handlers remain the same, as you need the same functionality that `style_object.htm` has.

The last change is in the JavaScript itself. When the `mouseover` event fires on the element, the associated `divAdvert_onMouseOver()` function is called. This function consists of two lines of code as opposed to the three lines you used for the `style` object.

```
function divAdvert_onMouseOver() {
    //Get the element
    var divAdvert = document.getElementById("divAdvert");

    //Change the className
    divAdvert.className = "newStyle";
}
```

The first statement retrieves the `<div/>` element by using the `getElementById()` method. The function goes on to change the `className` property to the value `newStyle`. With this line, the `divAdvert` element takes on a new style rule and the browser changes the way it looks.

When you move your mouse pointer off of the text, the `mouseout` event fires and `divAdvert_onMouseOut()` executes. This function is almost identical to `divAdvert_onMouseOver()`, except that the `className` is set back to its original value:

```
function divAdvert_onMouseOut() {  
    //Get the element  
    var divAdvert = document.getElementById("divAdvert");  
  
    //Change the className  
    divAdvert.className = "defaultStyle";  
}
```

By setting `className` back to `defaultStyle`, the browser displays the `<div/>` element as it previously did, with no italics or underlining.

Positioning and Moving Content

Changing the appearance of an element is an important pattern in DHTML, and it finds its place in many DHTML scripts. However, there is more to DHTML than just changing the way content appears on the page; you can also change the position of an element with JavaScript.

Earlier in the chapter you learned about the `position` CSS property, where you can specify an element to be absolutely or relatively positioned. You also learned about the `top` and `left` properties, which enable you to position the elements where you desire. You can do the same thing with JavaScript by changing the values of these properties.

Just Move It Over There

Moving content with JavaScript is just as easy as using the `style` object. You use the `position` property to change the type of position desired, and by using the `left` and `top` properties, you can position the element.

```
var divAdvert = document.getElementById("divAdvert");  
  
divAdvert.style.left = "100px"; //Set the left position  
divAdvert.style.top = "100px"; //Set the right position
```

This code first retrieves the `divAdvert` element. Then it sets the element 100 pixels from the left and top edges. Notice the addition of `px` to the value assigned to the positions. Many browsers require you to specify a unit when assigning a positional value; otherwise, the browser will not position the element.

Try It Out Moving an Element Around

Moving an element around on the page, as you've seen, is quite similar to changing other styles with the `style` object. However, the ability to move an element on the page is one that is used quite often, and you will definitely see it later in the chapter. Therefore, you are going to build a page that enables you to specify the location of an element through form fields.

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Positioning</title>
    <style type="text/css">
        #divBox {
            position: absolute;
            background-color: silver;
            width: 150px;
            height: 150px;
        }

        input {
            width: 100px;
        }
    </style>
    <script type="text/javascript">
        function moveBox() {
            var divBox = document.getElementById("divBox");
            var inputLeft = document.getElementById("inputLeft");
            var inputTop = document.getElementById("inputTop");

            divBox.style.left = parseInt(inputLeft.value) + "px";
            divBox.style.top = parseInt(inputTop.value) + "px";
        }
    </script>
</head>
<body>
    <div id="divBox">
        <form id="formBoxController" onsubmit="moveBox(); return false;">
            <p>Left: <input type="text" id="inputLeft" /></p>
            <p>Top: <input type="text" id="inputTop" /></p>
            <p><input type="submit" value="Move The Box" /></p>
        </form>
    </div>
</body>
</html>

```

Save this file as `positioning.htm`. When you load the page into your browser, you should see a silver box in the upper left-hand corner of the viewport. Inside this box you'll see a form with two fields and a button, as shown in Figure 12-12.

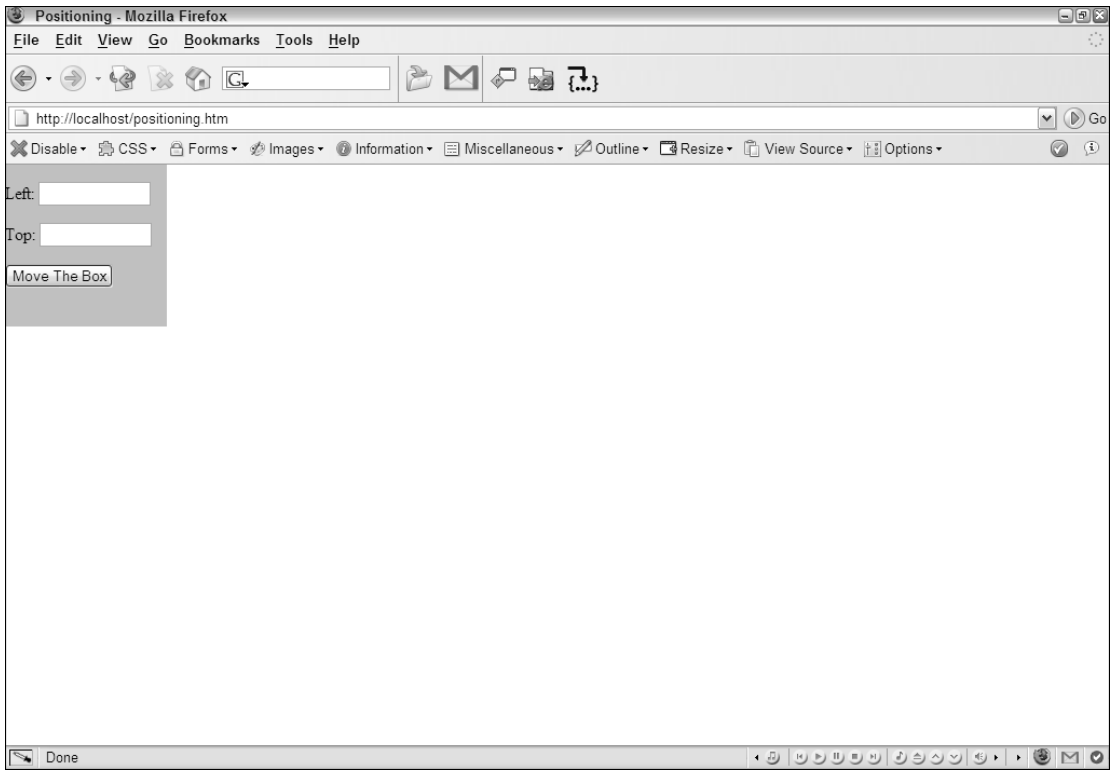


Figure 12-12

When you enter numerical values in the text fields and press the button, the box will move to the coordinates you specified. Figure 12-13 shows the box moved to 100,100.

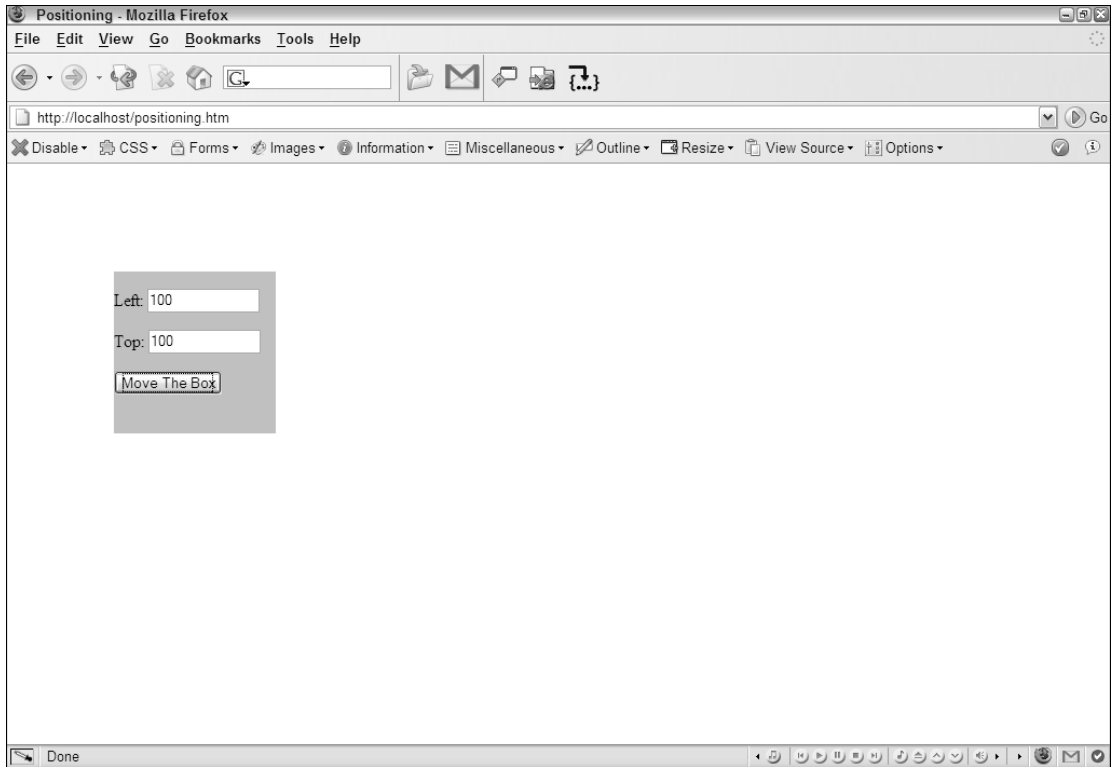


Figure 12-13

How It Works

In the body of the page, you define a `<div/>` tag with an `id` of `divBox`.

```
<div id="divBox"></div>
```

Inside this element is a form consisting of three `<input/>` elements. Two of these are text boxes in which you can input the left and top positions to move the `<div/>` to, and these have `ids` of `inputLeft` and `inputTop`, respectively. The third `<input/>` is a Submit button.

```
<div id="divBox">
  <form id="formBoxController">
    <p>Left: <input type="text" value="0" id="inputLeft" /></p>
    <p>Top: <input type="text" value="0" id="inputTop" /></p>
    <p><input type="submit" value="Move The Box" /></p>
  </form>
</div>
```

Chapter 12: Introduction to Dynamic HTML

When you click the Submit button, the browser fires the `submit` event for the form. When a submit button is pressed, the browser attempts to send data to the web server. This attempt at communication causes the browser to reload the page, making any change you made through DHTML reset itself. Therefore, you must force the browser to not reload the page. You do this by setting the `submit` event to return a value of `false`.

```
<div id="divBox">
  <form id="formBoxController" onsubmit="return false;">
    <p>Left: <input type="text" value="0" id="inputLeft" /></p>
    <p>Top: <input type="text" value="0" id="inputTop" /></p>
    <p><input type="submit" value="Move The Box" /></p>
  </form>
</div>
```

Now, when you click the Submit button, it appears that nothing happens. This is the desired result.

In order for the `<div/>` element to be moved around on the page, it needs to be positioned. This example positions the element absolutely, although it would be possible to position it relatively as well.

```
#divBox {
  position: absolute;
  background-color: silver;
  width: 150px;
  height: 150px;
}
```

Aside from the position, you also specify the box to have a background color of silver, and set the height and width to be 150 pixels each, to make it a square. At this size, however, the text boxes in the form actually extend past the box's borders. In order to fix this, set a rule for the `<input/>` tags as well.

```
input {
  width: 100px;
}
```

By setting the `<input/>` tags to be 100 pixels wide, you can fit everything nicely into the box. So at this point, the HTML is primarily finished and it's styled. All that remains is to write the JavaScript to retrieve the values from the form fields and move the box to the coordinates provided by the form.

The function responsible for this is called `moveBox()`, and it is the only function on this page.

```
function moveBox() {
  var divBox = document.getElementById("divBox"); //Get the box
  var inputLeft = document.getElementById("inputLeft"); //Get one form field
  var inputTop = document.getElementById("inputTop"); //Get the other one

  //more code here
}
```

The function starts by retrieving the HTML elements needed to move the box. First it gets the `<div/>` element itself, followed by the text boxes for the left and top positions, and stores them in the `inputLeft` and `inputTop` variables, respectively. With the needed elements selected, you can now move the box.

```
function moveBox() {  
    var divBox = document.getElementById("divBox");  
    var inputLeft = document.getElementById("inputLeft");  
    var inputTop = document.getElementById("inputTop");  
  
    divBox.style.left = parseInt(inputLeft.value) + "px";  
    divBox.style.top = parseInt(inputTop.value) + "px";  
}
```

These two new lines to `moveBox()` do just that. In the first new line, you use the `value` property to retrieve the value of the text box for the left position. You pass that value to the `parseInt()` function because you want to make sure that value is an integer. Then append `px` to the number, making sure that all browsers will position the box correctly. Next, do the same thing for positioning the top: Get the value from the `inputTop` text box, pass it to `parseInt()`, and append `px` to it.

One last thing: Currently, nothing happens when you click the submit button. You're using the form's submit event to suppress the page from attempting to send data to the server. Add in a call to `moveBox()` there, too. You need only to separate the statements with a semicolon.

```
<div id="divBox">  
    <form id="formBoxController" onsubmit="moveBox(); return false;">  
        <p>Left: <input type="text" value="0" id="inputLeft" /></p>  
        <p>Top: <input type="text" value="0" id="inputTop" /></p>  
        <p><input type="submit" value="Move The Box" /></p>  
    </form>  
</div>
```

And with that addition, the box moves to the location specified by the numbers input by the user.

Example: Animated Advertisement

Changing the appearance and position of an element are important patterns in DHTML, and they find their places in many DHTML scripts. Perhaps the most creative use of DHTML is in animating content on the page. You can perform a variety of animations with DHTML. You can fade text elements or images in and out, give them a swipe animation (making it look like as if they are wiped onto the page), and animate them to move around on the page.

Animation can give important information the flair it needs to be easily recognized by your reader, as well as adding a "that's cool" factor. Performing animation with DHTML follows the same principles of any other type of animation: You make seemingly insignificant changes one at a time in a sequential order until you reach the end of the animation. Essentially, with any animation you have the following requisites:

1. The starting state
2. The movement towards the final goal
3. The end state; stopping the animation

Moving an absolutely positioned element, as we're going to do in this section, is no different. First, with CSS, position the element at the start location. Then perform the animation up until you reach the end point, which signals the end of the animation.

Chapter 12: Introduction to Dynamic HTML

In this section you'll learn how to animate content to bounce back and forth between two points. To do this you need one important piece of information: the content's current location.

Are We There Yet?

The DOM in modern browsers exposes the `offsetTop` and `offsetLeft` properties of an HTML element object. These two properties return the calculated position relative to the element's parent element: `offsetTop` tells you the top location, and `offsetLeft` tells you the left position. The values returned by these properties are numerical values, so you can easily check to see where your element currently is in the animation. For example:

```
var endPointX = 394;

if (oElement.offsetLeft < endPointX) {
    //Continue animation
}
```

The preceding code specifies the end point—in this case, 394—and assigns it to the `endPointX` variable. You can then check to see if the element's `offsetLeft` value is currently less than that of the end point. If it is, you can continue the animation. This example brings us to the next topic in content movement: performing the animation.

Get 'Er Done

In order to perform an animation, you need to modify the `top` and `left` properties of the `style` object incrementally and quickly. In DHTML, you do this with periodic function execution until it's time to end the animation. To do this, use one of two methods of the window object: `setTimeout()` or `setInterval()`. This example uses the `setInterval()` method to periodically move an element.

Try It Out Animating Content

The following HTML page moves an element across the page from right to left:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
    <title>Moving Content</title>
    <style type="text/css">
        #divAdvert {
            position: absolute;
            font: 12px Arial;
            top: 4px;
            left: 0px;
        }
    </style>
    <script type="text/javascript">
        var switchDirection = false; //To keep track of which way we're going

        function doAnimation() {
            var divAdvert = document.getElementById("divAdvert"); //Get the element
            var currentLeft = divAdvert.offsetLeft; //Get the current left position
```



```

    var newLocation; //Will store the new location

    if (switchDirection == false) {
        newLocation = currentLeft + 2; //Move the text 2 pixels to the right

        if (currentLeft >= 400) { //We've reached our destination
            switchDirection = true; //So let's turn around
        }
    } else {
        newLocation = currentLeft - 2; //Move the text 2 pixels to the left

        if (currentLeft <= 0) { //We've reached our destination
            switchDirection = false; //So let's turn around
        }
    }

    divAdvert.style.left = newLocation + "px"; //Change the left position
}
</script>
</head>
<body onload="setInterval(doAnimation, 10)">
    <div id="divAdvert">Here is an advertisement.</div>
</body>
</html>

```

Save this page as `moving_content.htm` and load it into your browser. When you load the page into the browser, the content should start moving from left to right, starting at the left edge of the viewport. When the content reaches a left position of 400 pixels, the content switches directions and begins to move back toward the left edge. This animation is continuous, so it should bounce between the two points (0 and 400) perpetually.

How It Works

Inside the body of the page is a `<div/>` element. This element has an `id` of `divAdvert` so that you can retrieve it with JavaScript, as this is the tag you want to animate.

```
<div id="divAdvert">Here is an advertisement.</div>
```

There are no `style` attributes in this tag, as all the style information is inside the style sheet. In the style sheet, you define a starting point for this `<div/>`. You want the animation to go first from left to right, and you want it to start at the left edge of the browser.

```

#divAdvert {
    position: absolute;
    font: 12pt arial;
    top: 4px;
    left: 0px;
}

```

The first style declaration positions the element absolutely. Next, specify the font as 12-point Arial. The next declaration positions the element four pixels from the top of the browser's viewport. Setting the top position away from the topmost edge makes the text a little easier to read. Finally, position the `divAdvert` along the left edge of the viewport with the `left` property.

Chapter 12: Introduction to Dynamic HTML

Within the script block is a global variable called `switchDirection`.

```
var switchDirection = false; //To keep track of which way we're going
```

This variable keeps track of the direction in which the content is currently going. If `switchDirection` is `false`, then the content is moving from left to right, which is the default. If `switchDirection` is `true`, then the content is moving from right to left.

Next in the script block you find the `doAnimation()` function, which performs the animation.

```
function doAnimation() {  
    var divAdvert = document.getElementById("divAdvert"); //Get the element  
    var currentLeft = divAdvert.offsetLeft; //Get the current left position  
    var newLocation; //Will store the new location  
  
    // more code here  
}
```

First you retrieve the `divAdvert` element with the `getElementById()` method; you also retrieve the `offsetLeft` property and assign its value to the `currentLeft` variable. You use this variable to check the content's current position. Next you create a variable called `newLocation`. This variable will contain the new left position, but before you assign its value you need to know the direction in which the content is moving.

```
function doAnimation() {  
    var divAdvert = document.getElementById("divAdvert"); //Get the element  
    var currentLeft = divAdvert.offsetLeft; //Get the current left position  
    var newLocation; //Will store the new location  
  
    if (switchDirection == false) {  
        newLocation = currentLeft + 2; //Move the text 2 pixels to the right  
  
        if (currentLeft >= 400) { //We've reached our destination  
            switchDirection = true; //So let's turn around  
        }  
    }  
  
    //more code here  
}
```

First check the direction by checking the `switchDirection` variable. Remember, if it is `false`, the animation is moving from left to right; so assign `newLocation` to contain the content's current position and add 2, thus moving the content 2 pixels to the right.

You then need to check if the content has reached the left position of 400 pixels. If it has, then you need to switch the direction of the animation, and you do this by changing `switchDirection` to `true`. So the next time `doAnimation()` runs, it will begin to move the content from right to left.

The code for this new direction is similar to the previous code, except for a few key differences.

```
function doAnimation() {  
    var divAdvert = document.getElementById("divAdvert"); //Get the element  
    var currentLeft = divAdvert.offsetLeft; //Get the current left position
```

```
var newLocation; //Will store the new location

if (switchDirection == false) {
    newLocation = currentLeft + 2; //Move the text 2 pixels to the right

    if (currentLeft >= 400) { //We've reached our destination
        switchDirection = true; //So let's turn around
    }
}

else {
    newLocation = currentLeft - 2; //Move the text 2 pixels to the left

    if (currentLeft <= 0) { //We've reached our destination
        switchDirection = false; //So let's turn around
    }
}

//more code here
}
```

The first difference is the value assigned to `newLocation`; instead of adding 2 to the current location, you subtract 2, thus moving the content 2 pixels to the left. Next, check if `currentLeft` is less than or equal to 0. If it is, you know you've reached the ending point of the right-to-left movement and need to switch directions again by assigning `switchDirection` to be false.

Finally, set the new position of the content.

```
function doAnimation() {
    var divAdvert = document.getElementById("divAdvert"); //Get the element
    var currentLeft = divAdvert.offsetLeft; //Get the current left position
    var newLocation; //Will store the new location

    if (switchDirection == false) {
        newLocation = currentLeft + 2; //Move the text 2 pixels to the right

        if (currentLeft >= 400) { //We've reached our destination
            switchDirection = true; //So let's turn around
        }
    }

    else {
        newLocation = currentLeft - 2; //Move the text 2 pixels to the left

        if (currentLeft <= 0) { //We've reached our destination
            switchDirection = false; //So let's turn around
        }
    }

    divAdvert.style.left = newLocation + "px"; //Change the left position
}
```

This final line completes the `doAnimation()` function, and it sets the element's left position to the value contained in the `newLocation` variable.

Chapter 12: Introduction to Dynamic HTML

To run the animation, use the `onload` event handler in the `<body/>` element, and use the `window.setInterval()` method to continuously execute `doAnimation()`. The following code runs `doAnimation()` every 10 milliseconds:

```
<body onload="setInterval(doAnimation, 10)">
```

At this speed, the content moves at a pace that is easily seen by those viewing the page.

Summary

DHTML is a large topic, and we've touched on some of its fundamentals in this chapter. With this knowledge, you can explore other avenues of DHTML and create more complex user interface tricks.

The main points this chapter covered were as follows:

- ❑ Despite leaps and bounds by browser makers, some discrepancies still exist. You learned how to cope with two different event objects by branching your code to consolidate two different APIs into one.
- ❑ Style sheets can be used to set the style for various types of tags, or for individual tags. You can also define classes that you can apply to certain tags using the `class` attribute. Styles include the font, color, and position of a tag.
- ❑ DHTML enables you to change a page after it is loaded into the browser, and you can perform a variety of user interface tricks to add some flair to your page.
- ❑ You learned how to change a tag's style by using the `style` and `className` properties.
- ❑ You also learned the basics of animation in DHTML, and made text bounce back and forth between two points.

Exercise Questions

Suggested solutions to these questions can be found in Appendix A.

Question 1

Create a web page that contains two links. The first link should say `Show First Box` and the second `Show Second Box`. Then add two `<div/>` elements and set their `id` attributes to `boxOne` and `boxTwo`. Give them a height, width, background color, and position, and then hide them. Next, set up the links so that when you click the first one, only the first box shows, and when you click the second one, only the second box shows.

Question 2

Create a `<div/>` element that floats around the page. Use the edges of the browser's viewport as a boundary.

13

Dynamic HTML in Modern Browsers

In the last chapter you were introduced to DHTML, getting a small glimpse of what it can do with a scrolling text animation. You also saw that web developers need to use different code for different browsers to achieve the same results, which makes their lives difficult. In this chapter you'll be concentrating on writing code according to the web standards set by the W3C. In general, you can write code that will work with IE 6+, Firefox, Opera, and Safari without having to make big changes.

One of the most misunderstood sections in the W3C Web standards is the Document Object Model, or DOM for short. The DOM gives developers a way of representing everything on a web page so that it is accessible via a common set of properties and methods in JavaScript, or any other object-based programming language. By everything, we mean *everything*. You can change the graphics, tables, forms, and even text itself by altering a relevant DOM property with JavaScript.

The DOM should not be confused with the Browser Object Model (BOM) that was introduced in Chapter 5. You'll see the differences between the two in detail shortly. For now, though, think of the BOM as a browser-dependent representation of every feature of the browser, from the browser buttons, URL address line, and title bar to the browser window controls, as well as parts of the web page, too. The DOM, however, deals only with the contents of the browser window or web page—in other words, the HTML document. It makes the document available in such a way that any browser can use exactly the same code to access and manipulate the content of the document. To summarize, the BOM gives you access to the browser and some of the document, whereas the DOM gives you access to all of the document, but *only* the document.

The great thing about the DOM is that it is browser- and platform-independent. This means that developers can finally consider the possibility of writing a piece of JavaScript code that dynamically updates the page, as you saw in the last chapter, and that will work on any DOM-compliant browser without any tweaking. You should not need to code for different browsers or take excessive care when coding.

The DOM achieves this independence by representing the contents of the page as a generic tree structure. Whereas in the BOM you might expect to access something by looking up a property relevant to that part of the browser and adjusting it, the DOM requires navigation through its

Chapter 13: Dynamic HTML in Modern Browsers

representation of the page through nodes and properties that are not specific to the browser. You'll explore this structure a little later.

However, to use the DOM standard, ultimately developers require browsers that completely implement the standard, something that no browser does 100 percent efficiently. Unfortunately, IE 5.5 didn't support many aspects of the standard. IE 6 and 7 have greatly improved standards support, but still fall short of full implementation. Even Firefox, which at least aims to support the standard, still falls short in some ways.

This makes the DOM sound like an impossible ideal, yet it doesn't exist purely as a standard. Features of the DOM standard have been implemented in browsers as far back as Netscape 2. However, in Netscape versions 2, 3, and 4, many HTML page elements and their properties were not scriptable at all — whereas, as you have seen, IE made nearly all page elements and their properties scriptable. Unfortunately, the way in which Netscape 4 provided scripting access to some elements was often incompatible with the way in which IE made those elements available. Thus a Document Object Model standard was developed. The latest versions of IE, Mozilla, Opera, and Safari support many features outlined in the standard.

To provide a true perspective on how the DOM fits in, we need to take a brief look at its relationship with some of the other currently existing web standards. We should also talk about why there is more than one version of the DOM standard, and why there are different sections within the standard itself. (Microsoft, in particular, added a number of extensions to the W3C DOM.) After understanding the relationships, you can look at using JavaScript to navigate the DOM and to dynamically change content on web pages in more than one browser, in a way that used to be impossible with pure DHTML. The following items are on your agenda:

- ☐ The HTML, ECMAScript, XML, and XHTML web standards
- ☐ The DOM standards
- ☐ The DOM tree structure
- ☐ Writing cross-browser DHTML

Remember that the examples within this chapter are targeted only at the DOM and therefore will be supported only by IE 6+, Firefox 1+, Opera, and Safari.

Why Do We Need Web Standards?

When Tim Berners-Lee created HTML in 1991, he probably had little idea that this technology for marking up scientific papers via a set of tags for his own global hypertext project, known as the World Wide Web, would within a matter of years become a battleground between the two giants of the software business of the mid-1990s. HTML was a simple derivation from the meta-language Standard Generalized Markup Language (SGML) that had been kicking around academic institutions for decades. Its purpose was to preserve the structure of the documents created with it. HTML depends on a protocol, HyperText Transfer Protocol (HTTP), to transmit documents back and forth between the resource and the viewer (for example, the server and the client computer). These two technologies formed the foundation of the Web, and it quickly became obvious in the early 1990s that there needed to be some sort of policing of both specifications to ensure a common implementation of HTML and HTTP so that communications could be conducted worldwide.

In 1994, Tim founded the World Wide Web Consortium (W3C), a body that set out to oversee the technical evolution of the Web. It has three main aims:

- ❑ To provide universal access, so that anybody can use the Web
- ❑ To develop a software environment to allow users to make use of the Web
- ❑ To guide the development of the Web, taking into consideration the legal, social, and commercial issues that arise

Each new version of a specification of a web technology has to be carefully vetted by W3C before it can become a standard. The HTML and HTTP specifications are subject to this process, and each new set of updates to these specifications yields a new version of the standard. Each standard has to go through a working draft, a candidate recommendation, and a proposed recommendation stage before it can be considered a fully operational standard. At each stage of the process, members of the W3C consortium vote on which amendments to make, or even on whether to cancel the standard completely and send it back to square one.

It sounds like a very painful and laborious method of creating a standard format, and not something you'd think of as spearheading the cutting edge of technical revolution. Indeed, the software companies of the mid-1990s found the processes involved too slow, so they set the tone by implementing new innovations themselves and then submitting them to the standards body for approval. Netscape started by introducing new elements in its browser, such as the `` element, to add presentational content to the web pages. This proved popular, so Netscape added a whole raft of elements that enabled users to alter aspects of presentation and style on web pages. Indeed, JavaScript itself was such an innovation from Netscape.

When Microsoft entered the fray, it was playing catchup for the first two iterations of its Internet Explorer browser. However, with Internet Explorer 3 in 1996, they established a roughly equal set of features to compete with Netscape and so were able to add their own browser-specific elements. Very quickly, the Web polarized between these two browsers, and pages viewable on one browser quite often wouldn't appear on another. One problem was that Microsoft had used its much stronger position in the market to give away its browser for free, whereas Netscape still needed to sell its own browser because it couldn't afford to freely distribute its flagship product. To maintain a competitive position, Netscape needed to offer new features to make the user want to purchase its browser rather than use the free Microsoft browser.

Things came to a head with both companies' version 4 browsers, which introduced dynamic page functionality. Unfortunately, Netscape did this by the means of a `<layer>` element, whereas Microsoft chose to implement it via scripting language properties and methods. The W3C needed to take a firm stand here, because one of its three principal aims had been compromised: that of universal access. How could access be universal if users needed a specific vendor's browser to view a particular set of pages? They decided on a solution that used existing standard HTML elements and Cascading Style Sheets, both of which had been adopted as part of the Microsoft solution. As a result, Microsoft gained a dominant position in the browser war. It hasn't relinquished this position; the Netscape Navigator browser never had a counter to Internet Explorer's constant updates, and its replacement, Firefox, has been slow to expand its user base. Current usage statistics hover between 70 versus 30 percent and 90 versus 10 percent in Microsoft's favor.

With a relatively stable version of the HTML standard in place with version 4.01, which boasts a set of features that will take any browser manufacturer a long time to implement completely, attention was turned to other areas of the Web. A new set of standards was introduced in the late 1990s to govern the means of presenting HTML (style sheets) and the representation of the HTML document in script (the Document Object Model or DOM). Other standards emerged, such as Extensible Markup Language (XML), which offers a common format for representing data in a way that preserves its structure. You'll take a look now at the main standards that have been created, and learn a bit about what each of the technologies does.

The Web Standards

The W3C web site (www.w3.org) has a huge number of standards in varying stages of creation. Not all of these standards concern us, and not all of the ones that concern us can be found at this web site. However, the vast majority of standards that do concern us can be found there.

You're going to take a brief look now at the technologies and standards that have an impact on JavaScript and find out a little background information about each. Some of the technologies may be unfamiliar, but you need to be aware of their existence at the very least.

HTML

The HTML standard is maintained by W3C. This standard might seem fairly straightforward, given that each version should have introduced just a few new elements, but in reality the life of the standards body was vastly complicated by the browser wars. The versions 1.0 and 2.0 of HTML were simple, small documents, but when W3C came to debate HTML version 3.0, they found that much of the new functionality it was discussing had already been superceded by new additions, such as the `<applet/>` and `<style/>` elements, to the version 3.0 browser's `appletstyle`. Version 3.0 was discarded, and a new version, 3.2, became the standard.

However, a lot of the features that went into HTML 3.2 had been introduced at the behest of the browser manufacturers and ran contrary to the spirit of HTML, which was intended solely to define structure. The new features, stemming from the `` element, just confused the issue and added unnecessary presentational features to HTML. These features really became redundant with the introduction of style sheets. So suddenly, in the version 3 browsers, there were three distinct ways to define the style of an item of text. Which was the correct way? And if all three ways were used, which style did the text ultimately assume? Version 4.0 of the HTML standard was left with the job of un-muddling this chaotic mess and designated a lot of elements for deprecation (removal) in the next version of the standards. It was the largest version of the standard so far and included features that linked it to style sheets and the Document Object Model, and also added facilities for the visually impaired and other unfairly neglected minority interest areas. The current version of the HTML standard is 4.01.

ECMAScript

JavaScript itself followed a trajectory similar to that of HTML. It was first used in Netscape Navigator and then added to Internet Explorer. The Internet Explorer version of JavaScript was christened Jscript and wasn't far removed from the version of JavaScript found in Netscape Navigator. However, once again there were differences between the two implementations and a lot of care had to be taken in writing script for both browsers.

Oddly enough, it was left to the European Computer Manufacturers Association (ECMA) to propose a standard specification for JavaScript. This didn't appear until a few versions of JavaScript had already been released. Unlike HTML, which had been developed from the start with the W3C consortium, JavaScript was a proprietary creation. This is the reason that it is governed by a different standards body. Microsoft and Netscape both agreed to use ECMA as the standards vehicle/debating forum, because of its reputation for fast-tracking standards and perhaps also because of its perceived neutrality. The name ECMAScript was chosen so as not to be biased toward either vendor's creation and also because the "Java" part of JavaScript was a trademark of Sun licensed to Netscape. The standard, named ECMA-262, laid down a specification that was roughly equivalent to the JavaScript 1.1 specification.

That said, the ECMAScript standard covers only core JavaScript features, such as the primitive data types of numbers, strings, and Booleans, native objects like the `Date`, `Array`, and `Math` objects, and the procedural statements like `for` and `while` loops, and `if` and `else` conditionals. It makes no reference to client-side objects or collections, such as `window`, `document`, `forms`, `links`, and `images`. So, although the standard helps to make core programming tasks compatible when both JavaScript and JScript comply with it, it is of no use in making the scripting of client-side objects compatible between the main browsers. Some incompatibilities remain.

All current implementations of JavaScript are expected to conform to the current ECMAScript standard, which is ECMAScript edition 3, published in December 1999. As of November 2006, ECMAScript edition 4 is under development.

Although in the version 3 browsers there were quite a few irregularities between the Microsoft and Netscape dialects of JavaScript, they're now similar enough to be considered the same language. The Opera and Safari browsers also support and offer the same kind of support for the standard. This is a good example of how standards have provided a uniform language across browser implementations, although a feature war similar to the one that took place over HTML still rages to a lesser degree over JavaScript.

XML

Extensible Markup Language, or XML, is a standard for creating markup languages (such as HTML). XML itself has been designed to look as much like HTML as possible, but that's where the similarities end.

HTML is actually an application of the meta-language SGML, which is also a standard for generating markup languages. SGML has been used to create many markup languages, but HTML is the only one that enjoys universal familiarity and popularity. XML, on the other hand, is a direct subset of SGML. SGML is generally considered to be too complex for people to be able to accurately represent it on a computer, so XML is a simplified subset of SGML. XML is also much easier to read than SGML.

XML's main use is for the creation of customized markup languages that are very similar in look and structure to HTML. One main use of XML is in the representation of data. Whereas a normal database can store information, databases don't allow individual stored items to contain information about their structure. XML can use the element structure of markup languages to represent any kind of data in which information contained in the structure might otherwise be lost, from mathematical and chemical notations to the entire works of Shakespeare. For instance, an XML document could be used to record that Mark Anthony doesn't appear until Scene II Act I of Shakespeare's play *Julius Caesar*, whereas a relational database would struggle to do this without a lot of extra fields, as the following example shows:

```
<play>
  <act1>
    <scene1>
      ...
    </scene1>
    <scene2>
      <mark_anthony>
        Caesar, my lord?
      </mark_anthony>
    </scene2>
    <scene3>
      ...
    </scene3>
  </act1>
  <act2>
    ...
  </act2>
  <act3>
    ...
  </act3>
  <act4>
    ...
  </act4>
  <act5>
    ...
  </act5>
</play>
```

XML is also completely cross-platform, because it contains just text. This means that an application on Windows can package up the data in this format, and a completely different application on Unix should be able to unravel and read those data.

XML is more complex than HTML. Whereas a browser will take HTML code, interpret the relevant details, and display the corresponding web page without any intervention, interpreting XML requires several extra steps.

Because you're creating the markup language yourself, you need first to create a set of rules through which the language will be run. You can do this in one of two ways, either with an *XML schema* or with a *Document Type Definition* (DTD). Both of these are used to draw up rules, such as which elements you can use in your markup language, which attributes these elements take, and what kind of data these attributes are expecting.

Secondly, when you've written your XML document in your new language, it must be checked against both the syntax rules laid down for XML documents and the rules in the schema or the DTD to see if the code conforms. You'll be taking an in-depth look at XML in the next chapter.

XHTML

XHTML 1.0 is where the XML and HTML standards meet. XHTML is just a respecification of the HTML 4.01 standard as an XML application. The advantages of this allow XHTML to get around some of the problems caused by a browser's particular interpretation of HTML, and more importantly to provide a

specification that allows the Web to be used by clients other than browsers, such as those provided on handheld computers, mobile phones, or any software device that might be connected to the Internet (perhaps even your refrigerator!).

XHTML also offers a common method for specifying your own elements, instead of just adding them randomly. You can specify new elements via a common method using an XML DTD and an XML *namespace*. (A namespace is a means of identifying one set of elements uniquely from any other set of elements.) This is particularly useful for the new markup languages, such as Wireless Markup Language (WML), which are geared toward mobile technology and require a different set of elements to be able to display on the reduced interfaces.

That said, anyone familiar with HTML should be able to look at an XHTML page and understand what's going on. There are differences, but not ones that add new elements or attributes.

The following is a list of the main differences between XHTML and HTML:

- ☐ XHTML recommends an XML declaration to be placed at the top of the file in the following form: `<?xml-version='1.0' ?>`.
- ☐ You also have to provide a DTD declaration at the top of the file, referencing the version of the DTD standard you are using.
- ☐ You have to include a reference to the XML namespace within the HTML element.
- ☐ You need to supply all XHTML element names in lowercase, because XML is case-sensitive.
- ☐ The `<head/>` and `<body/>` elements must always be included in an XHTML document.
- ☐ Tags must always be closed and nested correctly. When only one tag is required, such as with line breaks, the tag is closed with a slash (for example, `
`).
- ☐ Attribute values must always be denoted by quotation marks.

This set of rules makes it possible to keep a strict hierarchical structure to the elements, which in turn makes it possible for the Document Object Model to work correctly. This is the route that HTML is currently taking, and all future HTML standards will be XHTML standards. This also makes it possible to standardize markup languages across all device types, so that the next version of WML (the markup language of mobile devices) will also be compliant with the XHTML standard. You should now be creating your HTML documents according to the previously specified rules. If you do so, you will find it much, much easier to write JavaScript that manipulates the page via the DOM and works in the way it was intended.

It's now time for you to consider the Document Object Model itself.

The Document Object Model

The Document Object Model (DOM) is, as previously mentioned, a way of representing the document independent of browser type. It allows a user to access the document via a common set of objects, properties, methods, and events, and to alter the contents of the web page dynamically using scripts.

Chapter 13: Dynamic HTML in Modern Browsers

Several types of script languages, such as JavaScript and VBScript, are available. Each requires a slightly different syntax and therefore a different approach when you're programming. Even when you're using a language common to all browsers, such as JavaScript, you should be aware that some small variations are usually added to the language by the vendor. So, to guarantee that you don't fall afoul of a particular implementation, the W3C has provided a generic set of objects, properties, and methods that should be available in all scripting languages, in the form of the DOM standard.

The DOM Standard

We haven't talked about the DOM standard so far, and for a particular reason: It's not the easiest standard to follow. Supporting a generic set of properties and methods has proved to be a very complex task, and the DOM standard has been broken down into separate levels and sections to deal with the different areas. The different levels of the standard are all at differing stages of completion.

Level 0

Level 0 is a bit of a misnomer, as there wasn't really a level 0 of the standard. This term in fact refers to the "old way" of doing things—the methods implemented by the browser vendors before the DOM standard. Someone mentioning level 0 properties is referring to a more linear notation of accessing properties and methods. For example, typically you'd reference items on a form with the following code:

```
document.forms[0].elements[1].value = "button1";
```

We're not going to cover such properties and methods in this chapter, because they have been superseded by newer methods.

Level 1

Level 1 is the first version of the standard. It is split into two sections: one is defined as core (objects, properties, and methods that can apply to both XML and HTML) and the other as HTML (HTML-specific objects, properties, and methods). The first section deals with how to go about navigating and manipulating the structure of the document. The objects, properties, and methods in this section are very abstract. The second section deals with HTML only and offers a set of objects corresponding to all the HTML elements. This chapter mainly deals with the second section—level 1 of the standard.

In 2000, level 1 was revamped and corrected, though it only made it to a working draft and not to a full W3C recommendation.

Level 2

Level 2 is complete and many of the properties, methods, and events have been implemented by modern browsers. It has sections that add specifications for events and style sheets to the specifications for core and HTML-specific properties and events. (It also provides sections on views and traversal ranges, neither of which will be covered in this book; you can find more information at www.w3.org/TR/2000/PR-DOM-Level-2-Views-20000927/ and www.w3.org/TR/2000/PR-DOM-Level-2-Traversal-Range-20000927/.) You will be making use of some of the features of the event and style sections of this level of the DOM later in this chapter because they have been implemented in the latest versions of both browsers.

Level 3

Level 3 is still under development. It is intended to resolve a lot of the complications that still exist in the event model in level 2 of the standard, and adds support for XML features, such as contents models and being able to save the DOM as an XML document.

Browser Compliance with the Standards

Almost no browser has 100 percent compliance with *any* standard, although some, such as Firefox and Opera, come pretty close with the DOM. Therefore, there is no guarantee that all the objects, properties, and methods of the DOM standard will be available in a given version of a browser, although a few level 1 and level 2 objects, properties, and methods have been available in all the browsers for some time. However, IE6++ and Firefox offer by far the closest compliance, with Opera and Safari catching up.

Much of the material in the DOM standards has only recently been clarified, and a lot of DOM features and support have been added to only the latest browser versions. For this reason, examples in this chapter will be guaranteed to work on only the latest versions of IE, Firefox, Opera, and Safari. Although cross-browser scripting is a realistic goal, backwards support isn't at all.

Although the standards might still not be fully implemented, they do give you an idea as to how a particular property or method should be implemented, and provide a guideline for all browser manufacturers to agree to work toward in later versions of their browsers. The DOM doesn't introduce any new HTML elements or style sheet properties to achieve its ends. The idea of the DOM is to make use of the existing technologies, and quite often the existing properties and methods of one or other of the browsers.

Differences Between the DOM and the BOM

As mentioned earlier, there are two main differences between the Document Object Model and the Browser Object Model. However, complicating the issue is the fact that a BOM is sometimes referred to under the name DOM. Look out for this in any literature on the subject.

First, the DOM covers only the document of the web page, whereas the BOM offers scripting access to all areas of the browsers, from the buttons to the title bar, including some parts of the page.

Second, the BOM is unique to a particular browser. This makes sense if you think about it: You can't expect to standardize browsers, because they have to offer competitive features. Therefore, you need a different set of properties and methods and even objects to be able to manipulate them with JavaScript.

Representing the HTML Document as a Tree Structure

Because HTML is standardized so that web pages can contain only the standard features supported in the language, such as forms, tables, images, and the like, a common method of accessing these features is needed. This is where the DOM comes in. It provides a uniform representation of the HTML document, and it does this by representing the entire HTML document/web page as a *tree structure*.

Only the more recent browser versions allow access to all parts of the web page via such a tree structure. Previous versions would only partially represent web pages in this tree structure, and leave bits, such as the text, beyond reach and inaccessible to script.

Chapter 13: Dynamic HTML in Modern Browsers

In fact, it is possible to represent any HTML document (in fact, any XML document) as a tree structure. The only precondition is that the HTML document should be well formed. You’ve already learned why this is such a desirable attribute, but it doesn’t hurt to emphasize it again. Different browsers might be tolerant, to a greater or lesser extent, of quirks such as unclosed tags, or HTML form controls not being enclosed within a `<form>` element; however, for the structure of the HTML document to be accurately depicted, you need to be able to always predict the structure of the document. Abuses of the structure, such as unclosed tags, stop you from depicting the structure as a true hierarchy, and therefore cannot be allowed. The ability to access elements via the DOM depends on the ability to represent the page as a hierarchy.

What Is a Tree Structure?

If you’re not familiar with the concept of trees, don’t worry. They’re just a diagrammatic means of representing a hierarchical structure.

Let’s consider the example of a book with several chapters. If instructed to, you could find the third line on page 543 after a little searching. If an updated edition of the book were printed with extra chapters, more likely than not you’d fail to find the same text if you followed those same instructions. However, if the instructions were changed to, say, “Find the chapter on still-life painting, the section on using watercolors, and the paragraph on positioning light sources,” you’d be able to find that even in a reprinted edition with extra pages and chapters, albeit with perhaps a little more effort than the first request required.

Books aren’t particularly dynamic examples, but given something like a web page, where the information could be changed daily, or even hourly, can you see why it would be of more use to give the second set of directions than the first? The same principle applies with the DOM. Navigating the DOM in a hierarchical fashion, rather than in a strictly linear way, makes much more sense. When you treat the DOM as a tree, it becomes easy to navigate the page in this fashion. Consider how you locate files on Windows using Windows Explorer, which creates a tree view of folders through which you can drill down. Instead of looking for a file alphabetically, you locate it by going into a particular folder.

The rules for creating trees are simple. You start at the top of the tree with the document and the element that contains all other elements in the page. The document is the *root node*. A *node* is just a point on the tree representing a particular element or attribute of an element, or even the text that an element contains. The root node contains all other nodes, such as the DTD declaration, the XML declaration, and the root element (the HTML or XML element that contains all other elements). The root element should always be the `<html>` element in an HTML document. Underneath the root element are the HTML elements that the root element contains. Typically an HTML page will have `<head>` and `<body>` elements inside the `<html>` element. These elements are represented as nodes underneath the root element’s node, which itself is underneath the root node at the top of the tree (see Figure 13-1).

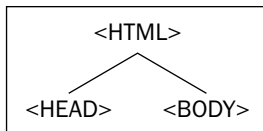


Figure 13-1

The two nodes representing the `<head>` and `<body/>` elements are examples of *child nodes*, and the `<html/>` element's node above them is a *parent node*. Since the `<head/>` and `<body/>` elements are both child nodes of the `<html/>` element, they both go on the same level underneath the parent node `<html/>` element. The `<head/>` and `<body/>` elements in turn contain other child nodes/HTML elements, which will appear at a level underneath their nodes. So child nodes can also be parent nodes. Each time you encounter a set of HTML elements within another element, they each form a separate node at the same level on the tree. The easiest way of explaining this clearly is with an example.

An Example HTML Page

Let's consider a basic HTML page such as this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" ><head>
</head>
<body>
  <h1>My Heading</h1>
  <p>This is some text in a paragraph</p>
</body>
</html>
```

The `<html/>` element contains `<head/>` and `<body/>` elements. Only the `<body/>` element actually contains anything. It contains an `<h1/>` element and a `<p/>` element. The `<h1/>` element contains the text `My Heading`. When you reach an item, such as text, an image, or an element, that contains no others, the tree structure will terminate at that node. Such a node is termed a *leaf node*. You then continue to the `<p/>` node, which contains some text, which is also a node in the document. You can depict this with the tree structure shown in Figure 13-2.

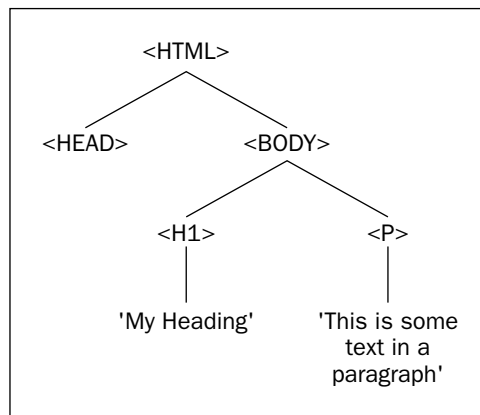


Figure 13-2

Simple, eh? This example is almost too straightforward, so let's move on to a slightly more complex one that involves a table as well.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
  <title>This is a test page</title>
</head>
<body>
  <span>Below is a table...</span>
  <table border="1">
    <tr>
      <td>Row 1 Cell 1</td>
      <td>Row 1 Cell 2</td>
    </tr>
    <tr>
      <td>Row 2 Cell 1</td>
      <td>Row 2 Cell 2</td>
    </tr>
  </table>
</body>
</html>
```

There is nothing out of the ordinary here. The document just contains a table with two rows, and two cells in each row. You can once again represent the hierarchical structure of your page (for example, the fact that the `<html/>` element contains a `<head/>` and a `<body/>` element, and that the `<head/>` element contains a `<title/>` element, and so on) using your tree structure, as shown in Figure 13-3.

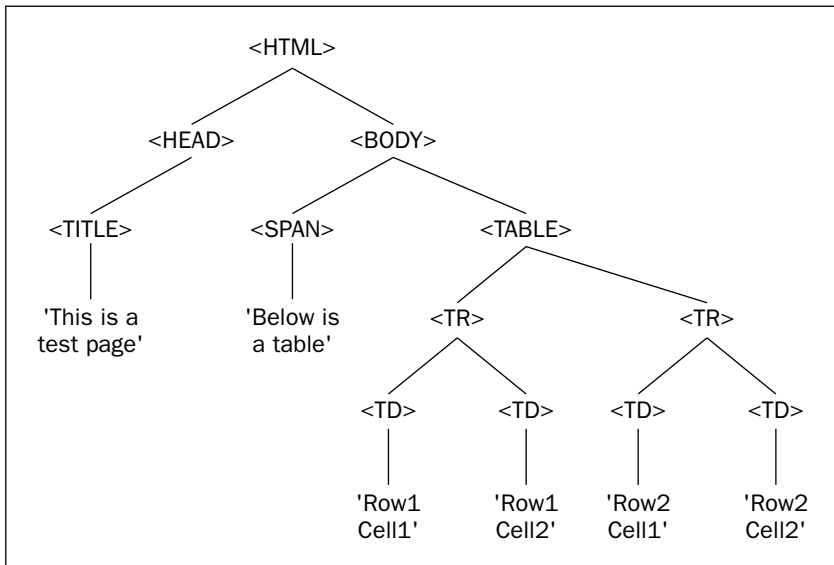


Figure 13-3

The top level of the tree is simple enough; the `<html/>` element contains `<head/>` and `<body/>` elements. The `<head/>` element in turn contains a `<title/>` element and the `<title/>` element contains some text. This text node is a child node that terminates the branch (a leaf node). You can then go back to the next node, the `<body/>` element node, and go down that branch. Here you have two elements contained within the `<body/>` element, the `` and `<table/>` elements. Although the `` element contains only text and terminates there, the `<table/>` element contains two rows `<tr/>`, and the two `<tr/>` elements contain two table cell `<td/>` elements. Only then do you get to the bottom of the tree with the text contained in each table cell. Your tree is now a complete representation of your HTML code.

The DOM Objects

What you have seen so far has been highly theoretical, so let's get a little more practical now.

The DOM provides you with a concrete set of objects, properties, and methods that you can access through JavaScript to navigate the tree structure of the DOM. Let's start with the set of objects, within the DOM, that is used to represent the nodes (elements, attributes, or text) on your tree.

Base DOM Objects

Three objects, shown in the following table, are known as the base DOM objects.

Object	Description
Node	Each node in the document has its own Node object
NodeList	This is a list of Node objects
NamedNodeMap	This provides access by name rather than by index to all the Node objects

This is where the DOM differs from the BOM quite extensively. The BOM objects have names that relate to a specific part of the browser, such as the `window` object, or the `forms` and `images` arrays. As mentioned earlier, to be able to navigate in the web page as though it were a tree, you have to do it abstractly. You can have no prior knowledge of the structure of the page; everything ultimately is just a node. To move around from HTML element to HTML element, or element to attribute, you have to go from node to node. This also means you can add, replace, or remove parts of your web page without affecting the structure as a whole, as you're just changing nodes. This is why you have three rather obscure-sounding objects that represent your tree structure.

I've already mentioned that the top of your tree structure is the root node, and that the root node contains the XML declaration, the DTD, and the root element. Therefore you need more than just these three objects to represent your document. In fact there are different objects to represent the different types of nodes on the tree.

High-Level DOM Objects

As you have seen, nodes come in a variety of types. Is the node an element, an attribute, or just plain text? The `Node` object has different objects to represent each possible type of node. The following is a complete list of all the different node type objects that can be accessed via the DOM. A lot of them won't concern you in this book because they pertain only to XML documents and not HTML documents, but you should notice that your three main types of nodes, namely element, attribute, and text, are all covered.

Object	Description
Document	The root node of the document
DocumentType	The DTD or schema type of the XML document
DocumentFragment	A temporary storage space for parts of the document
EntityReference	A reference to an entity in the XML document
Element	An element in the document
Attr	An attribute of an element in the document
ProcessingInstruction	A processing instruction
Comment	A comment in an XML document or HTML document
Text	Text that must form a child node of an element
CDataSection	A CDATA section within the XML document
Entity	An unparsed entity in the DTD
Notation	A notation declared within a DTD

We won't go over most of these objects in this chapter, but if you need to navigate the DOM of an XML document, you will have to use them.

Each of these objects inherits all the properties and methods of the `Node` object, but also has some properties and methods of its own. You will be looking at some examples in the next section.

DOM Properties and Methods

If you tried to look at the properties and methods of all the objects in the DOM, it would take up half the book. Instead you're going to actively consider only three of the objects, namely the `node` object, the `element` object, and the `document` object. This is all you'll need to be able to create, amend, and navigate your tree structure. Also, you're not going to spend ages trawling through each of the properties and methods of these objects, but rather look only at some of the most useful properties and methods and use them to achieve specific ends. You'll start with two of the most useful methods of the `document` object itself.

Methods of the document Object: Returning an Element or Elements

Let's begin at the most basic level. You have your HTML web page, so how do you go about getting back a particular element on the page in script? The two cross-browser ways of doing this are in the following table.

Methods of the document Object	Description
<code>getElementById(idvalue)</code>	Returns a reference (a node) to an element, when supplied with the value of the <code>id</code> attribute of that element
<code>getElementsByTagName(tagname)</code>	Returns a reference (a node list) to a set of elements that have the same tag as the one supplied in the argument

The first of the two methods, `getElementById()`, requires you to ensure that every element you want to access in the page uses an `id` attribute, otherwise a null value (a word indicating a missing or unknown value) will be returned by your method. Let's go back to your first example and add some `id` attributes to your elements.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"><head>
  <title>example</title>
</head>
<body>
  <h1 id="Heading1">My Heading</h1>
  <p id="Paragraph1">This is some text in a paragraph</p>
</body>
</html>
```

Now you can use the `getElementById()` method to return a reference to any of the HTML elements with `id` attributes on your page. For example, if you add the following code, you can reference the `<h1/>` element:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
  <title>example</title>
</head>
<body>
  <h1 id="Heading1">My Heading</h1>
  <p id="Paragraph1">This is some text in a paragraph</p>
  <script type="text/javascript">
    alert(document.getElementById("Heading1"));
  </script>
</body>
</html>
```

This will display the page shown in Figure 13-4.



Figure 13-4

You might have been expecting it to return something along the lines of `<h1/>` or `<h1 id="Heading1">`, but all it's actually returning is a reference to the `<h1/>` element. This reference to the `<h1/>` element is more useful though, as you can use it to alter attributes of the element, such as by changing the color or size. You can do this via the `style` object.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>example</title>
</head>
<body>
  <h1 id="Heading1">My Heading</h1>
  <p id="Paragraph1">This is some text in a paragraph</p>
  <script type="text/javascript">
    var H1Element = document.getElementById("Heading1");
    H1Element.style.fontFamily = "Arial";
  </script>
</body>
</html>
```

If you display this in the browser, you see that you can directly influence the attributes of the `<h1/>` element in script, as you have done here by changing its font type to Arial (see Figure 13-5).

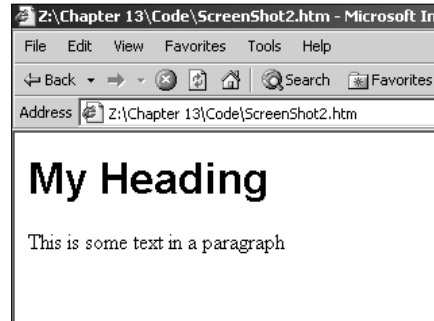


Figure 13-5

The second of the two methods, `getElementsByTagName()`, works in the same way, but, as its name implies, it can return more than one element. If you were to go back to the second example with the table and use this method to return the table cells (`<td>`) in your code, you would get a total of four table cells returned to your object. You'd still have only one object returned, but this object would be a collection of elements. To reference a particular element in the collection, you would have to be more precise. You need to specify an index number, which you do using the `item()` method.

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>This is a test page</title>
</head>
<body>
  <span>Below is a table... </span>
  <table border="1">
    <tr>
      <td>Row 1 Cell 1</td>
      <td>Row 1 Cell 2</td>
    </tr>
    <tr>
      <td>Row 2 Cell 1</td>
      <td>Row 2 Cell 2</td>
    </tr>
  </table>
  <script type="text/javascript">
    var TDElement = document.getElementsByTagName("td").item(0);
    TDElement.style.fontFamily = "arial";
  </script>
</body>
</html>

```

Like arrays, collections are zero-based, and so the first element in the table would correspond to the index number 0. If you ran this example, once again using the `style` object, it would alter the style of the contents of the first table cell only in the table, as shown in Figure 13-6.

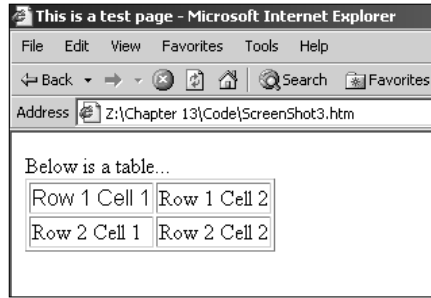


Figure 13-6

Once again, all the attributes of each element are available to the DOM. You can use these to alter any aspect of the element, from presentation to the actual links contained. If you wanted to reference all the cells in this way, you would have to mention each one explicitly in the code, and assign a new variable for each element as follows:

```
<script type="text/javascript">
  var TDElement0 = document.getElementsByTagName("td").item(0);
  var TDElement1 = document.getElementsByTagName("td").item(1);
  var TDElement2 = document.getElementsByTagName("td").item(2);
  var TDElement3 = document.getElementsByTagName("td").item(3);
  TDElement0.style.fontFamily = "arial";
  TDElement1.style.fontFamily = "arial";
  TDElement2.style.fontFamily = "arial";
  TDElement3.style.fontFamily = "arial";
</script>
```

One thing to note about the `getElementsByTagName()` method is that it takes the element names within quotation marks and without the angle brackets `<>` that normally surround tags.

Where the DOM and BOM Overlap

One quick point to consider here is that in the previous set of examples you've used a feature that you introduced in the previous chapter under the heading of the Browser Object Model to access the style properties of an element, namely the `style` object. However, you're still using part of the DOM. This is a common point of confusion, because although the DOM is concerned only with the contents of the browser window, the BOM concerns itself with some features inside the browser window as well as the different parts of the actual browser. This overlap inside the browser window, where both object models can be used, is where things aren't quite so clear. It happened because browsers had object models for the contents of the document long before there were standards outlining them. Style is one major area in which browsers have supported scripting properties and methods, which is definitely part of the document, but currently still browser-dependent.

Given this information, the `style` object (discussed in Chapter 12) might appear to be part of the Browser Object Model and not the Document Object Model because it is browser-dependent. However, although the `style` object isn't addressed in level 1 of the DOM, it isn't because the `style` object is non-standard. The `style` object isn't covered in the standard because the first version of the standard left styles to the second level. They're not totally resolved there either, and it will probably be the third

version in which they get properly sorted. Though it is browser-dependent, the `style` object actually works well in both browsers and supplies a very similar set of properties, which is why we're using it for this example. You could also use the DOM method `setAttribute()` (which you will look at shortly) to set the `style` attributes, but this is a lot messier and currently works only in Firefox.

Properties of the document Object: Returning a Reference to the Topmost Element

You've now got a reference to individual elements on the page, but what about the tree structure mentioned earlier? The tree structure encompasses all the elements and nodes on the page and gives them a hierarchical structure. If you want to reference that structure, you need a particular property of the `document` object that returns the outermost element of your document. In HTML, this should always be the `<html/>` element. The property that returns this element is `documentElement`, as shown in the following table.

Property of the document Object	Description
<code>documentElement</code>	Returns a reference to the outermost element of the document (the root element, for example <code><html/></code>)

You can use `documentElement` as follows. If you go back to the previous example code, you can transfer your entire DOM into one variable like this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>example</title>
</head>
<body>
  <h1 id="Heading1">My Heading</h1>
  <p id="Paragraph1">This is some text in a paragraph</p>
  <script type="text/javascript">
    var Container = document.documentElement;
  </script>
</body>
</html>
```

The variable `Container` now contains the root element, which is `<html/>`. The `documentElement` property has returned a reference to this element in the form of an object, an `Element` object to be precise. The `Element` object has its own set of properties and methods. If you want to use them, you can refer to them by using the variable name, followed by the method or property name.

```
Container.elementObjectProperty
```

Fortunately, the `Element` object has only one property.

Property of the Element Object

The property of the `Element` object is a reference to the tag name of the element, as shown in the following table.

Property of the Element Object	Description
<code>tagName</code>	Can return the element tag name

In the previous example the variable `Container` contained the `<html/>` element, but using this property you can demonstrate that property. Add the following highlighted line, which makes use of the `tagName` property.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>example</title>
</head>
<body>
  <h1 id="Heading1">My Heading</h1>
  <p id="Paragraph1">This is some text in a paragraph</p>
  <script type="text/javascript">
    var Container = document.documentElement;
    alert(Container.tagName);
  </script>
</body>
</html>
```

This code will now return proof that your variable `Container` holds the outermost element, and by implication all other elements within (see Figure 13-7).

Now that you can return any individual element, and the root element, you can look at how you can start navigating your tree structure.



Figure 13-7

Properties of the Node Object: Navigating the Document

You now have your element or elements from the web page, but what happens if you want to move through your page systematically, from element to element, or from attribute to attribute? This is where you need to step back to a higher level. To move among elements, attributes, and text, you have to move among nodes in your tree structure. It doesn't matter what is contained within the node, or rather, what sort of node it is. This is why you need to go back to one of the objects you called base objects. Your whole tree structure is made up of these base-level `Node` objects.

Following is a list of some common properties of the `Node` object that provide information about the node—whether it is an element, attribute, or text—and enable you to move from one node to another.

Chapter 13: Dynamic HTML in Modern Browsers

Properties of the Node Object	Description of Property
firstChild	Returns the first child node of an element
lastChild	Returns the last child node of an element
previousSibling	Returns the previous child node of an element at the same level as the current child node
nextSibling	Returns the next child node of an element at the same level as the current child node
ownerDocument	Returns the root node of the document that contains the node (note this is not available in IE 5 or 5.5)
parentNode	Returns the element that contains the current node in the tree structure
nodeName	Returns the name of the node
nodeType	Returns the type of the node as a number
nodeValue	Sets the value of the node in plain text format

Let's take a quick look at how some of these properties work. Consider once more the first example.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>example</title>
</head>
<body>
  <h1 id="Heading1">My Heading</h1>
  <p id="Paragraph1">This is some text in a paragraph</p>
  <script type="text/javascript">
    var H1Element = document.getElementById("Heading1");
    H1Element.style.fontFamily = "Arial";
  </script>
</body>
</html>
```

You can now use `H1Element` to navigate your tree structure, access the contents of the text, and change it. Note that Firefox, Opera, and Safari count all instances of whitespace as child nodes; therefore, this example won't work in them. If you add the following lines, you are setting the reference in the variable `PElement` to the next element in the tree structure on the same level.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>example</title>
```

```
</head>
<body>
  <h1 id="Heading1">My Heading</h1>
  <p id="Paragraph1">This is some text in a paragraph</p>
  <script type="text/javascript">
    var H1Element = document.getElementById("Heading1");
    H1Element.style.fontFamily = "Arial";
    var PElement = H1Element.nextSibling;
    PElement.style.fontFamily = "Arial";
  </script>
</body>
</html>
```

In effect, you are navigating through the tree structure as shown in Figure 13-8.

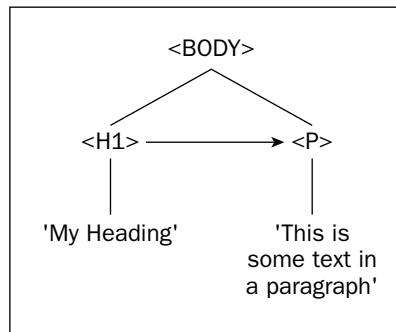


Figure 13-8

The same principles also work in reverse. You can go back and add yet more code that navigates back to the previous node and changes the text of your previous element to your example.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
</head>
<body>
  <h1 id="Heading1">My Heading</h1>
  <p id="Paragraph1">This is some text in a paragraph</p>
  <script type="text/javascript">
    var H1Element = document.getElementById("Heading1");
    H1Element.style.fontFamily = "Arial";
    var PElement = H1Element.nextSibling;
    PElement.style.fontFamily = "Arial";
    H1Element = PElement.previousSibling;
    H1Element.style.fontFamily = "Courier";
  </script>
</body>
</html>
```

Chapter 13: Dynamic HTML in Modern Browsers

What you're doing here is setting the first `<h1/>` element to the font Arial; you're then navigating across to the next sibling, which is the next child node of your `<body/>` element. The first child is the `<h1/>` element; the second one is the `<p/>` element. Note that with Firefox, Opera, and Safari things are different; there are lots of whitespace text nodes as child nodes in the body between the element nodes.

You set the font to Arial here as well. Your new two lines of code then use the `previousSibling` property to jump back to your `<h1/>` element, and then you again change the `fontFamily` style, but this time you change it to Courier. So the sum effect of your program is to change the `<h1/>` element to Courier, and the `<p/>` element to Arial.

Try It Out Navigating Your HTML Document Using the DOM

Up until now you've been cheating, because you haven't truly navigated your HTML document. You've just used `document.getElementById()` to return an element and navigated to different nodes from there. Now let's use the `documentElement` property of the `document` object and do this properly. You'll start at the top of your tree and move down through the child nodes to get at those elements; then you'll navigate through your child nodes and change the properties in the same way as before.

Type the following into your text editor:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>NavLast</title>
</head>
<body>
  <h1 id="Heading1">My Heading</h1>
  <p id="Paragraph1">This is some text in a paragraph</p>

  <script type="text/javascript">
var htmlElement;      // htmlElement stores reference to <html>
var headingElement;   // headingElement stores reference to <head>
var bodyElement;      // bodyElement stores reference to <body>
var h1Element;        // h1Element stores reference to <h1>
var pElement;         // pElement stores reference to <p>

htmlElement = document.documentElement;
headingElement = htmlElement.firstChild;

alert(headingElement.tagName);

if (headingElement.nextSibling.nodeType == 3)
{
  bodyElement = headingElement.nextSibling.nextSibling;
}
else
{
  bodyElement = headingElement.nextSibling;
}
```

```

alert(bodyElement.tagName);

if (bodyElement.firstChild.nodeType == 3)
{
h1Element = bodyElement.firstChild.nextSibling;
}
else
{
h1Element = bodyElement.firstChild;
}

alert(h1Element.tagName);
h1Element.style.fontFamily = "Arial";

if (h1Element.nextSibling.nodeType == 3)
{
pElement = h1Element.nextSibling.nextSibling;
}
else
{
pElement = h1Element.nextSibling;
}

alert(pElement.tagName);
pElement.style.fontFamily = "Arial";

if (pElement.previousSibling.nodeType==3)
{
h1Element = pElement.previousSibling.previousSibling
}
else
{
h1Element = pElement.previousSibling
}

h1Element.style.fontFamily = "Courier"
</script>

</body>
</html>

```

Save this as `navlast.htm`. Then open the page in your browser (this example works in IE, Firefox, Opera, and Safari), clicking OK in each of the message boxes until you see the page shown in Figure 13-9.

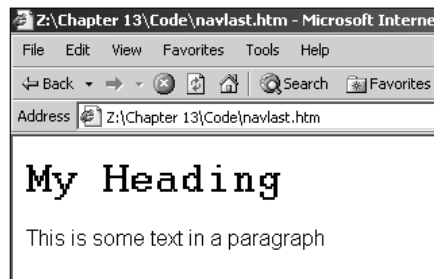


Figure 13-9

How It Works

You've hopefully made this example very transparent by adding several alerts to demonstrate where you are along each section of the tree. You've also named the variables with their various elements, to give a clearer idea of what is stored in each variable. (You could just as easily have named them *a*, *b*, *c*, *d*, and *e*, so don't think you need to be bound by this naming convention.)

You start at the top of the script block by retrieving the whole document using the `documentElement` property.

```
var htmlElement = document.documentElement;
```

The root element is the `<html/>` element, hence the name of your first variable. Now if you refer to your tree, you'll see that the HTML element must have two child nodes: one containing the `<head/>` element and the other containing the `<body/>` element. You start by moving to the `<head/>` element. You get there using the `firstChild` property of your Node object, which contains your `<html/>` element. You use your first alert to demonstrate that this is true.

```
alert(headingElement.tagName);
```

Your `<body/>` element is your next sibling across from the `<head/>` element, so you navigate across by creating a variable that is the next sibling from the `<head/>` element.

```
if (headingElement.nextSibling.nodeType == 3)
{
    bodyElement = headingElement.nextSibling.nextSibling;
}
else
{
    bodyElement = headingElement.nextSibling;
}

alert(bodyElement.tagName);
```

Here you check to see what the `nodeType` of the `nextSibling` of `headingElement` is. If it returns 3, you set `bodyElement` to be the `nextSibling` of the `nextSibling` of `headingElement`; otherwise you just set it to be the `nextSibling` of `headingElement`. Why do you do this? The answer lies with the implementation of the DOM in Firefox, Opera, and Safari. You would think that the next sibling of the `<head/>` element in the tree would be the `<body/>` element, and indeed it is in IE 6+. However, Firefox, Opera, and Safari count all instances of whitespace as nodes in the document. To navigate through the tree in a way that works in both browsers, you check to see whether the next sibling has a `nodeType` of 3. If it does, it is a text node, and you need to move along the next node.

You use an alert to prove that you are now at the `<body/>` element.

```
alert(bodyElement.tagName);
```

The `<body/>` element in this page also has two children, the `<h1/>` and `<p/>` elements. Using the `firstChild` property, you move down to the `<h1/>` element. Again you check whether the child node is whitespace for Firefox, Opera, and Safari. You use an alert again to show that you have arrived at `<h1/>`.

```
if (bodyElement.firstChild.nodeType == 3)
{
    h1Element = bodyElement.firstChild.nextSibling;
}
else
{
    h1Element = bodyElement.firstChild;
}

alert(h1Element.tagName);
```

After the third alert, the style will be altered on your first element, changing the font to Arial.

```
h1Element.style.fontFamily = "Arial";
```

You then navigate across to the `<p/>` element using the `nextSibling` property, again checking for whitespace.

```
if (h1Element.nextSibling.nodeType == 3)
{
    pElement = h1Element.nextSibling.nextSibling;
}
else
{
    pElement = h1Element.nextSibling;
}

alert(pElement.tagName);
```

You change the `<p/>` element's font to Arial also.

```
pElement.style.fontFamily = "Arial";
```

Finally, you use the `previousSibling` property to move back in your tree to the `<h1/>` element and this time change the font to Courier.

```
if (pElement.previousSibling.nodeType==3)
{
    h1Element = pElement.previousSibling.previousSibling
}
else
{
    h1Element = pElement.previousSibling
}

h1Element.style.fontFamily = "Courier";
```

This is a fairly easy example to follow because you're using the same tree structure you created with diagrams, but it does show how the DOM effectively creates this hierarchy and that you can move around within it using script.

Methods of the Node Object: Adding and Removing Elements from the Document

You can now move through your tree structure and alter the contents of elements as you go, but you still can't fundamentally alter the structure of your HTML document. Help is at hand, though, because the `Node` object's methods let you do this.

Following is a list of methods that enable you to alter the structure of an HTML document by creating new nodes and adding them to your tree.

Methods of the Node Objects	Description
<code>appendChild(newNode)</code>	Adds a new <code>node</code> object to the end of the list of child nodes. This method returns the appended node.
<code>cloneNode(cloneChildren)</code>	Returns a duplicate of the current node. It accepts a Boolean value. If the value is <code>true</code> , then the method clones the current node and all child nodes. If the value is <code>false</code> , only the current node is cloned and child nodes are left out of the clone.
<code>hasChildNodes()</code>	Returns <code>true</code> if a node has any child nodes.
<code>insertBefore(newNode, referenceNode)</code>	Inserts a new <code>node</code> object into the list of child nodes before the node stipulated by <code>referenceNode</code> . Returns the inserted node.
<code>removeChild(childNode)</code>	Removes a child node from a list of child nodes of the <code>node</code> object. Returns the removed node.
<code>replaceChild(newChild, oldChild)</code>	Replaces the old child <code>node</code> object with a new child <code>node</code> object. Returns the replaced node.

You'll look at how they work shortly.

Methods of the document Object: Adding and Removing Elements from the Document

In addition to the methods of the `Node` object listed previously, the `document` object itself boasts some methods for creating elements, attributes, and text, shown in the following table.

Methods of the document Object	Description
<code>createElement(elementName)</code>	Creates an element node with a specified name. Returns the created element.
<code>createTextNode(text)</code>	Creates and returns a text node with the specified text.
<code>createAttribute(attributeName)</code>	Creates an attribute node with a specified name. Returns the created attribute node.

The best way to demonstrate both sets of methods for the `Node` object and the `document` object at one time is with an example.

Try It Out Create HTML Elements and Text with DOM JavaScript

You'll create a web page with just paragraph `<p/>` and heading `<h1/>` elements, but instead of HTML you'll use the DOM properties and methods to place these elements on the web page. Figure 13-10 shows IE. However, you don't have to make any changes for the example to work equally well in Firefox. Now start up your preferred text editor and type in the following:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Creating Nodes</title>
</head>
<body>
  <script type="text/javascript">
    var newText = document.createTextNode("My Heading");
    var newElem = document.createElement("h1");

    newElem.appendChild(newText);
    document.body.appendChild(newElem);

    newText = document.createTextNode("This is some text in a paragraph");
    newElem = document.createElement("p");

    newElem.appendChild(newText);
    document.body.appendChild(newElem);
  </script>
</body>
</html>
```

Save this page as `create.htm` and open it in a browser.

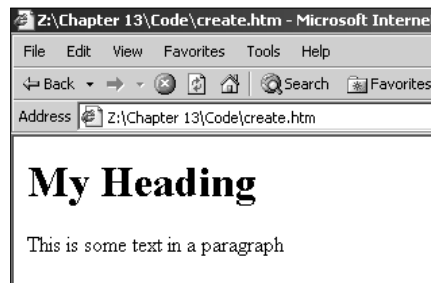


Figure 13-10

How It Works

It all looks a bit dull and ordinary, doesn't it? And yes, you could have done this much more simply with HTML. That isn't the point, though. The idea is that you use DOM properties and methods, accessed

Chapter 13: Dynamic HTML in Modern Browsers

with JavaScript, to insert these features. The first two lines of the script block are used to define the variables in your script, which are initialized to hold the text you want to insert into the page and the HTML element you wish to insert.

```
var newText = document.createTextNode("My Heading");
var newElem = document.createElement("h1");
```

You start at the bottom of your tree first, by creating a text node with the `createTextNode()` method. Then use the `createElement()` method to create an HTML heading.

At this point the two variables are entirely separate from each other. You have a text node, and you have an `<h1/>` element, but they're not connected. The next line enables you to attach the text node to your HTML element. You reference the HTML element you have created with the variable name `newElem`, use the `appendChild()` method of your node, and supply the contents of the `newText` variable you created earlier as a parameter.

```
newElem.appendChild(newText);
```

Let's recap. You created a text node and stored it in the `newText` variable. You created an `<h1/>` element and stored it in the `newElem` variable. Then you appended the text node as a child node to the `<h1/>` element. That still leaves you with a problem: You've created an element with a value, but the element isn't part of your document. You need to attach the entirety of what you've created so far to the document body. Again, you can do this with the `appendChild()` method, but this time supply it to the `document.body` object.

```
document.body.appendChild(newElem);
```

This completes the first part of your code. Now all you have to do is repeat the process for the `<p/>` element.

```
newText = document.createTextNode("This is some text in a paragraph");
newElem = document.createElement("p");

newElem.appendChild(newText);
document.body.appendChild(newElem);
```

You create a text node first; then you create an element. You attach the text to the element, and finally you attach the element and text to the body of the document. This completes the creation of parts of the HTML document in script.

You've now created elements and changed text, but you've left out one of the important parts of the web page, namely attributes. You'll look at how to use attributes now.

Methods of the Element Object: Getting and Setting Attributes

Although it is still acceptable to set the `style` attributes through the `style` object, if you want to set any other element attributes, you should use the DOM-specific methods of the `Element` object.

The three methods you can use to return and alter the contents of an HTML element's attributes are `getAttribute()`, `setAttribute()`, and `removeAttribute()`, as shown in the following table.

Methods of the Element Object	Description
<code>getAttribute(attributeName)</code>	Returns the value of an attribute
<code>setAttribute(attributeName, value)</code>	Sets the value of an attribute
<code>removeAttribute(attributeName)</code>	Removes the value of an attribute and replaces it with the default value

Let's take a quick look at how these methods work now. In the previous example, `createElement()` and `createTextNode()` were used to add HTML elements and text to your page, but you didn't actually make use of `createAttribute()`. That's because there's a much easier method of creating attributes: using the `setAttribute()` and `getAttribute()` methods.

Try It Out Creating Attributes

You're now going to take your previous example and add some attributes to it that will affect the presentation and layout of your text. You must be sure to replicate the case of these lines when you type them because incorrect case will prevent the example from working correctly.

Open `create.htm` in a text editor and add the following highlighted lines:

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Creating Attributes</title>
</head>
<body>

  <script type="text/javascript">
    var newText = document.createTextNode("My Heading");
    var newElem = newElem = document.createElement("h1");

    newElem.setAttribute("align", "center");
    newElem.appendChild(newText);
    document.body.appendChild(newElem);

    newText = document.createTextNode("This is some text in a paragraph");
    newElem = document.createElement("p");

    alert(newElem.getAttribute("align"));
    newElem.setAttribute("align", "right");
    alert(newElem.getAttribute("align"));

    newElem.appendChild(newText);
    document.body.appendChild(newElem);
  </script>
</body>
</html>

```

Save this as `attribute.htm` and open it in a browser. Your page will look like what is shown in Figure 13-11.



Figure 13-11

Click OK on the alert, and the second time it should display correctly (see Figure 13-12).



Figure 13-12

Click OK again to reach the screen shown in Figure 13-13.

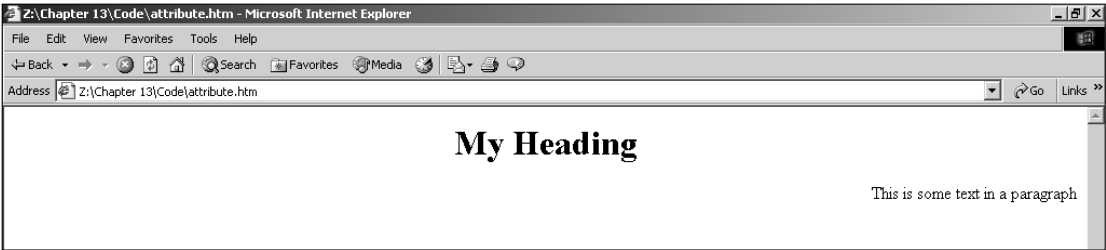


Figure 13-13

How It Works

You’ve added four lines of code here to augment your existing document structure. The first takes your `<h1/>` element and adds an `align` attribute to it.

```
var newText = document.createTextNode("My Heading");
var newElem = newElem = document.createElement("h1");
```

```
newElem.setAttribute("align", "center");
```

Note that in the `setAttribute()` method, `align` is in lowercase per the XHTML standards recommended earlier in this chapter. If it were in uppercase, as follows, the example would fail in some browsers (such as IE).

```
newElem.setAttribute("ALIGN", "center");
```

The `setAttribute()` method takes `align`, an existing attribute (meaning that it is specified for that particular element in the HTML specifications), and supplies it with the value `center`. The result positions your text in the center of the page.

Strictly speaking, the `align` attribute is deprecated under XHTML, but you have used it because it works and because it has one of the most easily demonstrable visual effects on a web page.

So `setAttribute()` takes the name of the attribute first and the value second. If you set the attribute name to be a nonexistent attribute, it will have no effect on the page. You can also set the attribute to a nonexistent value, which is perfectly legal, but once again there will be no effect on the page.

The second part of the code is very similar to the first.

```
newText = document.createTextNode("This is some text in a paragraph");
newElem = document.createElement("p");
```

```
alert(newElem.getAttribute("align"));
newElem.setAttribute("align", "right");
alert(newElem.getAttribute("align"));
```

After you've created the `<p/>` element and the accompanying text, you use `getAttribute()` to return the value of the `align` attribute. Because `align` hasn't been set yet, it returns no value, not even the default. You then use the `setAttribute()` method to set the `align` attribute to `right`, and use `getAttribute()` to return the value. This time it returns `right`. This is then reflected in the final display of the web page.

What have you seen so far? You started with a nearly empty DOM hierarchy. You then returned the HTML document to a variable, navigated through the different parts of it via DOM objects outside the hierarchy itself (the `Node` objects), and changed the contents of objects, thus altering the content of the web page. Then you inserted DOM objects into the hierarchy, thus inserting new elements into the page. This leaves just one area of the DOM to cover: the event model.

The DOM Event Model

The DOM event model is introduced in level 2 of the DOM standard. It's a way of handling events and providing information about these events to the script. It provides guidelines for a standard way of determining what generated an event, what type of event it was, and when and where the event occurred.

Chapter 13: Dynamic HTML in Modern Browsers

All of this was, of course, trackable in earlier versions of IE, and to a lesser extent in Netscape, through the `event` object as you saw in the last chapter. However, the main problem was that the ways of accessing this object and the names of its properties were completely different between the two browsers.

The DOM event model doesn't look complete in some ways, and might yet be tweaked in level 3 of the standard, but what it does do is introduce a basic set of objects, properties, and methods. It also makes some important distinctions.

First there is an `event` object, which provides information about the element that has generated an event and enables you to retrieve it in script. If you want to make it available in script, it must be passed as a parameter to the function connected to the event handler, as you saw in the previous chapter. It is not globally available, as the IE `event` object is.

The standard outlines several properties and methods of the `event` object that have long since been a source of dispute between IE and other browsers. You will be using only the properties, so here you will be considering them alone.

Properties of the event Object	Description
<code>bubbles</code>	Indicates whether an event can bubble (pass control from one element to another)
<code>cancelable</code>	Indicates whether an event can have its default action canceled
<code>currentTarget</code>	Indicates which event is currently being processed
<code>eventPhase</code>	Indicates which phase of the event flow an event is in
<code>target</code> (DOM browsers only)	Indicates which element caused the event; in the DOM event model, text nodes are the possible target of an event
<code>timeStamp</code> (Firefox only)	Indicates at what time the event occurred
<code>type</code>	Indicates the name of the event

Secondly, the DOM introduces a `MouseEvent` object, which deals with events generated specifically by the mouse. This is useful because you might need more specific information about the event, such as the position in pixels of the cursor, or the element the mouse has come from.

Properties of the mouse Event Object	Description
<code>altKey</code>	Indicates whether the Alt key was pressed when the event was generated.
<code>button</code>	Indicates which button on the mouse was pressed.
<code>clientX</code>	Indicates where in the browser window, in horizontal coordinates, the mouse pointer was when the event was generated.

Properties of the mouse Event Object	Description
<code>clientY</code>	Indicates where in the browser window, in vertical coordinates, the mouse pointer was when the event was generated.
<code>ctrlKey</code>	Indicates whether the Ctrl key was pressed when the event was generated.
<code>metaKey</code>	Indicates whether the meta key was pressed when the event was generated.
<code>relatedTarget</code> (DOM browsers only)	In the DOM event model text nodes are the (possible) target of the <code>mouseover</code> event. This object is similar to IE's <code>event.toElement</code> and <code>event.fromElement</code> .
<code>screenX</code>	Indicates where in the browser window, in horizontal coordinates relative to the origin in the screen coordinates, the mouse pointer was when the event was generated.
<code>screenY</code>	Indicates where in the browser window, in vertical coordinates relative to the origin in the screen coordinates, the mouse pointer was when the event was generated.
<code>shiftKey</code>	Indicates whether the Shift key was pressed when the event was generated.

Although any event might create an event object, only a select set of events can generate a `MouseEvent` object. On the occurrence of a `MouseEvent` event, you'd be able to access properties from the event object and the `MouseEvent` object. With a non-mouse event, none of the mouse object properties in the preceding table would be available. The following mouse events can create a mouse event object:

- ☐ `click` occurs when a mouse button is clicked (pressed and released) with the pointer over an element or text
- ☐ `mousedown` occurs when a mouse button is pressed with the pointer over an element or text
- ☐ `mouseup` occurs when a mouse button is released with the pointer over an element or text
- ☐ `mouseover` occurs when a mouse button is moved onto an element or text
- ☐ `mousemove` occurs when a mouse button is moved and it is already on top of an element or text
- ☐ `mouseout` occurs when a mouse button is moved out and away from an element or text

To get at an event using the DOM, all you have to do is query the event object created by the individual element that raised the event. For example, in the following code the `<p/>` element will raise a `dblclick` event:

```
<p ondblclick="handle(event)">Paragraph</p>

<script type="text/javascript">
function handle(e)
```

```
{
    alert(e.type);
}
</script>
```

You have to pass the event object created by the `<p/>` element as an argument in the function call to be able to use it within the function. You can then use the parameter passed as the event object and use its general properties made available through the DOM.

However, this is where the browsers get shaky. IE 6 and 7 do not support many of the DOM properties mentioned in the table. Instead, IE has its own set of IE-specific properties, which we won't be discussing here.

If you ran the previous example, it would just tell you what kind of event raised your event-handling function. This might seem self-evident in the preceding example, but if you had included the following extra lines of code, any one of three elements could have raised the function:

```
<p ondblclick="handle(event)">Paragraph</p>
<h1 onclick="handle(event)">Heading 1</h1>
<span onmouseover="handle(event)">Special Text</span>

<script type="text/javascript">
function handle(e)
{
    alert(e.type);
}
</script>
```

This makes the code much more useful. In general you will use relatively few event handlers to deal with any number of events, and you can use the event properties as a filter to determine what type of event happened and what HTML element triggered it, so that you can treat each event differently.

In the following example, you see that you can take different courses of action depending on what type of event is returned:

```
<p ondblclick="handle(event)">Paragraph</p>
<h1 onclick="handle(event)">Heading 1</h1>
<span onmouseover="handle(event)">Special Text</span>
...
<script type="text/javascript">
function handle(e)
{
    if (e.type == "mouseover")
    {
        alert("You moved over the Special Text");
    }
}
</script>
```


Try It Out Using the DOM Event Model

Let's take a quick look at an example that creates a mouse event object when the user clicks anywhere on the screen and returns to the user the x and y coordinates of the position of the mouse pointer when the mouse button was clicked (see Figure 13-14). This example will work in IE 5+ and Firefox.

Open a text editor and type in the following:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
</head>
<body onclick="handleClick(event)">
    Click anywhere on the screen if you're using Firefox.
    Click on this text if you're using IE, Opera, or Safari.

    <script language="JavaScript">
        function handleClick(e)
        {
            alert("You clicked on the screen at X" + e.clientX + " and Y" + e.clientY);
        }
    </script>
</body>
</html>
```

Save this as `mouseevent.htm` and run it in your browser.

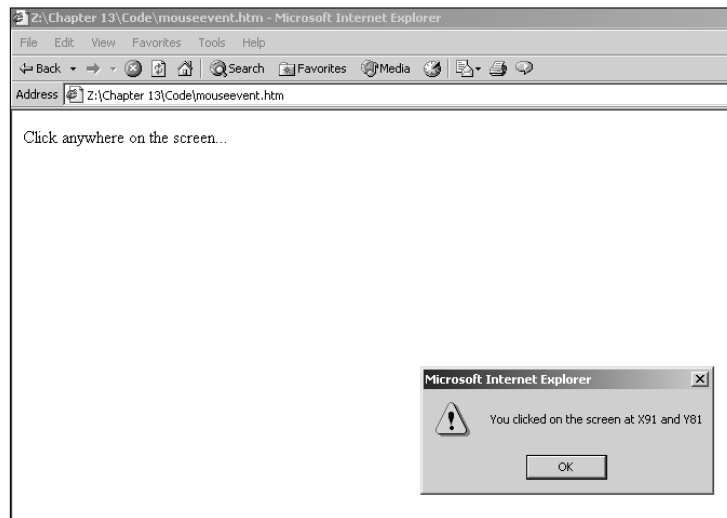


Figure 13-14

Now click OK, move the pointer in the browser window, and click again. A different result appears.

How It Works

This example is consistent with the event-handling behavior: The browser waits for an event, and every time that event occurs it raises the corresponding function. It will continue to wait for the event until you exit the browser or that particular web page. In this example, you use the `<body/>` element to raise a `click` event.

```
<body onclick="handleClick(event)">
```

Whenever that function is encountered, the `handleClick()` function is raised and a new `MouseEvent` object is generated. Remember that `MouseEvent` objects give you access to event object properties as well as `MouseEvent` object properties, even though you don't use them in this example.

The function takes the mouse event object and assigns it the reference `e`.

```
function handleClick(e)
{
    alert("You clicked on the screen at X" + e.clientX + " and Y" + e.clientY);
}
```

Inside the function you use the `alert()` statement to display the contents of the `clientX` and `clientY` properties of the mouse event object on the screen. This `MouseEvent` object is overwritten and re-created every time you generate an event, so the next time you click the mouse pointer it returns new coordinates for the `x` and `y` positions.

As described earlier, one problem that precludes greater discussion of the DOM event model is the fact that not all browsers support it in any detail. Specifically, IE, the most popular browser by far, doesn't fully support it. You will see in a later example that you are still in a position of having to code for both browsers when it comes to returning information about events via properties, but this should provide a little taste of how they work, and how they will work in future browsers. We won't go into any further detail here about events, but you will be returning to the DOM event object model later in the chapter in an example that discusses how to cross-code.

Despite advances in standards compliance, there are still differences among the major browsers that require you to use browser-specific code. In the following sections you'll walk through the creation of a DHTML toolbar for Internet Explorer. You'll then adapt that script for Firefox, and finally you'll make it cross-browser.

DHTML Example: Internet Explorer 5+

We've spent a few pages on the DOM event model, and although this chapter is primarily focused on the web standards, we should briefly cover the IE event model.

The IE Event Model

DHTML under IE provides the `event` object, which we briefly discussed in the previous chapter. This object is a property of the `window` object, and one such object exists for each open browser window. It provides you with lots of very useful information about the most recent event to fire, such as details

regarding the event that was fired, which element caused the event to fire, information about where the user's mouse pointer is, which buttons or keys were pressed, and much, much more.

The `event` object has a lot of properties and methods, but you are going to concentrate on only a few of the more useful ones. You'll take a look at these in the following sections.

Why Did the Event Fire?

The `event` object enables you to find information about what the user did to cause the event to fire. For example, you can check to see where the mouse pointer is, which mouse buttons have been clicked, and which keys have been pressed.

The properties `screenX` and `screenY` of the `event` object provide you with the coordinates of the user's mouse pointer at the time of the event. Because the `event` object is already created for you, you simply use it by typing its name followed by the property or method you're interested in.

```
event.screenX
```

The preceding entry gets you a reference to the horizontal position, in pixels, of the user's mouse pointer.

Contrast this with the following code:

```
event.screenY
```

This gets you a reference to the vertical position, in pixels, of the mouse pointer. Remember that screen coordinates start at 0, 0 in the top left-hand corner of the screen.

The `button` property of the `event` object tells you which mouse button, if any, the user has pressed. It returns a number, between 0 and 7 (inclusive), indicating the following:

- ☐ 0: No button is pressed.
- ☐ 1: Left button is pressed.
- ☐ 2: Right button is pressed.
- ☐ 3: Left and right buttons are both pressed.
- ☐ 4: Middle button is pressed.
- ☐ 5: Left and middle buttons are both pressed.
- ☐ 6: Right and middle buttons are both pressed.
- ☐ 7: All three buttons (left, right, and middle) are pressed.

Finally, you can find out whether a key has been pressed and, if so, which key it was by using the `keyCode` property of the `event` object. This returns a number that indicates the Unicode character code corresponding to the key pressed. If no key was pressed, the number returned will be 0.

Finding Out Which Elements Are Involved in the Event

The event object has three properties: `fromElement`, `srcElement`, and `toElement`. These properties provide a reference to the objects of the elements involved in the event. For example, in a `mouseover` event the following conditions apply:

- ❑ The `fromElement` property will refer to the object of whichever element the user's mouse pointer was on before moving over the element that fired the `onmouseover` event.
- ❑ The `srcElement` property will be the object of the element that caused the event to fire. For example, if the mouse pointer just rolled over an image, the `srcElement` will be that `` element's object.
- ❑ The `toElement` property is the object of the element that the user's mouse pointer is about to move to, for example in an `mouseout` event. How does the browser know where the mouse is going? Well, it raises the events for a particular user action after they have actually happened, although the delay is only microseconds. It's rather like a live TV broadcast that is actually transmitted with a short delay so that swear words can be bleeped out.

Each of these properties provides the actual object of the element, which you can use and manipulate as if you had referenced the element directly. This ability can be particularly useful when you're writing generic code that is placed in the event of a higher object, such as `document`.

Building a DHTML Toolbar

The DHTML script you'll write in this section will be a toolbar, a UI element that is used often in applications. The Back, Forward, Stop, and Home buttons in your web browser are good examples. The toolbar you're going to build will consist only of icons (so no text), and clicking a button will either run JavaScript code or take the user to another web page.

You'll generate the HTML dynamically using DOM methods. So before you get too deep into the code, let's look at what your generated HTML will look like.

The HTML Structure

The toolbar itself is nothing more than a `<div/>` element, and it will contain any number of buttons.

```
<div class="toolbar">
  <span href="go_someplace" class="toolbar-button">
    
  </span>
</div>
```

The buttons, too, are `` elements. Buttons have a CSS class of `toolbar-button`, and each contain an `href` attribute. The `` element does not have this attribute in the HTML specification, so the browser will ignore the attribute and its value when it displays the HTML. You can, however, access it with JavaScript and program the desired functionality by using the `click` event. When the user moves her mouse over a button, you'll change the CSS class name to `toolbar-button-hover`. This will enable you to easily change the style of the button. So to clarify, you'll need to handle the `mouseover`, `mouseout`, and `click` events.

Inside each button is an `` element, which serves as your icon. These have a CSS class of `toolbar-icon`, and you'll constrain the size of the image through CSS. Speaking of which, you should go over how you want to style your toolbar.

Adding Style

You want this toolbar to resemble the UI constructs people are accustomed to seeing. This is important because you want the user to know how to use the toolbar without any instruction. If she recognizes the toolbar, she'll know what to do.

Styling the Toolbar

Your toolbar will have a grayish background color.

```
.toolbar
{
    background-color: #E4E2D5;
    padding: 2px;
}
```

You'll also add some padding around the edges. It's obvious that the buttons are contained inside the toolbar; however, adding some padding will emphasize that fact, and give the toolbar a clean look.

Styling the Buttons

As stated earlier, the buttons are `` elements, and they have two states: a default state, and a state when the mouse pointer hovers over the button. Therefore, you'll need at least two rules, each with their own properties and values.

The first rule is the default state of the button; in other words, this is what it looks like before a mouse pointer hovers over the button.

```
.toolbar-button
{
    display: inline-block;
    height: 24px;
    width: 24px;
    padding: 3px;
}
```

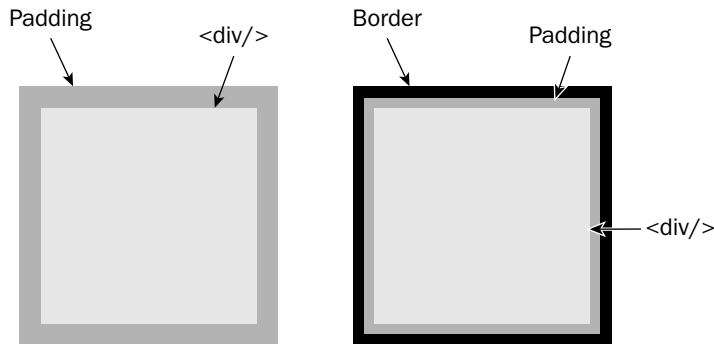
This rule sets the button to be displayed as an `inline-block` element. That may seem like a contradiction, but it is an actual value for this property. This value allows the inside of an element to be formatted as a block level element, but the element itself is formatted as an inline element. Next, the button is set to be 24 pixels in height and width. Lastly, three pixels of padding are added to the button. This padding is added for the same reasons padding was added to the `toolbar` class.

Now, add the style rule for the button's second state: when the mouse hovers over the button.

```
.toolbar-button-hover
{
    display: inline-block;
    height: 24px;
```

```
width: 24px;  
border: 1px solid #316AC5;  
background-color: #C1D2EE;  
padding: 2px;  
cursor: pointer;  
}
```

You've already seen the first three properties; they were used in the `toolbar-button` class. Next, a blue border one pixel in width is added to the element, and the background color is changed to a light blue. The padding is decreased to two pixels. You do this because padding and borders add extra height and width to the HTML element. Take a look at Figure 13-15, which illustrates this concept.



Both elements 30 pixels
in height and width

Figure 13-15

This is a side-by-side comparison of the `toolbar-button` and `toolbar-button-hover` classes. The `toolbar-button` class is 24 pixels in height and width plus three pixels of padding per side. That makes `toolbar-button` 30 pixels in height and width.

The `toolbar-button-hover` class starts with the same 24 pixels in height and width. You then add a one-pixel border, which adds two pixels to the height and width. Then you add two pixels of padding on each side, which makes `toolbar-button-hover` 30 pixels in height and width, just like `toolbar-button`. If you used three pixels of padding instead of two in `toolbar-button-hover`, the button would grow in size when the mouse pointer hovered over it, as Figure 13-16 shows.

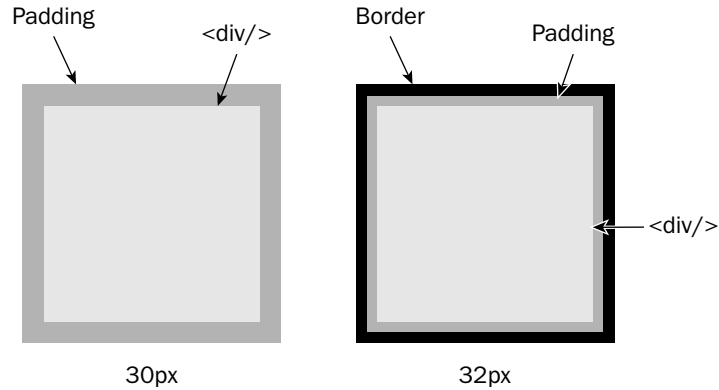


Figure 13-16

Let's take a second and look at these two rules. Notice that each has three properties with the same values (`display`, `height`, and `width`). You can take these properties out and write a new rule for all `` elements inside the toolbar.

```
.toolbar span
{
    display: inline-block;
    height: 24px;
    width: 24px;
}
```

You can now slim down the `toolbar-button` and `toolbar-button-hover` classes:

```
.toolbar-button
{
    padding: 3px;
}

.toolbar-button-hover
{
    border: 1px solid #316AC5;
    background-color: #C1D2EE;
    padding: 2px;
    cursor: pointer;
}
```

By making this change, you can easily add style that is shared by both states.

The last property we'll discuss for the button is the `cursor` property. The mouse cursor (or pointer) is an important user interface component. It can tell the user when something is going on in the background or let him know when he can highlight text. It also changes to a hand when he moves the cursor over a link, letting him know that something will happen when he clicks it.

Chapter 13: Dynamic HTML in Modern Browsers

As stated earlier, you want the user to understand what the toolbar is and what it does. By using the `cursor` property and setting it to `pointer`, you show the user a hand when he moves his mouse over a button. This offers the suggestion “Hey you! You can click me!”

Styling the Icons

The last style rule you need to write is one for the icons. These are simple `` elements with a CSS class name of `toolbar-icon`.

```
.toolbar-icon
{
    height: 24px;
    width: 24px;
}
```

By assigning `height` and `width`, you can constrain the image to a certain size. This will make sure that the icons look uniform.

Storing Button Information

You need some way to store the button information. In this script, you’ll use a multi-dimensional array to contain information for specific buttons. Let’s start with the first array. Let’s call it `myToolbar`.

```
var myToolbar = new Array();
```

Each element in this array, `myToolbar[x]`, will also be declared as an array. Each of these inner arrays will hold information for a particular button.

You start this process with the array element of index 0.

```
myToolbar[0] = new Array();
```

Now you can use the elements of this array, namely `myToolbar[0][0]` and `myToolbar[0][1]`, to hold information about the first button. That information will be the image location for the icon and the page (or JavaScript code) to load when the button is clicked. So you see that the following code holds the location of the icon:

```
myToolbar[0][0] = "img/green.gif";
```

The next line holds the web location, or JavaScript code, to load.

```
myToolbar[0][1] = "javascript: alert('You Clicked the Green Button!')";
```

Each button follows this pattern of defining a new `Array` object and inserting it in the first dimension. Then you make the second dimension of the array hold the icon information and the link or JavaScript code to load.

```
myToolbar[0]    = new Array();
myToolbar[0][0] = "img/green.gif";
myToolbar[0][1] = "javascript: alert('You Clicked the Green Button!')";

myToolbar[1]    = new Array();
```



```
myToolbar[1][0] = "img/blue.gif";
myToolbar[1][1] = "javascript: alert('You Clicked the Blue Button!')";

myToolbar[2]    = new Array();
myToolbar[2][0] = "img/red.gif";
myToolbar[2][1] = "http://www.wrox.com";
```

Building the Toolbar

You need a function to build the toolbar and populate it with buttons. Let's write a function called `createToolbar()` that will do that for you. It needs to accept two arguments: the first one is the name of your toolbar, and the second is your multi-dimensional array containing the button information. You know how the HTML should be structured, so let's get started.

The first step is to dynamically create the `<div/>` element for the toolbar.

```
function createToolbar(sName, aItems)
{
    var toolbar = document.createElement("div");
    toolbar.id   = sName;
    toolbar.className = "toolbar";

    //more code here

    document.body.appendChild(toolbar);
}
```

In this code you create the `<div/>` element with the `createElement()` method and use the toolbar's name, specified by the first argument, as its `id` attribute. You then assign its CSS class, `toolbar`, and append it to the document with the `appendChild()` method.

You now have an empty toolbar, so you need to use the `myToolbar` array to populate it with buttons. You'll do this with a `for` loop.

```
function createToolbar(sName, aButtons)
{
    var toolbar = document.createElement("div");
    toolbar.id   = sName;
    toolbar.className = "toolbar";

    for (var i = 0; i < aButtons.length; i++)
    {
        var thisButton = aButtons[i];

        var button = document.createElement("span");
        var icon = document.createElement("img");

        //more code here
    }

    document.body.appendChild(toolbar);
}
```

Chapter 13: Dynamic HTML in Modern Browsers

Inside the loop, you get the element of the array that corresponds to this button and assign it to `thisButton`. This enables you to easily access the button's information. Then you create the required `` and `` elements. The button variable references the `` element, and the icon variable references the `` element.

Next, add the `href` attribute to button with the `setAttribute()` method. The value of this attribute is contained in the `thisButton[1]` element of the `thisButton` array. Also set the CSS class.

```
function createToolbar(sName, aButtons)
{
    var toolbar = document.createElement("div");
    toolbar.id = sName;
    toolbar.className = "toolbar";

    for (var i = 0; i < aButtons.length; i++)
    {
        var thisButton = aButtons [i];

        var button = document.createElement("div");
        var icon = document.createElement("img");

        button.setAttribute("href", thisButton[1]);
        button.className = "toolbar-button";

        icon.src = thisButton[0];
        icon.className = "toolbar-icon";

        button.appendChild(icon);
        toolbar.appendChild(button);
    }

    document.body.appendChild(toolbar);
}
```

This code also sets the `src` and `className` properties for the icon. You then append icon to button and add the button to the toolbar.

With this code, the toolbar is created and populated with buttons, but it currently has no functionality. Remember, you want hover effects, and you want something to happen when a button is clicked.

Handling User Interaction

User interaction is an important part of DHTML; you usually want your HTML to react to something a user does, and the toolbar is no exception. As already mentioned, there are three areas of user interaction you want to handle:

- ☐ When the user moves her mouse pointer over a button
- ☐ When the user moves her mouse pointer off a button
- ☐ When the user clicks a button

You'll write one function to handle these events: `button_mouseHandler()`.

The `button_mouseHandler()` Function

Using one function to handle the three mouse events is a time- and code-saving measure, especially in the case of this DHTML script. The function begins with its definition and two variables:

```
function button_mouseHandler()
{
    var eType = event.type;
    var eSrc = event.srcElement;

    //more code here
}
```

This DHTML script is quite similar to the image rollover scripts you wrote in the previous chapter. Here, you're concerned only with the element that the event was fired upon (the source element) and the event type that called the event handler. The next step is to write the code for the `mouseover` event.

```
function button_mouseHandler()
{
    var eType = event.type;
    var eSrc = event.srcElement;

    if (eType == "mouseover")
    {
        eSrc.className = "toolbar-button-hover";
    }

    //more code here
}
```

This code checks to see if the event type is a `mouseover` event, and, if so, it changes the source element's `className` property to `toolbar-button-hover`.

Now it's time to handle the `mouseout` event. When the mouse pointer leaves the button, the desired effect is to return the previously highlighted button to its original state. Therefore, the following code changes the `className` property of the source element (of the `mouseout` event) back to `toolbar-button`.

```
function button_mouseHandler()
{
    var eType = event.type;
    var eSrc = event.srcElement;

    if (eType == "mouseover")
    {
        eSrc.className = "toolbar-button-hover";
    }
    else if (eType == "mouseout")
    {
        eSrc.className = "toolbar-button";
    }

    //more code here
}
```

Chapter 13: Dynamic HTML in Modern Browsers

Now things are beginning to take shape. When the mouse pointer moves over the button, its style changes to give a highlight effect, and the mouse pointer leaving the button returns it to its original state. Now you need to write the code to handle the `click` event, and the following code does this:

```
function button_mouseHandler()
{
    var eType = event.type;
    var eSrc = event.srcElement;

    //more code here

    if (eType == "mouseover")
    {
        eSrc.className = "toolbar-button-hover";
    }
    else if (eType == "mouseout")
    {
        eSrc.className = "toolbar-button";
    }
    else if (eType == "click")
    {
        eSrc.className = "toolbar-button";
        window.location.href = eSrc.getAttribute("href");
    }
}
```

The code handling the `click` event does two things. First, it returns the clicked button's `className` property back to `toolbar-button`, and second, it navigates to the desired web page, or executes JavaScript code.

But alas, all is not well. If you were to run this code, you would notice a few weird things happening. Buttons would highlight and unhighlight at strange times, the icons would grow to the size of the buttons, and you'd see some very strange results if you clicked on a button when the mouse pointer was over an icon (the browser would navigate to the URL specified in the `` element's `src` property). These behaviors may seem weird, but they are normal. As the mouse pointer moves over the `` element, it is no longer over the `` element (the button). Therefore, the `mouseout` event fires as the mouse leaves the `` and enters the ``.

A simple solution to this problem is to check the source element's `tagName` property, and, if it's `IMG`, to access the image's parent node: the `` element that represents the button.

```
function button_mouseHandler()
{
    var eType = event.type;
    var eSrc = event.srcElement;

    if (eSrc.tagName == "IMG")
    {
        eSrc = eSrc.parentNode;
    }

    if (eType == "mouseover")
```

```
{
    eSrc.className = "toolbar-button-hover";
}
else if (eType == "mouseout")
{
    eSrc.className = "toolbar-button";
}
else if (eType == "click")
{
    eSrc.className = "toolbar-button";
    window.location.href = eSrc.getAttribute("href");
}
}
```

Now the `eSrc` variable will always reference the `` element, making the button behave as you would expect it to.

Finishing `createToolbar()`

With the mouse event handler written, you can assign it to handle the appropriate events. Do this in `createToolbar()`.

```
function createToolbar(sName, aButtons)
{
    var toolbar = document.createElement("div");
    toolbar.id = sName;
    toolbar.className = "toolbar";

    for (var i = 0; i < aButtons.length; i++)
    {
        var thisButton = aButtons[i];

        var button = document.createElement("span");
        var icon = document.createElement("img");

        button.setAttribute("href", thisButton[1]);
        button.className = "toolbar-button";

        button.onclick = button_mouseHandler;
        button.onmouseover = button_mouseHandler;
        button.onmouseout = button_mouseHandler;

        icon.src = thisButton[0];
        icon.className = "toolbar-icon";

        button.appendChild(icon);
        toolbar.appendChild(button);
    }

    document.body.appendChild(toolbar);
}
```

Now the code for the toolbar is complete. You have the toolbar, you populated it with buttons, and you added interactivity for those buttons. Now you need only to call `createToolbar()`.

Finishing Up

Creating a toolbar is easy; however, there is one caveat you must consider. Since you generate the HTML elements dynamically and append them to `document.body`, you must create the toolbar while the document is loading, or after the document is loaded. If you attempt to load the toolbar at any other time, you'll get errors in your page.

In this exercise, you'll use the `onload` event handler to create the toolbar after the document is loaded. Following is the complete source code for the toolbar DHTML script. Open the text editor of your choice and type the following:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>IE Toolbar</title>
  <style type="text/css">
    .toolbar
    {
      background-color: #E4E2D5;
      padding: 2px;
    }

    .toolbar span
    {
      display: inline-block;
      height: 24px;
      width: 24px;
    }

    .toolbar-button
    {
      padding: 3px;
    }

    .toolbar-button-hover
    {
      border: 1px solid #316AC5;
      background-color: #C1D2EE;
      padding: 2px;
      cursor: pointer;
    }

    .toolbar-icon
    {
      height: 24px;
      width: 24px;
    }
  </style>
  <script type="text/javascript">
    function button_mouseHandler()
    {
      var eType = event.type;
```

```
var eSrc = event.srcElement;

if (eSrc.tagName == "IMG")
{
    eSrc = eSrc.parentNode;
}

if (eType == "mouseover")
{
    eSrc.className = "toolbar-button-hover";
}
else if (eType == "mouseout")
{
    eSrc.className = "toolbar-button";
}
else if (eType == "click")
{
    eSrc.className = "toolbar-button";
    window.location.href = eSrc.getAttribute("href");
}
}

function createToolbar(sName, aButtons) {
    var toolbar = document.createElement("div");
    toolbar.id = sName;
    toolbar.className = "toolbar";

    for (var i = 0; i < aButtons.length; i++)
    {
        var thisButton = aButtons[i];

        var button = document.createElement("span");
        var icon = document.createElement("img");

        button.setAttribute("href", thisButton[1]);
        button.className = "toolbar-button";

        button.onclick = button_mouseHandler;
        button.onmouseover = button_mouseHandler;
        button.onmouseout = button_mouseHandler;

        icon.src = thisButton[0];
        icon.className = "toolbar-icon";

        button.appendChild(icon);
        toolbar.appendChild(button);
    }

    document.body.appendChild(toolbar);
}

var myToolbar = new Array();

myToolbar[0] = new Array();
myToolbar[0][0] = "img/green.gif";
```

```
myToolbar[0][1] = "javascript: alert('You Clicked the Green Button!')";

myToolbar[1] = new Array();
myToolbar[1][0] = "img/blue.gif";
myToolbar[1][1] = "javascript: alert('You Clicked the Blue Button!')";

myToolbar[2] = new Array();
myToolbar[2][0] = "img/red.gif";
myToolbar[2][1] = "http://www.wrox.com";
</script>
</head>
<body onload="createToolbar('myToolbar', myToolbar);">

</body>
</html>
```

Save this file as `toolbar_ie.htm`. When you load it into Internet Explorer, you should see something like what is shown in Figure 13-17.



Figure 13-17

Move your mouse pointer over the buttons, and you'll see them become highlighted. When you move your mouse over another button, the previous button un-highlights itself. Click any of the buttons. The green and blue buttons will display an alert box, and the red button will take you to `www.wrox.com`.

DHTML Example: The Toolbar in Firefox and Opera

Writing the toolbar script for Firefox is surprisingly easy, as most of the code can be reused. As you've already learned, IE and Firefox share many similarities, but they greatly differ in their event models.

The toolbar script follows this same pattern. You'll have to make a change in the CSS and change the event handlers to work with the DOM event model, but other than that your code will remain unchanged.

Try It Out The Toolbar in Firefox and Opera

Open your text editor of choice and type the following:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Firefox, Opera, and Safari Toolbar</title>
  <style type="text/css">
    .toolbar
    {
      background-color: #E4E2D5;
      padding: 2px;
    }

    .toolbar span
    {
      display: -moz-inline-stack;
      display: inline-block;
      height: 24px;
      width: 24px;
    }

    .toolbar-button
    {
      padding: 3px;
    }

    .toolbar-button-hover
    {
      border: 1px solid #316AC5;
      background-color: #C1D2EE;
      padding: 2px;
      cursor: pointer;
    }

    .toolbar-icon
    {
      height: 24px;
      width: 24px;
    }
  </style>
  <script type="text/javascript">
    function button_mouseHandler(e)
    {
      var eType = e.type;
      var eSrc = e.target;

      if (eSrc.tagName == "IMG")
      {
        eSrc = eSrc.parentNode;
```

```
    }

    if (eType == "mouseover")
    {
        eSrc.className = "toolbar-button-hover";
    }
    else if (eType == "mouseout")
    {
        eSrc.className = "toolbar-button";
    }
    else if (eType == "click")
    {
        eSrc.className = "toolbar-button";
        window.location.href = eSrc.getAttribute("href");
    }
}

function createToolbar(sName, aButtons)
{
    var toolbar = document.createElement("div");
    toolbar.id = sName;
    toolbar.className = "toolbar";

    for (var i = 0; i < aButtons.length; i++)
    {
        var thisButton = aButtons[i];

        var button = document.createElement("span");
        var icon = document.createElement("img");

        button.setAttribute("href", thisButton[1]);
        button.className = "toolbar-button";

        button.onclick = button_mouseHandler;
        button.onmouseover = button_mouseHandler;
        button.onmouseout = button_mouseHandler;

        icon.src = thisButton[0];
        icon.className = "toolbar-icon";

        button.appendChild(icon);
        toolbar.appendChild(button);
    }

    document.body.appendChild(toolbar);
}

var myToolbar = new Array();

myToolbar[0] = new Array();
myToolbar[0][0] = "img/green.gif";
myToolbar[0][1] = "javascript: alert('You Clicked the Green Button!')";

myToolbar[1] = new Array();
```

```

myToolbar[1][0] = "img/blue.gif";
myToolbar[1][1] = "javascript: alert('You Clicked the Blue Button!')";

myToolbar[2] = new Array();
myToolbar[2][0] = "img/red.gif";
myToolbar[2][1] = "http://www.wrox.com";
</script>
</head>
<body onload="createToolbar('myToolbar', myToolbar)">

</body>
</html>

```

Save this file as `toolbar_ff.htm`. Open it in Firefox, Opera, or Safari, and you should see something like what is shown in Figure 13-18.

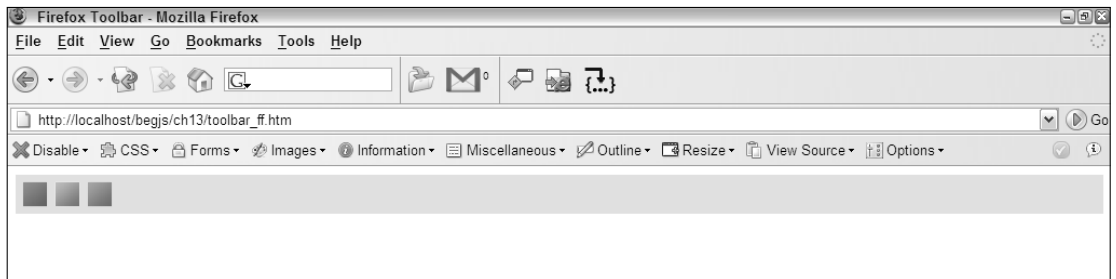


Figure 13-18

How It Works

This code remains largely the same as in the IE example. The changes, few as they may be, begin with the CSS. In the IE example you styled the buttons to be displayed inline-block, forcing the buttons to sit side by side while adding block-style options. In this second example you had to do something different.

```

.toolbar span
{
    display: -moz-inline-stack;
    display: inline-block;
    height: 24px;
    width: 24px;
}

```

Firefox does not support the `inline-block` value, but it does have a vendor-specific equivalent called `-moz-inline-stack`. Directly under it is another `display` style declaration, which is for Opera. Firefox will ignore this second `display` style declaration, as it does not recognize `inline-block` as a valid value for the `display` property.

Now let's jump to the mouse event handler. The primary changes made to the function are the values of the `eType` and `eSrc` variables, as well as an added parameter.

```
function button_mouseHandler(e)
{
    var eType = e.type;
    var eSrc = e.target;

    if (eSrc.tagName == "IMG")
    {
        eSrc = eSrc.parentNode;
    }

    if (eType == "mouseover")
    {
        eSrc.className = "toolbar-button-hover";
    }
    else if (eType == "mouseout")
    {
        eSrc.className = "toolbar-button";
    }
    else if (eType == "click")
    {
        eSrc.className = "toolbar-button";
        window.location.href = eSrc.getAttribute("href");
    }
}
```

Since you're now dealing with the DOM event model, you need to add a parameter to the function for the event object. Then you use the `target` property to retrieve the element where the event fired. The remainder of the function remains untouched: You make sure that `eSrc` is a button and change the element's `className` property according to the event.

Creating Cross-Browser DHTML

By now you've written one DHTML script and adapted it to work in both IE and Firefox. In this section, you'll combine the two versions into one cross-browser version. You probably already have an idea of what code you'll change, as you've already changed it once. However, here you'll employ a few tricks as well to ensure that the script works in both browsers.

Try It Out The Cross-Browser Toolbar

Open your text editor and type the following:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Cross-Browser Toolbar</title>
    <style type="text/css">
        .toolbar
        {
            background-color: #E4E2D5;
```

```
padding: 2px;
}

.toolbar span
{
    display: -moz-inline-stack;
    display: inline-block;
    height: 24px;
    width: 24px;
}

.toolbar-button
{
    padding: 3px;
}

.toolbar-button-hover
{
    border: 1px solid #316AC5;
    background-color: #C1D2EE;
    padding: 2px;
    cursor: pointer;
}

.toolbar-icon
{
    height: 24px;
    width: 24px;
}
</style>
<script type="text/javascript">
function button_mouseHandler(e)
{
    var eType;
    var eSrc;

    if (window.event)
    {
        eType = event.type;
        eSrc = event.srcElement;
    }
    else
    {
        eType = e.type;
        eSrc = e.target;
    }

    if (eSrc.tagName == "IMG")
    {
        eSrc = eSrc.parentNode;
    }

    if (eType == "mouseover")
    {
        eSrc.className = "toolbar-button-hover";
```

```
    }
    else if (eType == "mouseout")
    {
        eSrc.className = "toolbar-button";
    }
    else if (eType == "click")
    {
        eSrc.className = "toolbar-button";
        window.location.href = eSrc.getAttribute("href");
    }
}

function createToolbar(sName, aButtons)
{
    var toolbar = document.createElement("div");
    toolbar.id = sName;
    toolbar.className = "toolbar";

    for (var i = 0; i < aButtons.length; i++)
    {
        var thisButton = aButtons[i];

        var button = document.createElement("span");
        var icon = document.createElement("img");

        button.setAttribute("href", thisButton[1]);
        button.className = "toolbar-button";

        button.onclick = button_mouseHandler;
        button.onmouseover = button_mouseHandler;
        button.onmouseout = button_mouseHandler;

        icon.src = thisButton[0];
        icon.className = "toolbar-icon";

        button.appendChild(icon);
        toolbar.appendChild(button);
    }

    document.body.appendChild(toolbar);
}

var myToolbar = new Array();

myToolbar[0] = new Array();
myToolbar[0][0] = "img/green.gif";
myToolbar[0][1] = "javascript: alert('You Clicked the Green Button!')";

myToolbar[1] = new Array();
myToolbar[1][0] = "img/blue.gif";
myToolbar[1][1] = "javascript: alert('You Clicked the Blue Button!')";

myToolbar[2] = new Array();
myToolbar[2][0] = "img/red.gif";
myToolbar[2][1] = "http://www.wrox.com";
```

```
</script>
</head>
<body onload="createToolbar('myToolbar', myToolbar)">

</body>
</html>
```

Save this file as `toolbar_xb.htm`. Open it in IE, Firefox, and/or Opera. You should see the same thing you saw in the previous two examples.

How It Works

As in the Firefox and Opera example, we'll look only at the changes made to this version; those changes are confined to the `button_mouseHandler()` function. Because this is a cross-browser example, the code must cater to both the IE and DOM event models.

```
function button_mouseHandler(e)
{
    var eType;
    var eSrc;

    if (window.event)
    {
        eType = event.type;
        eSrc = event.srcElement;
    }
    else
    {
        eType = e.type;
        eSrc = e.target;
    }

    if (eSrc.tagName == "IMG")
    {
        eSrc = eSrc.parentNode;
    }

    if (eType == "mouseover")
    {
        eSrc.className = "toolbar-button-hover";
    }
    else if (eType == "mouseout")
    {
        eSrc.className = "toolbar-button";
    }
    else if (eType == "click")
    {
        eSrc.className = "toolbar-button";
        window.location.href = eSrc.getAttribute("href");
    }
}
```

This new code uses object detection to assign `eType` and `eSrc` their proper values. When this process is complete, the function behaves as it did in the previous examples.

Chapter 13: Dynamic HTML in Modern Browsers

This example hasn't been very large, or overly complex. However, the concepts and problems reflect a difficulty DHTML authors face all the time: working with two different event models and using CSS workarounds to ensure that the DHTML displays correctly in both types of browser. If you're aware of this difficulty, dealing with it really isn't too hard.

Summary

This chapter has featured quite a few diversions and digressions, but these were necessary to demonstrate the position and importance of the Document Object Model in JavaScript.

This chapter covered the following points:

- ❑ You started by outlining four of the main standards — HTML, ECMAScript, XML, and XHTML — and examined the relationships among them. You saw that a common aim emerging from these standards was to provide guidelines for coding HTML web pages. Those guidelines in turn benefited the Document Object Model, making it possible to access and manipulate any item on the web page using script if web pages were coded according to these guidelines.
- ❑ You examined the Document Object Model and saw that it offered a browser- and language-independent means of accessing the items on a web page, and that it resolved some of the problems that dogged older browsers. You saw how the DOM represents the HTML document as a tree structure and how it is possible for you to navigate through the tree to different elements and use the properties and methods it exposes in order to access the different parts of the web page.
- ❑ Although sticking to the standards provides the best method for manipulating the contents of the web page, none of the main browsers yet implements it in its entirety. You looked at the most up-to-date examples and saw how they provided a strong basis for the creation of dynamic, interoperable web pages because of their support of the DOM.

Exercise Questions

Suggested solutions to these questions can be found in Appendix A.

Question 1

Here's some HTML code that creates a web page. Re-create this page, using JavaScript to generate the HTML using only DOM objects, properties, and methods. Test your code in IE, Firefox, Opera, and Safari (if you have it) to make sure it works in them.

Hint: Comment each line as you write it to keep track of where you are in the tree structure, and create a new variable for every element on the page (for example, not just one for each of the TD cells, but nine variables).

```
<html>
<head>
</head>
<body>
  <table>
    <thead>
      <tr>
        <td>Car</td>
        <td>Top Speed</td>
        <td>Price</td>
      </tr>
    </thead>
    <tbody>
      <tr>
        <td>Chevrolet</td>
        <td>120mph</td>
        <td>$10,000</td>
      </tr>
      <tr>
        <td>Pontiac</td>
        <td>140mph</td>
        <td>$20,000</td>
      </tr>
    </tbody>
  </table>
</body>
</html>
```

Question 2

Augment your DOM web page so that the table has a border and only the headings of the table (that is, not the column headings) are center-aligned. Again, test your code in IE, Firefox, Opera, and Safari (if you have it).

Hint: Add any extra code to the end of the script code you have already written.

14

JavaScript and XML

In the previous chapter you took a brief look at Extensible Markup Language (XML). Like HTML, it consists of elements. You saw that its purpose was to describe data rather than to actually display information in any particular format, which is the purpose of HTML. There is nothing special about XML. It is just plain text with the addition of some XML tags enclosed in angle brackets. You can use any software that can handle plain text to create and edit XML.

In this chapter you'll be covering the fundamentals of XML. It's a huge topic and deserves a whole book to do it justice, so this'll be a taster to get you started. Before you get down to coding, let's look at what XML can be used for.

What Can XML Do for Me?

Developers like XML for a variety of reasons. It makes many web development tasks much simpler than they would be with HTML; it also makes possible many tasks that HTML simply cannot do. It is a powerful language with the ability to mold to your specific needs.

XML is a data-centric language. It not only contains data, but it describes those data by using semantic element names. The document's structure also plays a part in the description. Unlike HTML, XML is not a formatting language; in fact, a properly structured XML document is devoid of any formatting elements. This concept is often referred to as the "separation of content and style," and is part of XML's success, as it makes the language simple and easy to use.

For example, you can use XML as a data store like a database. In fact, XML is well suited for large and complex documents because the data are structured; you design the structure and implement it using your own elements to describe the data enclosed in the element. The ability to define the structure and elements used in an XML document is what makes XML a *self-describing* language. That is, the elements describe the data they contain, and the structure describes how data are related to each other.

Another method in which XML has become useful is in retrieving data from remote servers. Probably the most widely known applications of this method are the RSS and Atom formats for

web syndication. These XML documents, and others like them, contain information readily available to anyone. Web sites or programs can connect to the remote server, download a copy of the XML document, and use the information however needed.

A third, and extremely helpful, application of XML, is the ability to transfer data between incompatible systems. An XML document is a plain text document; therefore, all operating systems can read and write to XML files. The only major requirement is an application that understands the XML language and the document structure. For example, Microsoft recently released details on Microsoft Office Open XML, the file format used in Microsoft Office 2007. The files themselves are actually Zip files. However, any program written to read the XML files contained in the Zip file can display the data with no problem; it doesn't matter whether they were written under Windows XP, Mac OS X, any flavor of Linux, or any other operating system.

The Basics of XML

The advantage of XML over other document formats is that it specifies a protocol by which a document can describe itself. For example, a document might describe itself as a particular chapter in a book, containing the chapter's title, the sections, and the text, which is broken into different paragraphs. In order to write XML documents, you need to have a firm grasp on the fundamentals of the language.

Understanding XML Syntax

XML is a simple language with simple syntax and rules. By following the guidelines in this section, you'll create XML documents with little trouble.

XML Is a Well-Formed Language

Well-formed documents are those that comply with the rules of XML syntax. The requirements include, but are not limited to, the following:

- ❑ All elements must have an opening and closing tag, or a closing slash in the empty element.
- ❑ Elements must be nested correctly within the root element.

It is important to make well-formed XML documents because every XML parser is built according to the XML specification. It is an unforgiving language, meaning that any XML parser that follows the standard specification will never display documents that are not well formed.

Close the Tags!

The XML specification requires that all elements must have an opening tag and a closing tag. This is one way in which XML is unlike HTML, in which several elements have no closing tag (like `` and `
`). So when you open a tag, you need to close it when it contains the data that you want it to.

```
<?xml version="1.0" encoding="iso-8859-1"?>

<myElement>Some data go here</myElement>
```

This example shows a simple XML document. The first line contains the XML declaration. This tells the application that is going to use this XML document which version of the specification the document uses. Right now there is only version 1.0 (version 1.1 is coming soon). Note that the line starts with `<?` and ends with `?>`. These are the delimiters that indicate there is an XML declaration instruction between them. In this case, the instruction tells which version of XML the document uses. The declaration also states the character encoding used in the document. In this case the document conforms to the 1.0 specification of XML and uses the ISO-8859-1 (Latin-1/West European) character set.

XML declarations have no closing tag.

After the declaration, you see a couple of tags enclosing some text. The first tag is the opening tag for the `myElement` element, and the second is the closing tag for the same element. XML is a case-sensitive language; therefore, make sure the closing tag exactly matches the opening tag.

A well-formed element does not need to have any data between the opening and closing tags. Take the following line as an example:

```
<myElement></myElement>
```

This, too, is a well-formed XML element, even though it contains no data. This is referred to as an *empty element*. In elements such as these, use a shorthand version of closing the tag.

```
<myElement />
```

This line of code, as far as an XML parser is concerned, is identical to the previous line of code. So when you have empty elements, you can use the shorthand way of closing them.

There is no rule in XML that states an element must contain data.

Correctly Nest Elements

For years, browser makers have built their Web browsers to render and display pages that are not well formed. Not that it's a problem, as the Web has grown by leaps and bounds because of the everyman/-woman. However, if you loaded these HTML documents into an XML parser, it would throw error after error. The largest culprit would most likely be incorrectly nested elements.

XML requires properly nested elements; they cannot overlap as they can in HTML. For example, the following XML document is not well formed:

```
<?xml version="1.0" encoding="iso-8859-1"?>

<myDogs>
  <name>Morgan</name>
  <name>Molly
</myDogs></name>
```

This XML is almost well formed; however, the second `<name/>` element's closing tag comes after the `<myDogs/>` closing tag. This is an example of overlapping elements and will make any XML document invalid. Close an open element before you close its parent element.

If you follow these two rules, you'll have an easy time when writing XML documents. And speaking of which, let's delve more into XML's syntax.

Document Structure

XML was designed to provide a means of storing data in a structured way. All XML documents must have at least one element. The first element in the document is called the *root element* or the *document element*. No matter which name you use, both terms mean the same thing. All XML documents must have a root element, and they cannot have more than one.

Think of an operating system's directory structure. Everything begins with the root directory (C:). This main directory can have many subdirectories, and those subdirectories can have many more subdirectories, and so on.

An XML document is very similar. You start with the root element and build the document from there. For example, look at the following XML document:

```
<?xml version="1.0" encoding="iso-8859-1"?>

<myDogs>
  <name>Morgan</name>
  <name>Molly</name>
</myDogs>
```

The first element, `<myDogs />`, is the root element of the document. From here, two elements called `<name />` are added. You could even go farther and add more data (and elements) until you're satisfied with the document. There is no limit to the amount of elements you can use in a document; just remember that there can be only one root element, and that the document builds off of that element.

XML Elements

XML uses elements to describe the data enclosed in the document. So when you create your own XML documents, make sure that the elements properly describe the data enclosed in them. Let's expand upon the dogs document.

```
<?xml version="1.0" encoding="iso-8859-1"?>

<myDogs>
  <dog breed="Labrador Retriever">
    <name>Morgan</name>
  </dog>
  <dog breed="Labrador Retriever">
    <name>Molly</name>
  </dog>
</myDogs>
```

This document has some new information, and the document structure has changed as a result. Despite the changes, the document is still well formed, as each opening tag has a corresponding closing tag, and the elements are nested correctly.

The very first line of an XML document contains the XML declaration. In this case the document conforms to the 1.0 specification of XML and uses the ISO-8859-1 (Latin-1/West European) character set.

The next line describes the start tag of the root element of the document. In this example, it's saying "This document contains information on my dogs."

```
<myDogs>
```

The next line opens the `<dog/>` element, a child of the root element. This element contains information regarding one of the dogs.

```
<dog breed="Labrador Retriever">
  <name>Morgan</name>
</dog>
```

Obviously, this information has changed from the simpler XML document in the previous section. Adding the `<dog/>` element enables you to add as much information as you want for each dog.

Also added to the `<dog/>` element is the `breed` attribute, which contains the particular breed of the dog. Elements can have attributes, which are values passed to the application (in this case the Internet browser such as IE or Netscape) but are not part of the element content. Attributes are contained in the opening tag of the element and their values must be enclosed with quotes. Some people use attributes in place of creating a child node.

Lastly, the `<dog/>` element contains one child element called `name`, which holds the name of the dog.

The next `<dog/>` element contains the same type of data:

```
<dog breed="Labrador Retriever">
  <name>Molly</name>
</dog>
```

And finally, the last line defines the end of the root element.

```
</myDogs>
```

The elements (and attributes) in this document properly describe the data they enclose. You can look at this document and have no problem deciphering what the data are, and how they relate to the elements.

Character Data

Character data may be any legal (Unicode) character with the exception of `<`. The `<` character is reserved for the start of an opening or closing tag.

XML also provides a few useful entity references to clarify whether you are specifying character data or markup. Specifically, XML provides the following entity references:

Actual Character	Entity Reference
>	>
<	<
&	&
'	'

Chapter 14: JavaScript and XML

Obviously, the `<` entity reference is useful for character data. The other entity references can be used within markup where there could be confusion, such as the following:

```
<statement value = "She said, "Don't go there!" " />
```

This line should be written as follows:

```
<statement value = "She said, &quot;Don&apos;t go there!&quot;" />
```

CDATA

A pretty good rule of thumb is to consider anything outside of an element's tags to be character data and anything inside the tags to be markup. But, unfortunately, in one case this is not true. In the special case of CDATA blocks, all element tags and entity references are ignored by an XML processor, which treats them just like any old character data.

CDATA blocks have been provided as a convenience measure when you want to include large blocks of special characters as character data but do not want to have to use entity references. What if you wanted to write about an XML document in XML? Consider the following example:

```
<example>
  &lt;document&gt;
    &lt;name&gt;mrs smith&lt;/name&gt;
    &lt;email&gt;mrssmith@herdomain.com&lt;/email&gt;
  &lt;/document&gt;
</example>
```

As this code demonstrates, such a task requires the use of entity references for all opening and closing tags, which looks messy and makes the code tricky to read.

To avoid this inconvenience, use a CDATA block to specify that all character data should be considered character data whether or not they “look like” an opening/closing tag or entity reference, like this:

```
<example>
  <![CDATA[
    <document>
      <name>mrs smith</name>
      <email>mrssmith@herdomain.com</email>
    </document>
  ]]>
</example>
```

Comments

Not only will you sometimes want to include elements that you want the XML processor to ignore (that is, display as character data) in XML documents, but sometimes you will want to put into the document character data that you want the XML processor to ignore (that is, not display at all). This is where comments come in.

Note that comments will be displayed when you use the default style sheet. This type of text is called comment text.

XML comments are similar to HTML comments. In HTML, comments are specified with the `<!--` and `-->` syntax. In XML, comments are created in just the same way! So the following would be a valid XML comment:

```
<!--List Audioslave -->
<name>Chris Cornell</name>
<name>Tom Morello</name>
<name>Timmy C</name>
<name>Brad Wilk</name>
<!-- End the names -->
```

Keep a few rules in mind when using comments in XML documents. First, never have a hyphen or a double hyphen (`-` or `--`) within the text of the comment; these might be confusing to the XML processor. Second, never place a comment within an opening or closing tag. Thus, the following code would be poorly formed XML:

```
<name <!--The name --> >Tom Morello</name>
```

Likewise, never place a comment inside an entity declaration and never place a comment before the XML declaration.

XML declarations must always be the first line in any XML document.

Use comments to comment out groups of elements, if necessary. Thus, in the following case, all the names will be ignored except for Brad Wilk:

```
<!-- don't show these names
<name>Chris Cornell </name>
<name>Tom Morello </name>
<name>Timmy C</name>
-->
<name>Brad Wilk</name>
```

When commenting groups of elements, make sure that the remaining XML is well formed.

Creating an XML Document

It's time to put all this theory into practice. XML is all about data, so data are needed for the example, and you'll use information about my dogs. Here are some data fields to include:

- ☐ Name
- ☐ Age
- ☐ Breed
- ☐ Full blood
- ☐ Coat color

Using these data fields, it's possible to create the document's structure.

```
<?xml version="1.0" encoding="iso-8859-1"?>

<myDogs>
  <dog breed=" ">
    <name></name>
    <age>
      <years></years>
      <months></months>
    </age>
    <fullBlood></fullBlood>
    <color></color>
  </dog>
</myDogs>
```

What's listed here is the basic document structure; the data will be added later. Let's discuss the elements listed here.

Remember, elements should be descriptive of the data they will contain. Also keep in mind that XML is case-sensitive. An element's closing tag should exactly match its corresponding opening tag.

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

As you already know, the first line is the XML declaration, telling the XML parser that you're using version 1.0 and the ISO-8859-1 (Latin-1/West European) character set. Next comes the root element, `<myDogs />`.

```
<myDogs>
</myDogs>
```

This element is descriptive of the document; therefore, anyone who sees this document can easily infer that the data contained in it relate to my dogs. And since this is the root element, all other tags are contained within it.

```
<myDogs>
  <dog breed=" ">
    </dog>
</myDogs>
```

The `<dog />` element is a direct child to `<myDogs />`, and it contains data specific to an individual dog. It has one attribute, called `breed`, whose value matches that of the specific breed of this dog. The next elements, children of `<dog />`, contain information that is even more specific:

```
<myDogs>
  <dog breed=" ">
    <name></name>
    <age>
      <years></years>
      <months></months>
    </age>
    <fullBlood></fullBlood>
    <color></color>
  </dog>
</myDogs>
```

The `<name/>` element contains the dog's name, and the `<age/>` element expresses the dog's age. Because ages typically are counted in months and years, `<age/>` contains two child elements, `<years/>` and `<months/>`. Next, the `<fullBlood/>` element contains one of two Boolean values: `yes` or `no`. Naturally, if the dog is a purebreed, the value contained in this element is `yes`. And last, the `<color/>` element will contain the color of the dog's coat.

What you have here is a well-formed XML document structure, but let's take it a step further and see how you can make it valid.

Document Type Definition

If you make a layout mistake in building your XML file, the browser will tell you where things don't match up; it will not inform you if the mistake is related to the actual data. We are all human and prone to mistakes (not often, but they do happen!).

This section introduces a new term: *valid*. XML documents can be well formed or they can be valid, or they can be both. Valid documents are well-formed documents that also conform to a Document Type Definition (DTD). A DTD provides the structure for an XML document. Look at an e-mail; it has structure. The e-mail has a To: field, a Subject: field, and a body. There are even optional CC: and BCC: fields. The e-mail program fills in the From: field, and the date and time sent, for us, but they are there. The structure is so familiar you don't really notice it, unless it gets messed up in the transmission! This structure makes e-mail easy to read. If a program attempts to process this e-mail (which is data, when you get down to it), it must know the e-mail's structure in order to parse it. Knowing the structure means being able to parse it correctly each and every time, time after time.

DTDs lay out the way an XML file is to be marked up. Anyone following this DTD can process an XML file from others who have also built their XML files to the same DTD. Following a DTD also enables you to trap errors if a file that is not well formed or that has the wrong data is passed to the application.

Enough about what a DTD can do. Let's build one and see.

Creating the First DTD File

The purposes of a DTD are as follows:

- ☐ Declare what the markup is
- ☐ Declare what the markup means

But how do you build one? There is an entire specification on how to write DTDs. This section only touches on some of the items that can be used in a DTD.

Open any text editor (Notepad in Windows works just fine) and start a new file, name it `mydogs.dtd`, and follow along. Here is what the DTD for the `myDogs` document will look like:

```
<!-- The myDogs DTD -->
<!ELEMENT myDogs (dog+)>

<!-- The <dog/> section -->
```

```
<!ELEMENT dog (name, age, fullBlood, color)>
  <!ATTLIST dog breed CDATA #REQUIRED>
  <!ELEMENT name (#PCDATA)>
  <!ELEMENT age (years, months)>
    <!-- The <age/> section-->
    <!ELEMENT months (#PCDATA)>
    <!ELEMENT years (#PCDATA)>
    <!-- END age section-->
  <!ELEMENT fullBlood (#PCDATA)>
  <!ELEMENT color (#PCDATA)>
  <!-- END <dog/> section -->
<!-- END of myDogs DTD-->
```

The first line, which follows, is a comment. Anything between the `<!--` and `-->` will be ignored.

```
<!-- The myDogs DTD -->
```

It's always good to comment your code. All element declarations follow the following format:

```
<!ELEMENT elementName (content)>
```

This declaration is used to declare the elements used in the document. To define the root element, use `myDogs` as the element name. Under `myDogs` are multiple `<dog/>` elements, thus the `+` symbol.

```
<!ELEMENT myDogs (dog+)>
```

Next is another comment to indicate that what follows is the definition for the `<dog/>` element.

```
<!-- The <dog/> section -->
```

If you refer back to the desired XML structure, you'll see that the `<dog/>` element has an attribute called `breed` and contains the following elements: `name`, `age`, `fullBlood`, and `color`.

```
<!ELEMENT dog (name, age, fullBlood, color)>
```

The commas between each child name and the next indicate that `<dog/>` has strict ordering. If any element is out of order, the XML document is not valid. Because `age` has sub-elements, they will be defined later. Next, define the `breed` attribute.

```
<!ATTLIST dog breed CDATA #REQUIRED>
```

Defining attributes are similar to elements, and follow the following pattern:

```
<!ATTLIST elementName attributeName attributeType defaultValue>
```

There are several attribute types, but the most common is `CDATA`. For the default value, `#REQUIRED` is used to tell the XML parser that the `breed` attribute must be used in every `<dog/>` element. There are also other possible default values to use, but `#REQUIRED` best fits the document's needs. (After all, you want to know what type of dog you have, don't you?)

Use # in #REQUIRED (and in the following code, in #PCDATA) to prevent these words from being interpreted as element names.

Here is the start of the actual element type definitions. These follow the same rules we just went over.

```
<!ELEMENT name (#PCDATA)>
<!ELEMENT age (months, years)>
  <!-- The <age/> section-->
  <!ELEMENT years (#PCDATA)>
  <!ELEMENT months (#PCDATA)>
  <!-- END age section-->
<!ELEMENT fullBlood (#PCDATA)>
<!ELEMENT color (#PCDATA)>
<!-- END <dog/> section -->
<!-- END of myDogs DTD-->
```

That's it, and this is the simplest way to write the DTD. When you write the XML document, you'll reference this DTD by using the <!DOCTYPE> declaration in the XML document.

Bring on the Data

You now have the means to make a well-formed XML document valid. Let's add some actual data to it. The following table shows the data to use:

Data Fields	Dog 1	Dog 2	Dog 3
Name	Morgan	Molly	Casey
Age	10 months	8 years	6 years
Breed	Labrador Retriever	Labrador Retriever	Pomeranian
Full Blood	Yes	Yes	Yes
Color	Chocolate	Yellow	Brown

Once again, open any plain-text editor, and create this XML document:

```
<?xml version="1.0" encoding="iso-8859-1"?>

<!DOCTYPE myDogs SYSTEM "myDogs.dtd">

<myDogs>
  <dog breed="Labrador Retriever">
    <name>Morgan</name>
    <age>
      <years>0</years>
      <months>10</months>
    </age>
    <fullBlood>yes</fullBlood>
    <color>chocolate</color>
  </dog>
```

```
<dog breed="Labrador Retriever">
  <name>Molly</name>
  <age>
    <years>8</years>
    <months>11</months>
  </age>
  <fullBlood>yes</fullBlood>
  <color>yellow</color>
</dog>
<dog breed="Pomeranian">
  <name>Casey</name>
  <age>
    <years>6</years>
    <months>2</months>
  </age>
  <fullBlood>yes</fullBlood>
  <color>brown</color>
</dog>
</myDogs>
```

Save this file as `mydogs.xml`. There's not much of anything new in this XML document. The document structure matches that of the design made earlier in the section — but, of course, data now populate the `<dog/>` elements. There is, however, the addition of the DTD file you created earlier.

```
<!DOCTYPE myDogs SYSTEM "myDogs.dtd">
```

`!DOCTYPE` tells the XML parser that you are specifying a document to use as part of this file. Immediately following is `myDogs`, the name of the root element. `SYSTEM` lets the parser know that the document uses external DTD, a DTD found outside of the XML file. Finally, `"myDogs.dtd"` is the location of the DTD. If the DTD were on a web server, the URI would be a path like `http://myserver/myxmlstuff/myfile.dtd`. If the DTD and the XML file reside in the same folder, only the name of the DTD file needs referencing.

If you did not want an external DTD but instead wished to include the DTD within the XML file, this is what a portion of it would look like:

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

```
<!DOCTYPE myDogs [
  <!-- The myDogs DTD-->
  <!ELEMENT myDogs (dog+)>
  <!-- The dog section -->
  <!ELEMENT dog (name, age, fullBlood, color)>
  <!ATTLIST dog breed CDATA #REQUIRED>
  .....(rows removed for demonstration).....
  <!-- END of myDogs DTD-->
]>
<myDogs>
```

Either way you go, an XML parser that can validate XML documents will consider this file to be valid.

Go ahead and load `mydogs.xml` into your browser. In IE 7, you should see a page like what is shown in Figure 14-1.

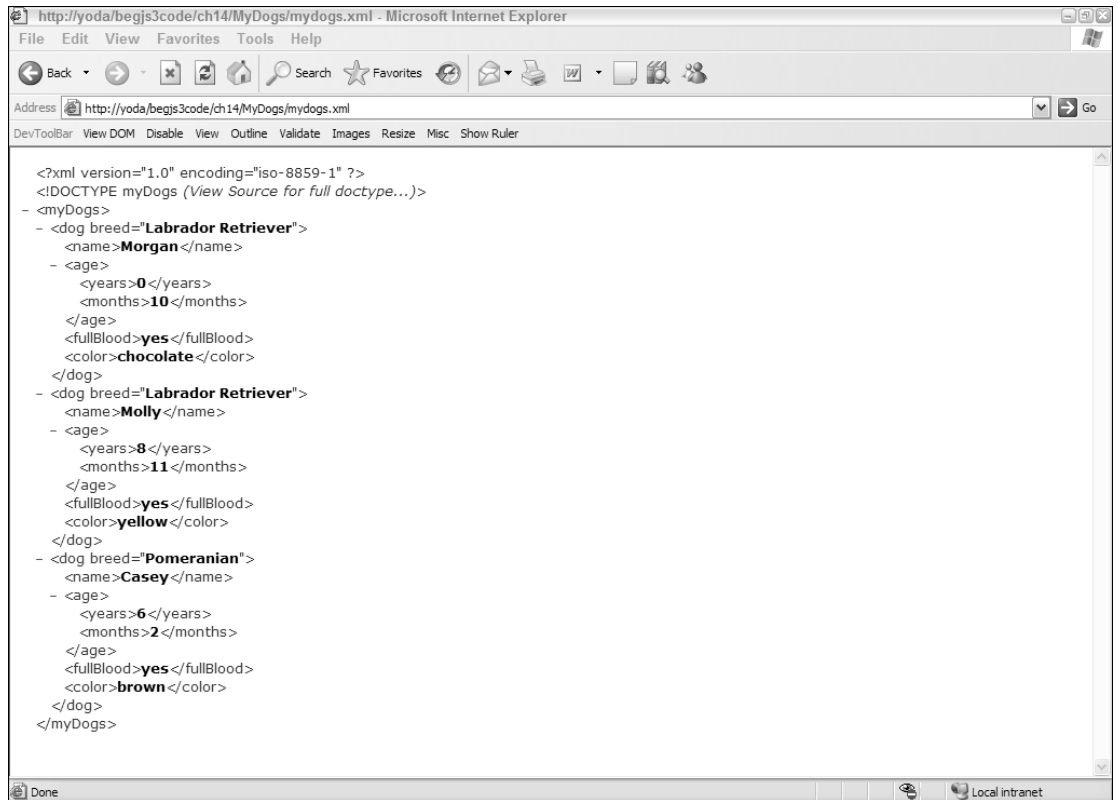


Figure 14-1

And in Firefox, the document should look like what is shown in Figure 14-2.

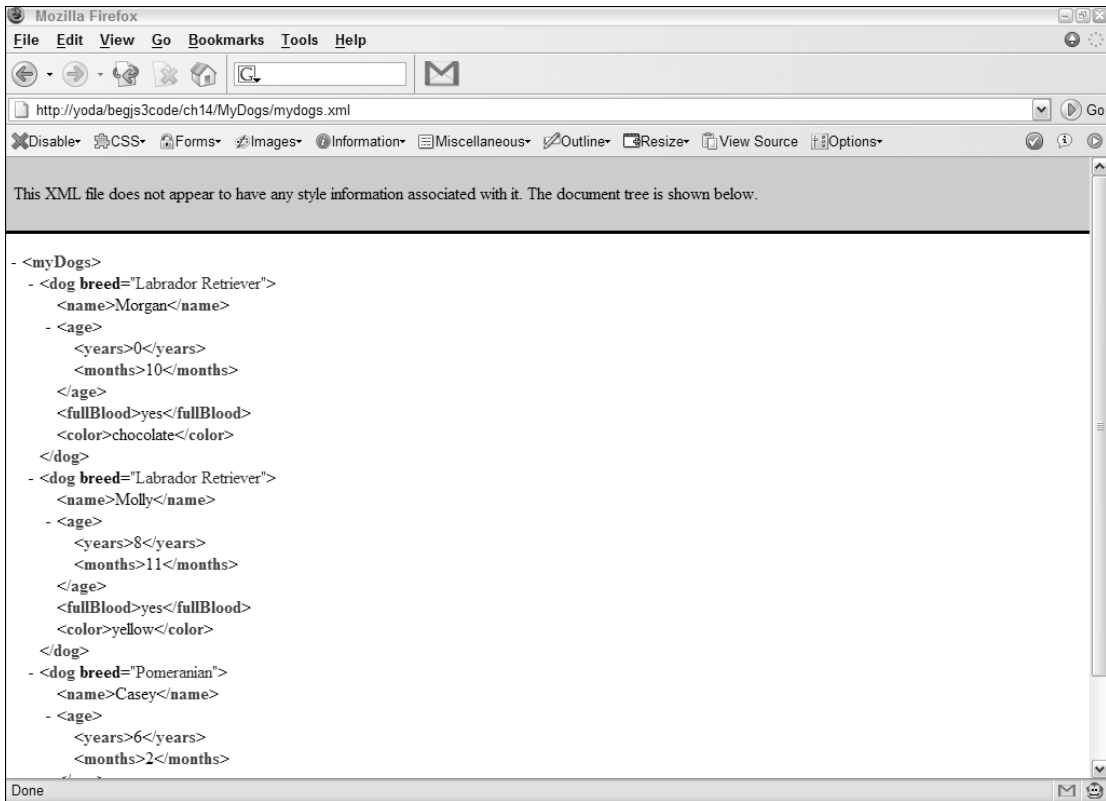


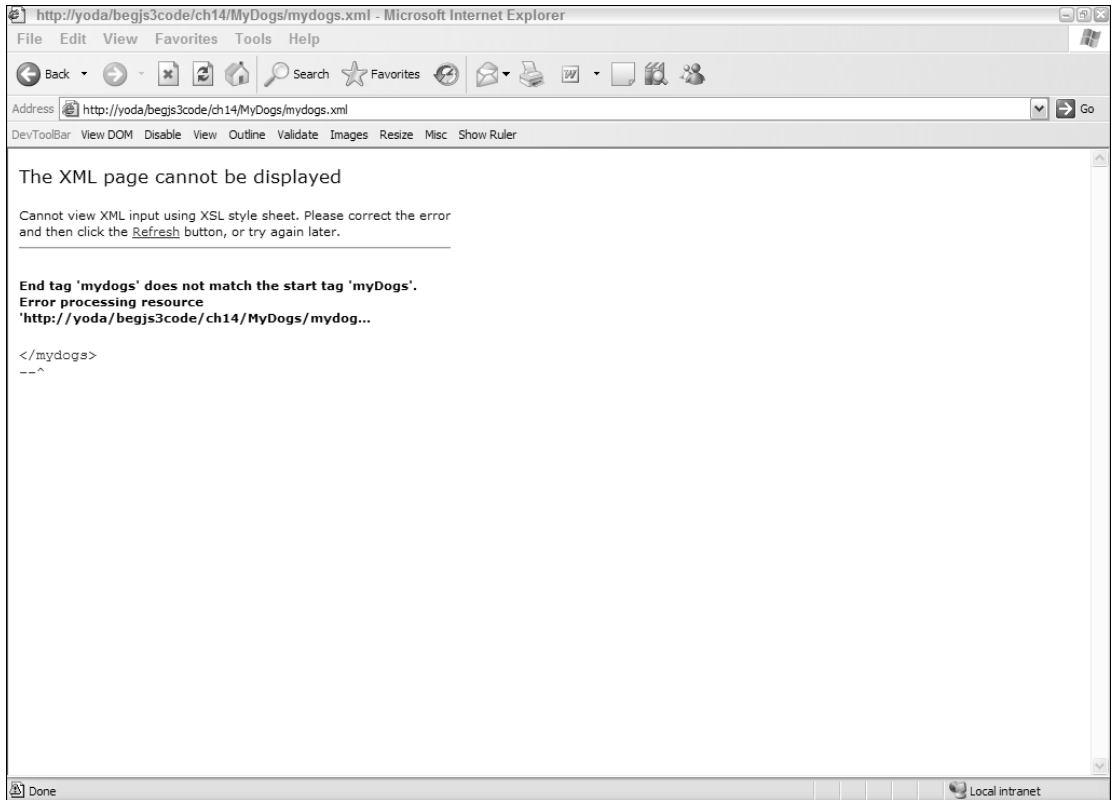
Figure 14-2

If a mistake is made in the XML documents, even a small one, the browser displays an error message like those in Figure 14-3 and Figure 14-4.

The deliberate mistake made here was the changing of the closing tag of `</myDogs>` to `</mydogs>`.

Because the browser can display XML, it checks the document to make sure it is well formed. Note that you may see a different look depending on which browser you use. Both IE and Firefox use XSLT, which formats the document to look as it does (you'll learn more about XSLT later in the chapter). Other browsers, however, do not.

Firefox cannot validate XML documents. IE doesn't validate documents by default, but you can download a tool to do so. It is called Internet Explorer Tools for Validating XML and Viewing XSLT Output and can be found at www.microsoft.com/downloads/details.aspx?FamilyID=d23c1d2c-1571-4d61-bda8-adf9f6849df9&DisplayLang=en.

**Figure 14-3**

Here is something else: Look at the XML file in the browser (see Figure 14-1), and you'll see several red dashes next to some opening tags. Clicking these dashes closes the corresponding part of the document hierarchy. You can expand and collapse elements/nodes that have children. Therefore, it's possible to collapse an entire `<dog/>` element, or the whole document.

The reason you can display this XML file in IE5+ and Firefox and have it be colored and collapsible is that those browsers have a default style sheet built in for those XML files that don't reference a style sheet of their own. The use of a style sheet with XML is termed Extensible Stylesheet Language (XSL). (More on this later.)

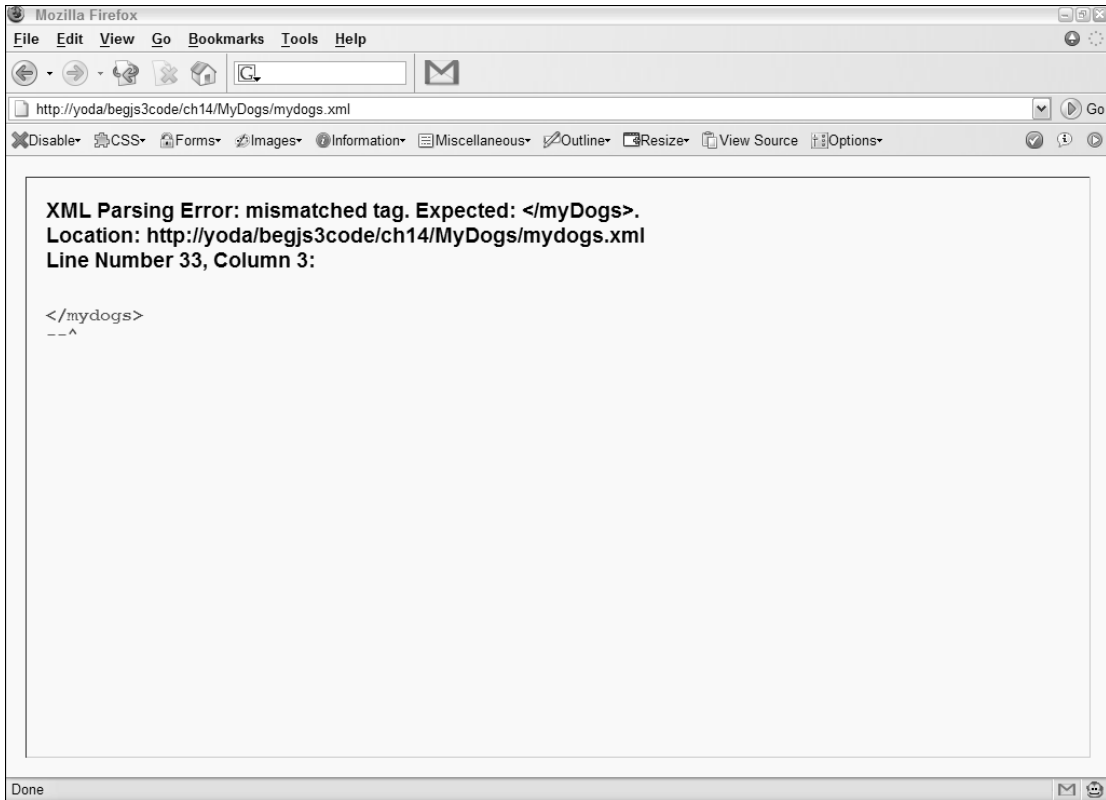


Figure 14-4

Altering the Look of XML

So far you have defined data and have seen those data displayed in a browser, but will your users like reading raw information? Nope, and this section shows you how to display and format it. This section doesn't teach Cascading Style Sheets (CSS); you saw the basics of CSS in Chapter 12. Instead, it covers only how to use them with XML.

Style Sheets and XML

Using CSS serves several purposes:

- ☐ It improves the clarity of the document when the document is displayed
- ☐ You can write once, use often
- ☐ One change can be reflected in many places
- ☐ You can present the same data in different ways for different purposes

With a style sheet, all the display rules are kept in one file that is linked to the source document. So you need to define what your display rules will be and associate them with your XML tags.

Create a new file called `mydogs.css` and type the following:

```
dog
{
  margin: 15px;
  display: block;
}

name
{
  display: block;
  font: bold 18pt Arial;
}

age
{
  display: block;
  color: red;
}

fullBlood
{
  display: none;
}

color
{
  display: block;
  font-style: italic;
}
```

Here some style rules are defined to match the elements that have data: `dog`, `name`, `age`, `fullBlood`, and `color`. Each one of these defined elements in the XML document contains data to be displayed. Just like you learned in Chapter 12, element names can be used in CSS to apply a style to that element.

What makes a style rule work are the properties, which are between the curly braces (`{ }`). Action is taken against the matched element. Starting with your `dog` element there are two properties: `margin` and `display`. The `margin` property is new. It's basically an edge that surrounds an element. If `margin` is `0px`, then the edge around the element is zero pixels. If it is `15px`, then there is a 15-pixel buffer around the element. In this case, the `margin` property emphasizes where a dog's data begins and ends.

The remainder of the elements, with the exception of `fullBlood`, are set to be placed on their own line with `display: block` (`fullBlood` is set to not be shown at all). The `name` element's text is bolded 18 point Arial. `age`'s text is colored red, and the `color` element is italicized.

There are now two files: `mydogs.xml` and `mydogs.css`, but the XML file needs to reference the CSS file. Open `mydogs.xml` and add the following line after the XML declaration. Save the file as `mydogs_css.xml`.

```
<?xml-stylesheet href="mydogs.css" type="text/css"?>
```

When you open your XML file in IE, you should see the screen shown in Figure 14-5.

Chapter 14: JavaScript and XML

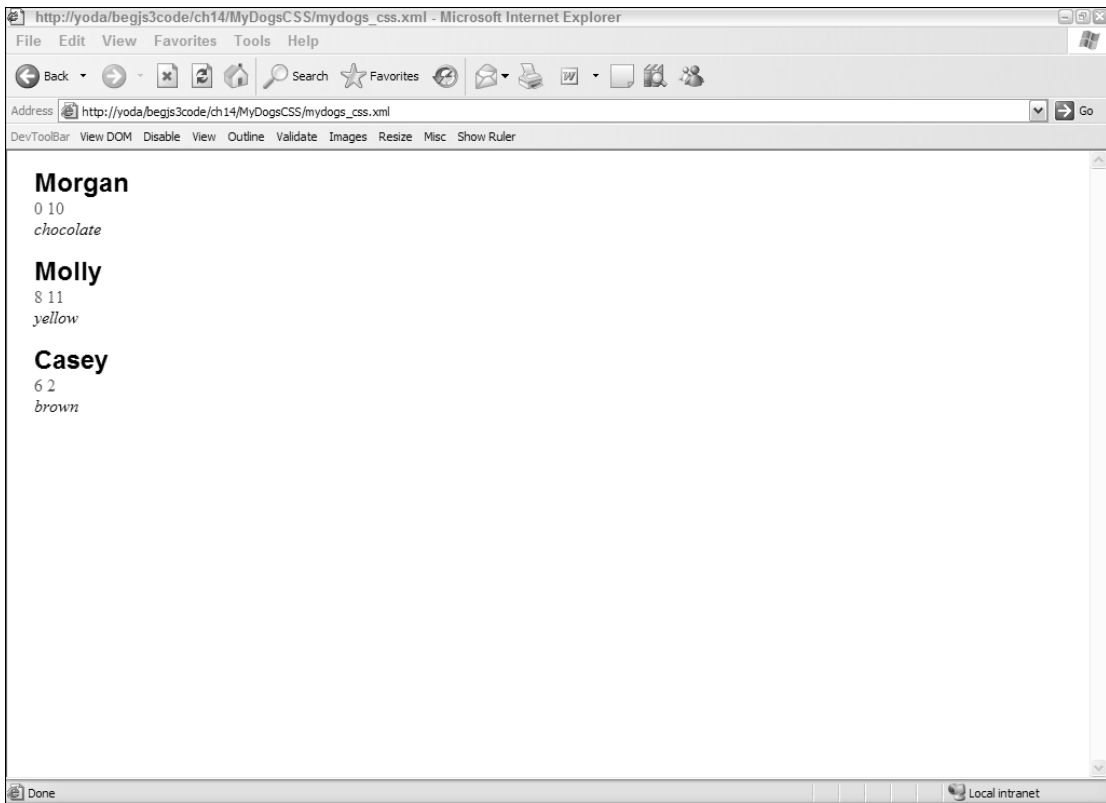


Figure 14-5

Figure 14-6 shows how the file looks in Firefox.

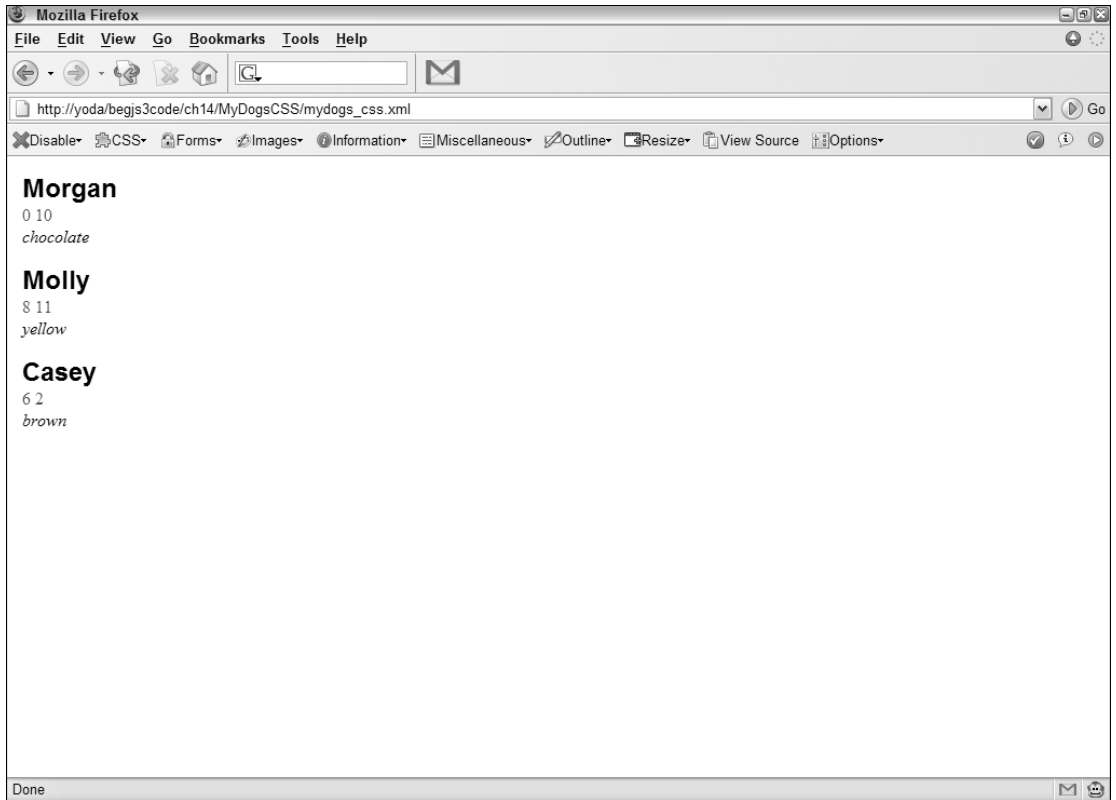
One thing you may notice is that the data are displayed, yet there's nothing really to describe what they're about. You, of course, know exactly what these data are, because you wrote it. CSS can make data look pretty, but what if you want more? It's a good thing you asked.

Extensible Stylesheet Language

The following sections show you how to use XSL to format XML documents using XSLT transformations.

What Is an XSLT Transformation?

XSLT is a template-based programming language that enables you to restructure your content to create the look you require. XSLT transformations can allow different kinds of software applications to exchange data. They can also be used to generate multiple views of the same source document to enable a web site's content to be displayed on myriad devices.

**Figure 14-6**

XSLT's elements and attributes provide a declarative programming language for processing XML data. You can use XSLT's vocabulary to grab content from other documents, create new elements and attributes, or, more commonly, some combination of the two.

An XSLT 1.0 transformation requires two starting documents (an XML source document and an XSLT style sheet) to produce a single result document (the upcoming XSLT 2.0 won't require this). The process begins when an XSLT style sheet is applied to an XML document to generate a result tree that is usually saved as an XML or HTML document (see Figure 14-7).

XSLT is a powerful language capable of turning XML into virtually any other text-based language or format.

Like CSS, XSLT has a construct called a rule, and it is responsible for selecting elements to which to apply the transformation.

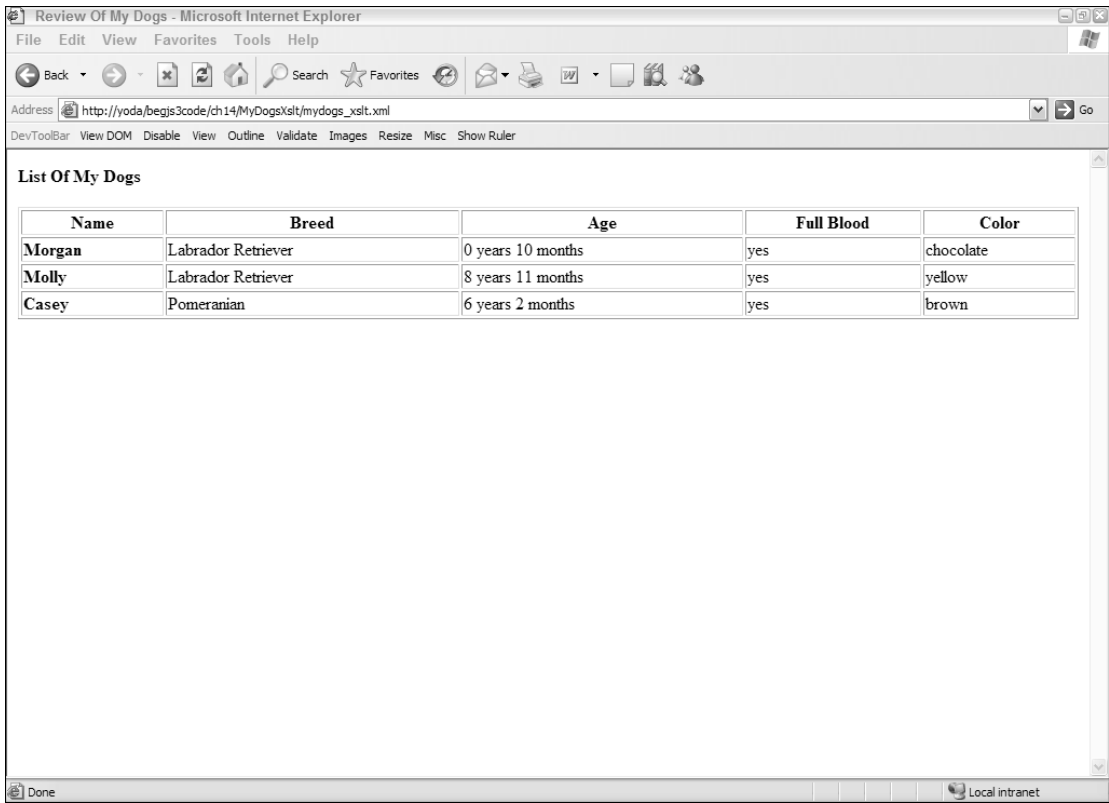


Figure 14-7

What Is a Template Rule?

An XSLT style sheet contains template rules that define the following:

- ❑ The criteria for selecting elements from the source tree
- ❑ Instructions for restructuring the selected content to create the result tree

Template rules are the key building blocks of XSLT transformations. A template rule has two parts: a pattern and a template. The pattern (usually an element name) is then matched against elements in the source tree to determine which nodes to select for processing. The template provides instructions for processing the selected content.

Each template rule constructs a result tree fragment of markup and content. An XSLT transformation’s result tree is created when all the result tree fragments are combined in the order specified to form a completed whole.

Now let’s go through an example style sheet and get into more detail about XSLT’s template rules.

Example Template

The XSLT style sheet document needs to include an XML version declaration (just like any other XML document).

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

Next, declare XSLT's namespace prefix. XSLT's namespace-prefixed elements enable an XSLT processor to distinguish them from other kinds of markup in the style sheet, such as the HTML elements contained within a template rule.

XSLT's namespace prefix is declared within the style sheet's root element, which is allowed to be either `xsl:stylesheet` or `xsl:transform`.

An XSLT version attribute should be included between the root element and the `xsl` namespace-prefix declaration.

The XSLT namespace is identified by its unique URI value (`http://www.w3.org/1999/XSL/Transform`), not by the `xsl` prefix.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

Now that the namespace prefix is declared, you can go about creating some template rules that will transform the source glossary document into an HTML table. Much like an old-fashioned mail merge, the HTML markup fragments in the following template provide the XSLT processor with a model for restructuring the source content.

The `xsl:template` element is used to encapsulate template rules. The value of its `match` attribute contains an XPath statement that is used as a pattern to select source elements for processing. A document's first template almost always has a `match` value of `/`, which means the document root.

```
<?xml version="1.0" encoding="utf-8"?>

<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <html>
      <head>
        <title>Review Of My Dogs</title>
      </head>
      <body>
        <h4>List Of My Dogs</h4>
        <table width="100%" border="1">
          <thead>
            <tr>
              <th>Name</th>
              <th>Breed</th>
              <th>Age</th>
              <th>Full Blood</th>
              <th>Color</th>
            </tr>
          </thead>
        </table>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

```
        </tr>
      </thead>
      <tbody>
        <xsl:apply-templates/>
      </tbody>
    </table>
  </body>
</html>
</xsl:template>
```

The `apply-templates` element signals an XSLT processor to look for other template rules and apply them at that point in the style sheet's structure. The results of each rule combine to form the result tree of the completed transformation.

The following template matches each `<dog/>` element in the XML document, and creates table rows until it runs out of `<dog/>` elements to process.

```
<xsl:template match="dog">
  <tr>
    <td>
      <strong>
        <xsl:value-of select="name" />
      </strong>
    </td>
    <td>
      <xsl:value-of select="@breed" />
    </td>
    <td>
      <xsl:apply-templates select="age" />
    </td>
    <td>
      <xsl:value-of select="fullBlood" />
    </td>
    <td>
      <xsl:value-of select="color" />
    </td>
  </tr>
</xsl:template>
```

Notice how the `value-of` element also uses a `select` attribute to choose the content from the source document you wish to use to create the result document. This code also uses the `<xsl:apply-templates />` directive; its `select` attribute contains `"age"`, so the XML parser knows to look for another template that selects the `<age/>` element.

```
<xsl:template match="age">
  <xsl:value-of select="years"/> years
  <xsl:value-of select="months"/> months
</xsl:template>
</xsl:stylesheet>
```


This template selects `<age/>` elements and retrieves the value of their children: `<years/>` and `<months/>`. To give these values meaning, this code adds the words `years` and `months` after the elements' values. This way anyone viewing this information will have no problem deciphering what these values represent.

The finished XSL file looks like this:

```
<?xml version="1.0" encoding="utf-8"?>

<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <html>
      <head>
        <title>Review Of My Dogs</title>
      </head>
      <body>
        <h4>List Of My Dogs</h4>
        <table width="100%" border="1">
          <thead>
            <tr>
              <th>Name</th>
              <th>Breed</th>
              <th>Age</th>
              <th>Full Blood</th>
              <th>Color</th>
            </tr>
          </thead>
          <tbody>
            <xsl:apply-templates/>
          </tbody>
        </table>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="dog">
    <tr>
      <td>
        <strong>
          <xsl:value-of select="name" />
        </strong>
      </td>
      <td>
        <xsl:value-of select="@breed" />
      </td>
      <td>
        <xsl:apply-templates select="age" />
      </td>
```

```
<td>
  <xsl:value-of select="fullBlood" />
</td>
<td>
  <xsl:value-of select="color" />
</td>
</tr>
</xsl:template>

<xsl:template match="age">
  <xsl:value-of select="years"/> years
  <xsl:value-of select="months"/> months
</xsl:template>

</xsl:stylesheet>
```

Be sure to save it as `mydogs.xml`.

Linking an XML Document to Its XSL Style Sheet

Much as in the CSS example earlier in the chapter, associating an XSL style sheet to an XML document requires the use of an XML processing instruction. In fact, the syntax used is identical except for the value of the `type` attribute, which will be either `text/css` or `text/xml`, depending on which flavor of style sheet is used.

Following are the two variations side by side, to illustrate the different values of the `type` attribute.

```
<?xml-stylesheet type="text/xml" href="mydogs.xml"?>
<?xml-stylesheet type="text/css" href="mydogs.css"?>
```

The correct placement of the style sheet declaration is within a document's prologue (after the XML declaration and before the root element).

Now alter the `mydogs_css.xml` file and tell it to refer to the `mydogs.xml` file. Open up `mydogs_css.xml` and alter the top part as shown here:

```
<?xml version="1.0" encoding="iso-8859-1"?>

<?xml-stylesheet type="text/xml" href="mydogs.xml"?>

<!DOCTYPE myDogs SYSTEM "mydogs.dtd">

<myDogs>
```

```
<dog>
...
...
```

Apart from deleting the reference to your `mydogs.css` file and adding a reference to the `mydogs.xsl` file, you leave the file the same. Save it as `mydogs_xslt.xml` and load it into IE, and you'll see the page shown in Figure 14-8.

Figure 14-9 shows what it looks like in Firefox.

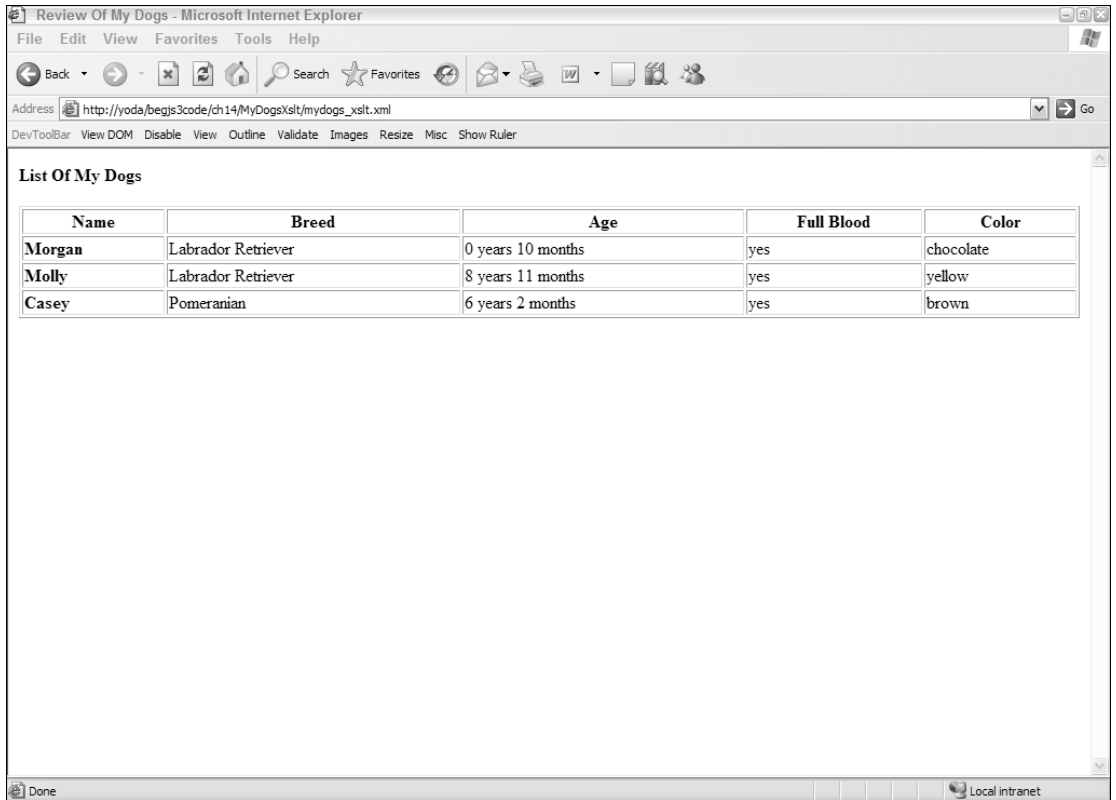


Figure 14-8

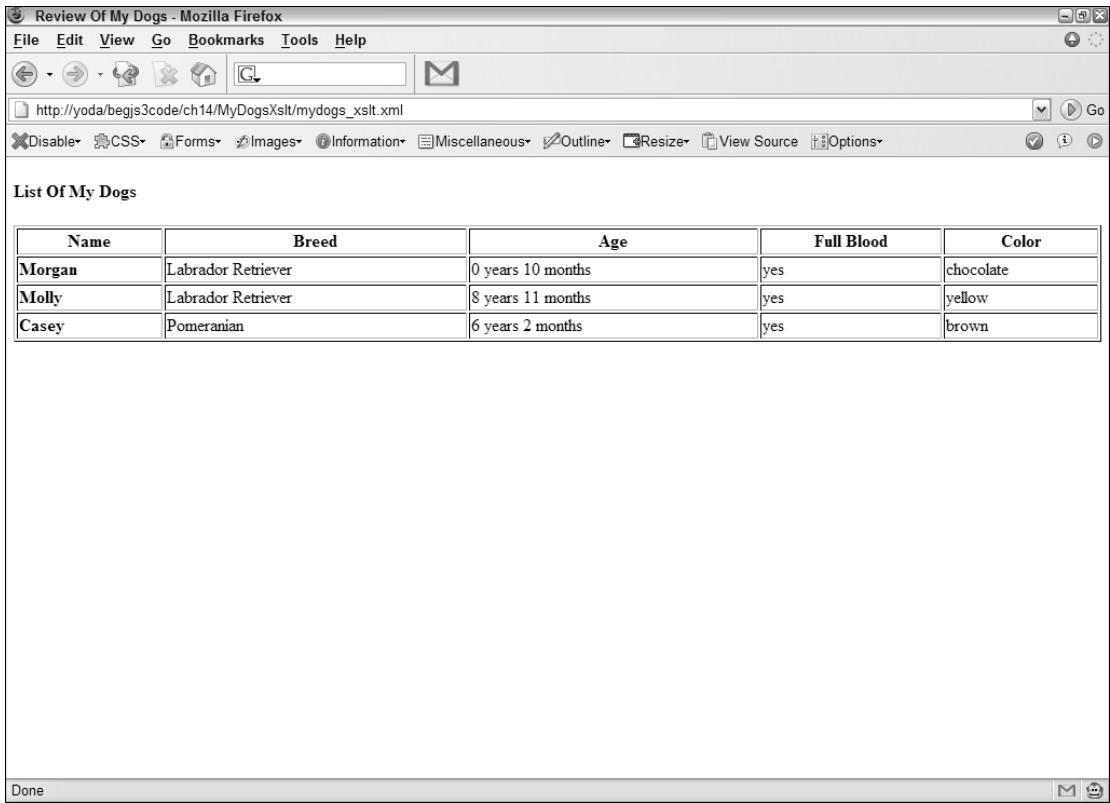


Figure 14-9

Manipulating XML with JavaScript

The good news is that you learned most of what you need to manipulate XML in a web browser from the previous chapter, when you learned about manipulating the DOM. The bad news is that although browser support for XML JavaScript is getting better and better with each new browser release, it's still far from perfect and a lot of cross-browser issues still exist. Because of this, this section will concentrate on script cross-compatible with IE5.5+ (on Windows; the Mac version of IE doesn't support XML), Firefox, and Opera, because these browsers have much improved XML JavaScript support.

The first task is to read the XML document. This is where the most cross-browser problems are located because IE and the other browsers have different ways of reading documents, Firefox and Opera being the more standards-compliant and IE being the easier to use and more comprehensive. The good news is that once the XML document is loaded, the differences between IE, Firefox, and Opera are smaller, although Microsoft has added a lot of useful (but nonstandard) extensions to its implementation.

Safari 2 does have some XML support, but its implementation for loading XML documents is much different from that of IE, Firefox, and Opera, and requires the use of the `XMLHttpRequest` object covered in Chapter 16.

Retrieving an XML File in IE

Internet Explorer relies upon the `ActiveXObject()` object and the MSXML library to fetch and open XML documents. A variety of ActiveX objects are available for scripting; to create an ActiveX object, simply call the `ActiveXObject()` constructor and pass a string containing the version of the ActiveX object you wish to create.

```
var xmlDoc = new ActiveXObject("Microsoft.XMLDOM");
xmlDoc.load("myfile.xml");
```

This code creates an XML DOM object that enables you to load and manipulate XML documents by using the version string `"Microsoft.XMLDOM"`. When the XML DOM object is created, load an XML document by using the `load()` method. This code loads a fictitious file called `myfile.xml`.

There are multiple versions of the Microsoft MSXML library, with each newer version offering more features and better performance than the one before. However, the user's computer must have these versions installed before you can use them, and the version selection code can become complex. Thankfully, Microsoft recommends checking for only two versions of MSXML. Their version strings are as follows:

- ❑ `Msxml2.DOMDocument.6.0`
- ❑ `Msxml2.DOMDocument.3.0`

You want to use the latest version possible when creating an XML DOM, and the following function does this:

```
function createDocument()
{
    //Temporary DOM object.
    var xmlDoc;

    //Create the DOM object for IE
    if (window.ActiveXObject)
    {
        var versions =
        [
            "Msxml2.DOMDocument.6.0",
            "Msxml2.DOMDocument.3.0"
        ];

        for (var i = 0; i < versions.length; i++)
        {
```

```
        try
        {
            xmlDoc = new ActiveXObject(versions[i]);
            return xmlDoc;
        }
        catch (error)
        {
            //do nothing here
        }
    }
    //no version was found; return null
    return null;
}
```

This code defines the `createDocument()` function. Its first line creates the `xmlDoc` variable. This is a temporary variable used in the creation of an XML DOM. The next line of code is an `if` statement, and it checks to see if the browser is IE by seeing if `window.ActiveXObject` exists. If the condition is `true`, then an array called `versions` is created, and the two MSXML versions are added as elements to the array.

```
var versions =
[
    "Msxml2.DOMDocument.6.0",
    "Msxml2.DOMDocument.3.0"
];
```

The order in which they're added is important; you want to always check for the latest version first, so the version strings are added with the newest at index 0.

Next is a `for` loop to loop through the elements of the `versions` array. Inside the loop is a `try...catch` statement.

```
for (var i = 0; i < versions.length; i++)
{
    try
    {
        xmlDoc = new ActiveXObject(versions[i]);
        return xmlDoc;
    }
    catch (error)
    {
        //do nothing here
    }
}
```

If the `ActiveXObject` object creation fails in the `try` block, then code execution drops to the `catch` block. Nothing happens at this point: The loop iterates to the next index in `versions` and attempts to create another `ActiveXObject` object with the other MSXML version strings. If every attempt fails, then the loop exits and returns `null`. Use the `createDocument()` function like this:

```
var xmlDoc = createDocument();
```

By using this function, you can create the latest MSXML XML DOM object easily.

Determining When the XML File Has Loaded

Before you actually attempt to manipulate the XML file, first make sure it has completely loaded into the client's browser cache. Otherwise, you're rolling the dice each time the page is viewed and running the risk of a JavaScript error being thrown whenever the execution of your script precedes the complete downloading of the XML file in question. Fortunately, there are ways to detect the current download state of an XML file.

The `async` property denotes whether the browser should wait for the specified XML file to fully load before proceeding with the download of the rest of the page. This property, whose name stands for *asynchronous*, is set by default to `true`, meaning the browser will not wait on the XML file before rendering everything else that follows. Setting this property to `false` instructs the browser to load the file first and then, and only then, to load the rest of the page.

```
var xmlDoc = createDocument();
xmlDoc.async = false; //Download XML file first, then load rest of page.
xmlDoc.load("myfile.xml");
```

The simplicity of the `async` property is not without its flaw. When you set this property to `false`, IE will stall the page until it makes contact and has fully received the specified XML file. When the browser is having trouble connecting and/or downloading the file, the page is left hanging like a monkey on a branch. This is where the `onreadystatechange` event handler and `readyState` property can help (as long as the `async` property is `true`).

The `readyState` property of IE exists for XML objects and many HTML objects, and returns the current loading status of the object. The following table shows the four possible return values.

Return Values for the <code>readyState</code> Property	Description
1	The object is initializing, but no data are being read (loading).
2	Data are being loaded into the object and parsed (loaded).
3	Parts of the object's data have been read and parsed, so the object model is available. However, the complete object data are not yet ready (interactive).
4	The object has been loaded, its content parsed (completed).

Chapter 14: JavaScript and XML

The value you're interested in here is the last one, 4, which indicates the object has fully loaded. To use the `readyState` property, assign a function to handle the `readystatechange` event, which fires every time the `readyState` changes.

```
//The function to handle the onreadystatechange event.
function xmlDoc_readyStateChange()
{
    //Check for the readyState. If it's 4, it's loaded!
    if (xmlDoc.readyState == 4)
    {
        alert("XML file loaded!");
    }
}

var xmlDoc = createDocument();
xmlDoc.onreadystatechange = xmlDoc_readyStateChange;

xmlDoc.load("myfile.xml");
```

This code first creates a function called `xmlDoc_readyStateChange()`. Use this function to handle the `readystatechange` event. Inside the function, check the `readyState` property to see if its value is equal to 4. If it is, then the XML file is completely loaded and the alert text "XML file loaded!" is displayed. Next, create the XML DOM object and assign the `xmlDoc_readyStateChange()` function to the `onreadystatechange` event handler. The last line of code initiates the loading of `myfile.xml`.

Retrieving an XML File in Firefox and Opera

Loading an XML document in Firefox and Opera is a little different from doing the same thing in IE, as these browsers use a more standards-centric approach. Creating an XML DOM doesn't require the use of an add-on as it does in IE; the DOM is a part of the browser and JavaScript implementation.

```
var xmlDoc = document.implementation.createDocument("", "", null);
xmlDoc.load("myfile.xml");
```

This code creates an empty DOM by using the `createDocument()` method of the `document.implementation` object. After the DOM object is created, use the `load()` method to load an XML document; it is supported by Firefox and Opera as well.

Determining When the File is Loaded

Much like IE, Firefox and Opera support the `async` property, which allows the file to be loaded asynchronously or synchronously. The behavior of loading synchronously is the same in these browsers as in IE. However, things change when you want to load a file asynchronously.

Not surprisingly, Firefox and Opera use a different implementation from IE when it comes to checking the load status of an XML file. In fact, these two browsers do not enable you to check the status with something like the `readyState` property. Instead, Firefox and Opera expose an `onload` event handler that executes when the file is loaded and the DOM object is ready to use.


```
//Handles the onload event
function xmlDoc_load()
{
    alert("XML is loaded!");
}

var xmlDoc = document.implementation.createDocument("", "", null);
xmlDoc.onload = xmlDoc_load;
xmlDoc.load("myfile.xml");
```

This code loads the fictitious file `myfile.xml` in asynchronous mode. When the load process completes, the load event fires and calls `xmlDoc_load()`, which then shows the text `XML is loaded!` to the user.

Cross-Browser XML File Retrieval

As you can see, the different ways of creating XML DOM objects require you to seek a cross-browser solution. You can easily do this with object detection to determine which browser is in use. In fact, you can easily edit the `createDocument()` function to include Firefox and Opera support. Look at the following code:

```
function createDocument()
{
    //Temporary DOM object.
    var xmlDoc;

    //Create the DOM object for IE
    if (window.ActiveXObject)
    {
        var versions =
        [
            "Msxml2.DOMDocument.6.0",
            "Msxml2.DOMDocument.3.0"
        ];

        for (var i = 0; i < versions.length; i++)
        {
            try
            {
                xmlDoc = new ActiveXObject(versions[i]);
                return xmlDoc;
            }
            catch (error)
            {
                //do nothing here
            }
        }
    }

    //Create the DOM for Firefox and Opera
    else if (document.implementation && document.implementation.createDocument)
    {
        xmlDoc = document.implementation.createDocument("", "", null);
        return xmlDoc;
    }
}
```

```
    //no version was found; return null
    return null;
}
```

The code highlighted in gray is the only new code added to the function, and it creates an XML DOM for Firefox and Opera with `document.implementation.createDocument()` and returns the DOM object to the caller. Using the function is exactly as you saw earlier: Now it works across browsers.

```
var xmlDoc = createDocument();
xmlDoc.async = false;
xmlDoc.load("myfile.xml");
```

Displaying a Daily Message

Now that you know how to load XML documents, let's jump right into building your first XML-enabled JavaScript application, a message-of-the-day display.

To begin, use the following simple XML file. You'll retrieve and display the daily message using DHTML. Save it as `motd.xml`.

```
<?xml version="1.0"?>

<messages>
  <daily>Today is Sunday.</daily>
  <daily>Today is Monday.</daily>
  <daily>Today is Tuesday.</daily>
  <daily>Today is Wednesday.</daily>
  <daily>Today is Thursday.</daily>
  <daily>Today is Friday.</daily>
  <daily>Today is Saturday.</daily>
</messages>
```

As you can see, this basic XML file is populated with a different message for each day of the week. First add the `createDocument()` function to the page.

```
<html>
<head>
  <title>Message of the Day</title>
  <script type="text/javascript">
    function createDocument()
    {
      //Temporary DOM object.
      var xmlDoc;

      //Create the DOM object for IE
      if (window.ActiveXObject)
      {
        var versions =
        [
          "Msxml2.DOMDocument.6.0",
          "Msxml2.DOMDocument.3.0"
```

```

    ];

    for (var i = 0; i < versions.length; i++)
    {
        try
        {
            xmlDoc = new ActiveXObject(versions[i]);
            return xmlDoc;
        }
        catch (error)
        {
            //do nothing here
        }
    }
}

//Create the DOM for Firefox and Opera
else if (document.implementation && document.implementation.createDocument)
{
    xmlDoc = document.implementation.createDocument("", "", null);
    return xmlDoc;
}

//no version was found; return null
return null;
}

//More code to come
</script>
</head>

```

Before you dig into the body of the page, there's one more function you need to add to the head of the page. This function is called `getDailyMessage()`, which retrieves and returns the message of the day.

```

<html>
<head>
    <title>Message of the Day</title>
    <script type="text/javascript">
        function createDocument()
        {
            //Temporary DOM object.
            var xmlDoc;

            //Create the DOM object for IE
            if (window.ActiveXObject)
            {
                var versions =
                [
                    "Msxml2.DOMDocument.6.0",
                    "Msxml2.DOMDocument.3.0"
                ];

                for (var i = 0; i < versions.length; i++)
                {
                    try
                    {

```

```
        xmlDoc = new ActiveXObject(versions[i]);
        return xmlDoc;
    }
    catch (error)
    {
        //do nothing here
    }
}
//Create the DOM for Firefox and Opera
else if (document.implementation && document.implementation.createDocument)
{
    xmlDoc = document.implementation.createDocument("", "", null);
    return xmlDoc;
}
//no version was found; return null
return null;
}
```

```
//Gets the message from the XML file
function getDailyMessage()
{
    //Get the node list of <daily/> elements.
    var messages = xmlDoc.getElementsByTagName("daily");
    //Create a date object.
    var dateobj = new Date();
    //And get today's day.
    var today = dateobj.getDay();

    //Return the message.
    return messages[today].firstChild.nodeValue;
}
</script>
```

```
</head>
```

First use the `getElementsByTagName()` method to retrieve the `<daily/>` elements. As you already know, this will return a node list of all the `<daily/>` elements. The next task is to find a numerical representation of the day of the week. Do this by first creating a `Date` object and using its `getDay()` method. This gives you a digit between 0 and 6 with 0 being Sunday, 1 being Monday, and so on; the digit is assigned to the `today` variable. Finally, use that variable as an index of the `messages` node list to select the correct `<daily/>` element and retrieve its text.

You may have noticed that `xmlDoc` in `getDailyMessage()` isn't declared anywhere in the head of the HTML document. It is used in `createDocument()`, but that variable is declared within the context of the function. You actually declare the global `xmlDoc` in the body of the HTML page.

```
<body>
<div id="messageContainer"></div>

<script type="text/javascript">
    //Create the DOM object
    var xmlDoc = createDocument();
```



```
}
//Create the DOM for Firefox and Opera
else if (document.implementation && document.implementation.createDocument)
{
    xmlDoc = document.implementation.createDocument("", "", null);
    return xmlDoc;
}
//no version was found; return null
return null;
}

var xmlDocDocument = createDocument();
xmlDocDocument.load("mydogs_js.xml");

function displayDogs()
{
    //Get the <dog/> elements.
    var dogNodes = xmlDocDocument.getElementsByTagName("dog");
    //Create a <table/> element.
    var table = document.createElement("table");
    table.setAttribute("cellPadding", 5); //Give the table some cell padding.
    table.setAttribute("width", "100%");
    table.setAttribute("border", "1");

    /*** Begin <thead/> Element. ***/
    var tableHeader = document.createElement("thead");
    //Create a <tr/> element.
    var tableRow = document.createElement("tr");

    //Loop through the child nodes of a <dog/> element.
    for (var i = 0; i < dogNodes[0].childNodes.length; i++)
    {
        var currentNode = dogNodes[0].childNodes[i];
        //Check to see if the child node is an element.
        if (currentNode.nodeType == 1)
        {
            //Create a <th/> element.
            var tableHeaderCell = document.createElement("th");
            //Create a text node with currentNode's nodeName.
            var textData = document.createTextNode(currentNode.nodeName);

            //Append the text node to the heading.
            tableHeaderCell.appendChild(textData);
            //Append heading to the row.
            tableRow.appendChild(tableHeaderCell);
        }
    }
    //Append the row with the column headers to the <thead/>
    tableHeader.appendChild(tableRow);
    //Append the <thead/> to the table.
    table.appendChild(tableHeader);
    /*** End <thead/> Element. ***/

    /*** Begin <tbody/> Element. ***/
    var tableBody = document.createElement("tbody");
```

```

//Loop through the <dog/> elements.
for (var i = 0; i < dogNodes.length; i++)
{
    //Create a new <tr/> element.
    var tableRow = document.createElement("tr");

    //Now loop through this <dog/>'s child nodes.
    for (var j = 0; j < dogNodes[i].childNodes.length; j++)
    {
        //Store the current node for easier access.
        var currentNode = dogNodes[i].childNodes[j];
        //Check the node to see if it's an element.
        if (currentNode.nodeType == 1)
        {
            //Create a data cell.
            var tableDataCell = document.createElement("td");
            //Create a text node with currentNode's nodeName.
            var textData = document.createTextNode
            (
                currentNode.firstChild.nodeValue
            );

            //Append the text node to the data cell.
            tableDataCell.appendChild(textData);

            //Append the data cell to the row.
            tableRow.appendChild(tableDataCell);
        }
    }
    //Append the row to the <tbody/>.
    tableBody.appendChild(tableRow);
}

//Append the tbody to the table.
table.appendChild(tableBody);

/** End <tbody/> Element. */

document.body.appendChild(table);
}
</script>
</head>
<body>
    <a href="javascript: displayDogs();">Display Dogs</a>
</body>
</html>

```

Open up `mydogs.xml`, and modify it to look like this. Save the modified version as `mydogs_js.xml`.

```

<myDogs>
  <dog>
    <name>Morgan</name>
    <breed>Labrador Retriever</breed>
    <age>0 years, 10 months</age>
  </dog>
</myDogs>

```

```
<fullBlood>yes</fullBlood>
<color>chocolate</color>
</dog>
<dog>
  <name>Molly</name>
  <breed>Labrador Retriever</breed>
  <age>8 years, 11 months</age>
  <fullBlood>yes</fullBlood>
  <color>yellow</color>
</dog>
<dog>
  <name>Casey</name>
  <breed>Pomeranian</breed>
  <age>6 years, 2 months</age>
  <fullBlood>yes</fullBlood>
  <color>brown</color>
</dog>
</myDogs>
```

When you open this page in your browser, you'll see a web page with only a link visible. When you click the link, you should see something like what is shown in Figure 14-10.

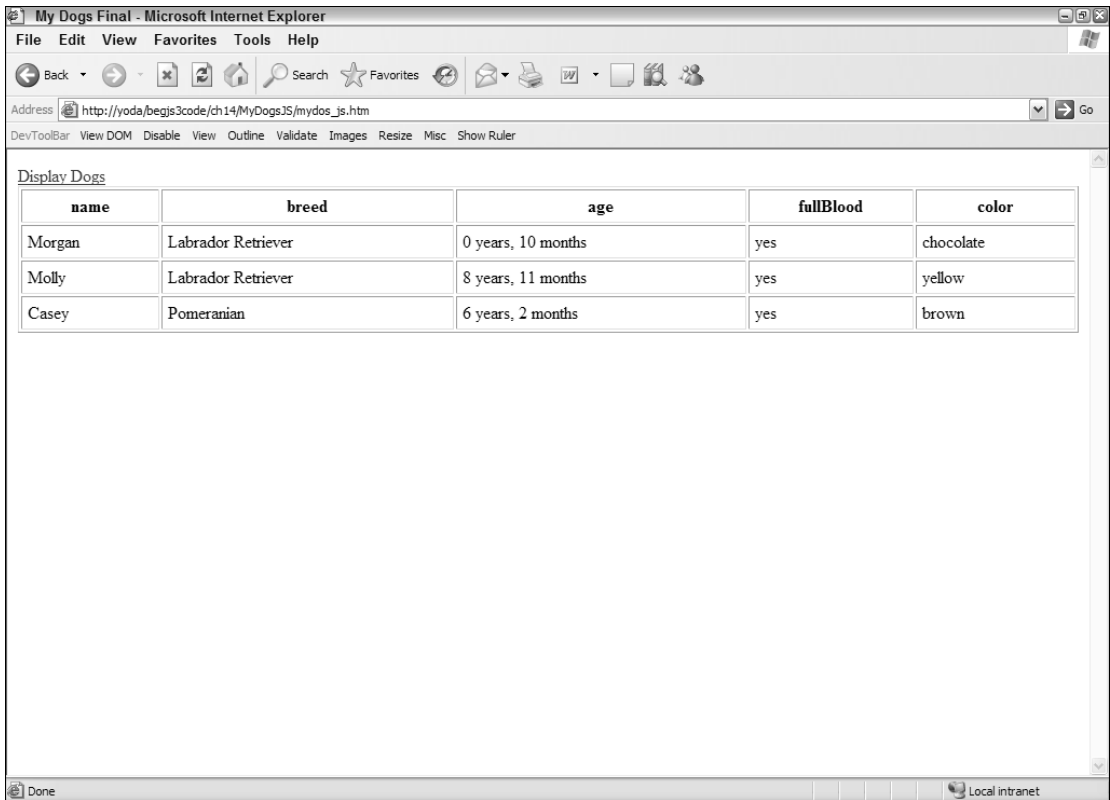


Figure 14-10

How It Works

The first thing you do is create a DOM object and load an XML document into it.

```
function createDocument()
{
    //Temporary DOM object.
    var xmlDoc;

    //Create the DOM object for IE
    if (window.ActiveXObject)
    {
        var versions =
        [
            "Msxml2.DOMDocument.6.0",
            "Msxml2.DOMDocument.3.0"
        ];

        for (var i = 0; i < versions.length; i++)
        {
            try
            {
                xmlDoc = new ActiveXObject(versions[i]);
                return xmlDoc;
            }
            catch (error)
            {
                //do nothing here
            }
        }
    }
    //Create the DOM for Firefox and Opera
    else if (document.implementation && document.implementation.createDocument)
    {
        xmlDoc = document.implementation.createDocument("", "", null);
        return xmlDoc;
    }
    //no version was found; return null
    return null;
}

var xmlDocument = createDocument();
xmlDocument.load("mydogs_final.xml");
```

The workhorse of this page is the next function, `displayDogs()`. Its job is to build a table and populate it with the information from the XML file.

```
function displayDogs()
{
    //Get the <dog/> elements.
    var dogNodes = xmlDocument.getElementsByTagName("dog");
    //Create a <table/> element.
    var table = document.createElement("table");
    table.setAttribute("cellPadding",5); //Give the table some cell padding.
```

Chapter 14: JavaScript and XML

```
table.setAttribute("width", "100%");
table.setAttribute("border", "1");
```

The first thing you do is use the `getElementsByTagName()` method to retrieve the `<dog/>` elements and assign the resulting array to `dogNodes`. Next, create a `<table/>` element by using the `document.createElement()` method, and set its `cellPadding`, `width`, and `border` attributes.

Next, create the table header and heading cells. For the column headers, use the tag names of the `<dog/>` element's children (name, breed, age, and so on).

```
/** Begin <thead/> Element. */
var tableHeader = document.createElement("thead");
//Create a <tr/> element.
var tableRow = document.createElement("tr");

//Loop through the child nodes of a <dog/> element.
for (var i = 0; i < dogNodes[0].childNodes.length; i++)
{
    var currentNode = dogNodes[0].childNodes[i];

    //More code here.
}
//Append the row with the column headers to the <thead/>
tableHeader.appendChild(tableRow);
//Append the <thead/> to the table.
table.appendChild(tableHeader);
```

The first few lines of this code create `<thead/>` and `<tr/>` elements. Then the code loops through the first `<dog/>` element's child nodes (more on this later). After the loop, append the `<tr/>` element to the table header and add the header to the table. Now let's look at the loop.

```
/** Begin <thead/> Element. */
var tableHeader = document.createElement("thead");
//Create a <tr/> element.
var tableRow = document.createElement("tr");

//Loop through the child nodes of a <dog/> element.
for (var i = 0; i < dogNodes[0].childNodes.length; i++)
{
    var currentNode = dogNodes[0].childNodes[i];
    //Check to see if the child node is an element.
    if (currentNode.nodeType == 1)
    {
        //Create a <th/> element.
        var tableHeaderCell = document.createElement("th");
        //Create a text node with currentNode's nodeName.
        var textData = document.createTextNode(currentNode.nodeName);

        //Append the text node to the heading.
        tableHeaderCell.appendChild(textData);
        //Append heading to the row.
        tableRow.appendChild(tableHeaderCell);
    }
}
```

```

}
//Append the row with the column headers to the <thead/>
tableHeader.appendChild(tableRow);
//Append the <thead/> to the table.
table.appendChild(tableHeader);

```

The goal is to use the element names as headers for the column. However, you're looping through every child node of a `<dog/>` element, so any instance of whitespace between elements is counted as a child node in Firefox and Opera. To solve this problem, check the current node's type with the `nodeType` property. If it's equal to 1, then the child node is an element. Next create a `<th/>` element, and a text node containing the current node's `nodeName`, which you append to the header cell. And finally, append the `<th/>` element to the row.

The second part of `displayDogs()` builds the body of the table and populates it with data. It is similar in look and function to the header-generation code.

```

/**/ Begin <tbody/> Element. ***/
var tableBody = document.createElement("tbody");

//Loop through the <dog/> elements.
for (var i = 0; i < dogNodes.length; i++)
{
    //Create a new <tr/> element.
    var tableRow = document.createElement("tr");

    //More code here

    //Append the row to the <tbody/>.
    tableBody.appendChild(tableRow);
}

//Append the tbody to the table.
table.appendChild(tableBody);

/**/ End <tbody/> Element. ***/

```

First create the `<tbody/>` element. Next, loop through the `dogNodes` array, cycling through the `<dog/>` elements. Inside this loop, create a `<tr/>` element and append it to the table's body. When the loop exits, append the `<tbody/>` element to the table. Now add data cells to the row:

```

/**/ Begin <tbody/> Element. ***/
var tableBody = document.createElement("tbody");

//Loop through the <dog/> elements.
for (var i = 0; i < dogNodes.length; i++)
{
    //Create a new <tr/> element.
    var tableRow = document.createElement("tr");

    //Now loop through this <dog/>'s child nodes.
    for (var j = 0; j < dogNodes[i].childNodes.length; j++)
    {
        //Store the current node for easier access.

```

```
        var currentNode = dogNodes[i].childNodes[j];
        //Check the node to see if it's an element.
        if (currentNode.nodeType == 1)
        {
            //Create a data cell.
            var tableDataCell = document.createElement("td");
            //Create a text node with currentNode's nodeName.
            var textData = document.createTextNode(
                currentNode.firstChild.nodeValue
            );

            //Append the text node to the data cell.
            tableDataCell.appendChild(textData);

            //Append the data cell to the row.
            tableRow.appendChild(tableDataCell);
        }
        //Append the row to the <tbody/>.
        tableBody.appendChild(tableRow);
    }

    //Append the tbody to the table.
    table.appendChild(tableBody);

    /** End <tbody/> Element. */
```

This inner loop cycles through the child elements of `<dog/>`. First assign the `currentNode` variable to reference the current node. This will enable you to access this node a little more easily (much less typing!). Next, check the node's type. Again, some browsers count whitespace as child nodes, so you need to make sure the current node is an element. When it's confirmed that the current node is an element, create a `<td/>` element and a text node containing the text of `currentNode`. Append the text node to the data cell, and append the data cell to the table row created in the outer `for` loop.

At this point the table is completed, so add it to the HTML page. You do this with the following:

```
        document.body.appendChild(table);
    }
```

Now all you have to do is invoke the `displayDogs()` function. To do this, you place a hyperlink in the page's body to call the function when clicked.

```
<a href="javascript: displayDogs();" >Display Dogs</a>
```

Summary

In this chapter you've taken a very brief look at XML, created your first XML document (a DTD), and then formatted the XML document using CSS and XSL. You looked at the following:

- ☐ What XML is used for
- ☐ How to create a well-formed and valid XML document
- ☐ The rules of XML syntax
- ☐ The XML data elements
- ☐ XSLT (Extensible Stylesheet Language Transformations)
- ☐ How XSLT can be used instead of CSS to display XML, and how it goes much further and enables HTML formatting to be created on the fly with XSLT
- ☐ Manipulating XML documents with JavaScript
- ☐ How to load an XML file and then manipulate its document with JavaScript

Exercise Questions

Suggested solutions to these questions can be found in Appendix A.

Question 1

Create an XML document that logically orders the following data for a school:

- ☐ Child's name
- ☐ Child's age
- ☐ Class the child is in

Use the following data:

Bibby Jones	13	1B
Beci Smith	12	1B
Jack Wilson	14	2C

Question 2

Using JavaScript, load the information from the XML file into a page and display it when the user clicks a link.

15

Using ActiveX and Plug-Ins with JavaScript

Today's browsers provide a lot of built-in functionality; however, there are many things they cannot do unaided, such as playing video or sound. Functionality of this sort has become quite common on the Internet, and it is thanks to plug-ins and their ability to extend browser functionality.

Plug-ins are applications that are downloaded and, as their name suggests, "plugged into" the browser. Many different plug-ins exist today; the more common ones include Adobe Flash Player, which plays Flash movies, and RealNetworks' Real Audio and video player, which plays real audio and media files.

Essentially, plug-ins are objects that encapsulate all the functionality they need to perform their tasks, such as playing audio files, in a way that hides the complexity from the programmer. They are usually written in languages such as C++ and Java.

Plug-ins usually, but not always, have some sort of user interface. For example, the Real Audio plug-in has a user interface that displays buttons to play, pause, and stop the playing of an audio file (see Figure 15-1).

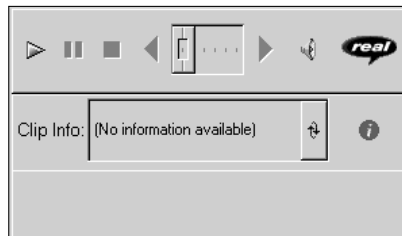


Figure 15-1

Some plug-ins make objects with various methods and properties available to you. You can access these through JavaScript, much as you access the methods and properties of the `window` object or the `Math` object. For example, the RealOne player plug-in makes available the `DoPlay()` method that you can use to play a sound clip.

Plug-ins have been around for quite some time; in fact, Netscape supported them back in version 3. You probably won't be shocked to find out that Microsoft does things differently from the other browser makers. IE does not support plug-ins, but IE 4.0+ running on Windows does support ActiveX controls, which provide the same functionality.

Fortunately, as you'll see, using ActiveX controls is similar to using Firefox plug-ins, and with a few tweaks can be done with almost the same code. The main difference is actually making sure that the plug-in or ActiveX control is available for use and ready to run in the user's browser in the first place. We'll cover this problem in more detail for Firefox and IE before going on to discuss using the plug-ins and ActiveX controls.

Checking for and Embedding Plug-ins in Firefox

It's nice to create a script to use a WizzoUltra3D plug-in for the web page experience of a lifetime, but unless the visitor to your web page also has the WizzoUltra3D plug-in installed on his computer, his experience of the web page is going to be one full of bugs and error messages. It is therefore important that you not only correctly add the HTML required to use the plug-in in your page, but also use JavaScript to check to see if the user's browser has the plug-in installed that your page makes use of. You look at both these topics in this section.

Even though this section focuses on Firefox, the same principles can be applied to Apple Safari and Opera.

Adding a Plug-in to the Page

To make use of a plug-in that is installed in the user's browser, you need to use HTML to tell the browser where and when in your page you want to use it. This process is called *embedding* the plug-in.

In Firefox, the key to embedding plug-ins is the non-standard `<embed/>` element. This inserts the visible interface, if any, of the plug-in at that point in the page. The `<embed/>` element supports a number of general attributes applicable to all plug-ins, such as `height`, `width`, `pluginspage`, `src`, and `type`. You'll look at the last two of these attributes, `src` and `type`, in more detail here. You will also look at the `pluginspage` attribute in the next section.

Most plug-ins display content that is stored on a web server. For example, a plug-in for sound, such as Real Audio, will play music from a file with the `.ra` extension, and the Flash plug-in will play Flash movies, that is, files with the `.swf` extension. The `<embed/>` element's `src` attribute enables you to specify the initial file for the plug-in to load and play. This will be a URL pointing to the file, usually hosted on the same web server as the HTML page. It's from this file that the browser determines what sort of plug-in is required. For example, if the `src` is `http://www.myserver.com/myflashmovie.swf`, then by checking the type of the file, the browser can see that a Flash player plug-in needs to be used.

However, not all plug-ins require data from an external source and therefore a value for the `src` attribute. In such situations, how can the browser tell what plug-in to load? Well, that's where the `<embed/>` element's `type` attribute comes in. The actual value for the `type` attribute will be specific to the plug-in. You can find out this information by typing `about:plugins` in the location bar. The plug-in information loads into the browser, as shown in Figure 15-2.

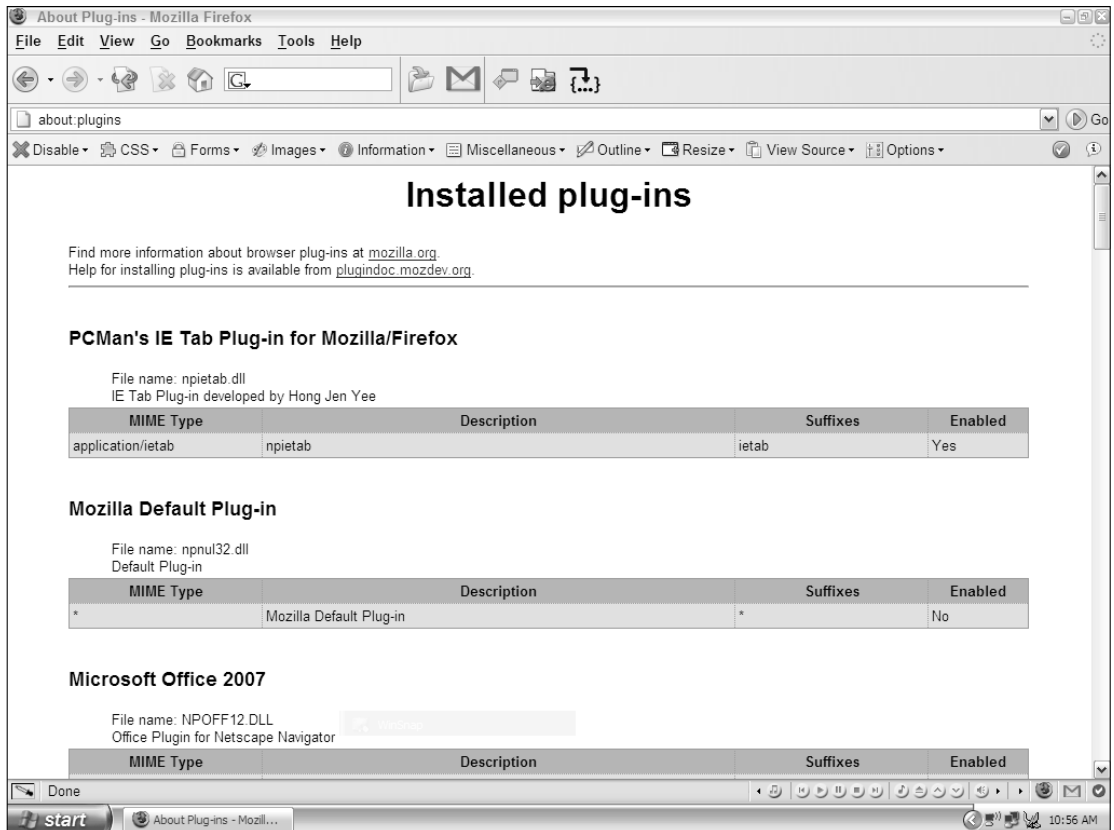


Figure 15-2

You'll see a list of all the plug-ins installed on your browser. The value required for the `type` attribute is listed as the Multipurpose Internet Mail Extensions (MIME) type, which specifies a type of content such as a web page, an image, or a Flash file. For example, the MIME type for Flash is `application/x-shockwave-flash`.

In addition to a number of attributes common to all plug-ins, you can also use the `<embed/>` element to specify properties specific to a particular plug-in. For example, the Flash plug-in supports the `quality` attribute, which determines the image quality of the Flash movie. To set this attribute in the `<embed/>` element, you just add it to the list of attributes set, as shown in the following example:

```
<embed id="FlashPlugIn1"
      src="topmenu.swf"
      border=0
```

```
height=100
width=500
quality=high
type="application/x-shockwave-flash"
</embed>
```

Although Firefox supports the `<embed/>` element, it also supports the use of the HTML standard `<object/>` element for embedding plug-ins into the page, in a similar way to IE, which you will see shortly.

Checking for and Installing Plug-ins in Firefox

After you decide what type of plug-in you want to embed into the page, what happens if the browser finds that this particular plug-in does not exist on the user's computer?

To solve this problem you can set the `pluginspage` attribute of `<embed/>` to point to a URL on the plug-in creator's page. If the plug-in is not on the user's computer, a link to the URL specified in the `pluginspage` attribute will be displayed within the web page. The user can click the link and load the plug-in so that your web page will function properly.

For example, with Flash the `pluginspage` attribute needed is this:

```
PLUGINSOURCE="http://www.adobe.com/shockwave/download/index.cgi?P1_Prod_Version=ShockwaveFlash">
```

However, if the user doesn't have the plug-in installed, you might prefer to send her to a version of your web site that doesn't rely on that plug-in. How do you know whether a plug-in is installed?

The `navigator` object, introduced in Chapter 5, has a property called `plugins`, which is an array of `Plugin` objects, one for each plug-in installed on that browser. You can access a `Plugin` object in the `plugins` array either by using an index value that indexes all the plug-ins installed on the user's browser, or by using the name of the plug-in application.

Internet Explorer has a `navigator.plugins` array, but it is always empty.

Each `Plugin` object has four properties: `description`, `filename`, `length`, and `name`. You can find these values by viewing the plug-ins information page that you saw earlier.

Let's use Flash as an example. Type **about:plugins** in the location bar and press enter. Figure 15-3 shows the Installed plug-ins page in Netscape, but this page remains largely the same in Firefox 2. You can see that Flash has Shockwave Flash as its `name` property. The `filename` and `description` properties have obvious meanings. The `length` property gives the number of MIME types supported by the plug-in.

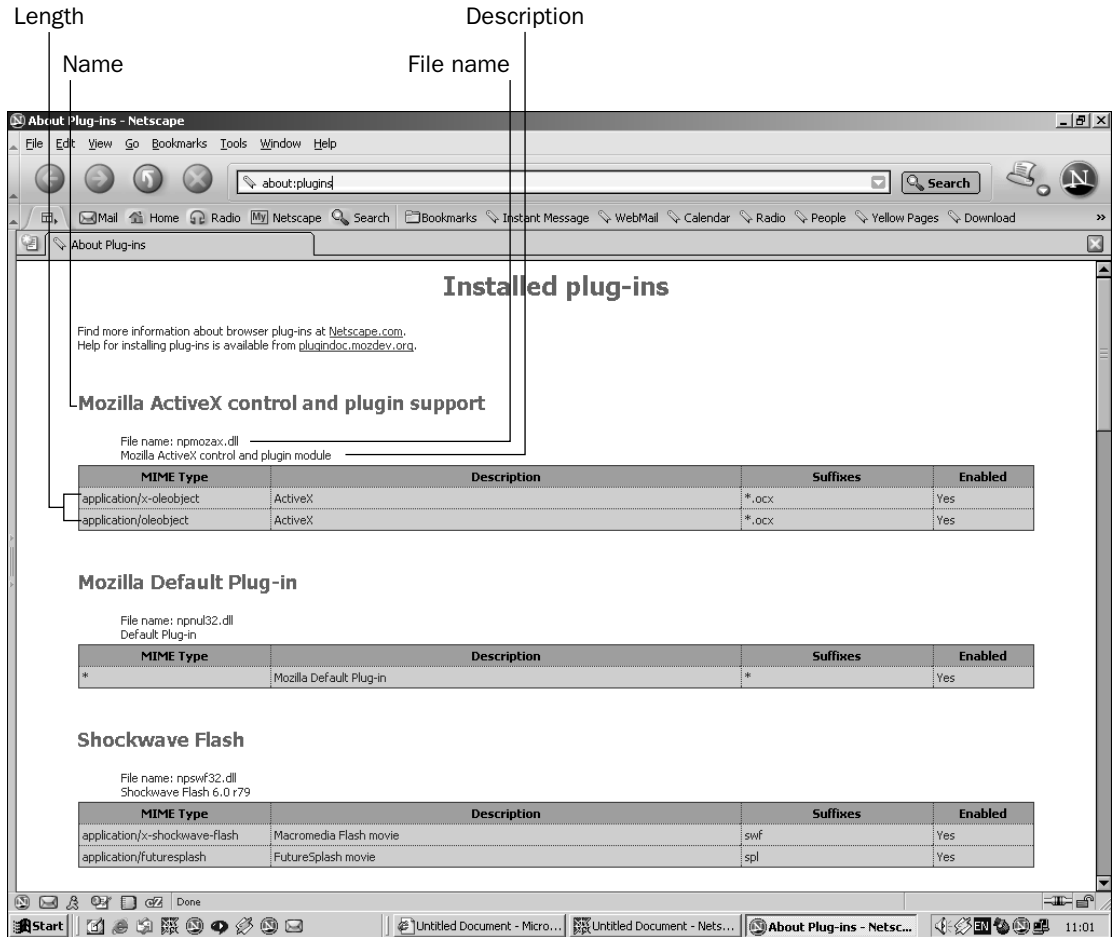


Figure 15-3

As mentioned earlier, the name property can be used to reference the `Plugin` object in the `plugins` array. So, the following code will set the variable `shockWavePlugin` to the `Plugin` object for Flash, if it's installed:

```
var shockWavePlugin = navigator.plugins["Shockwave Flash"];
```

If it's not, `navigator.plugins["Shockwave Flash"]` will return as undefined.

You can use the following to redirect users on browsers that do not have installed the plug-in you need:

```
if (navigator.plugins["Shockwave Flash"])
{
    window.location.replace("my_flash_enabled_page.htm");
}
else
```

Chapter 15: Using ActiveX and Plug-Ins with JavaScript

```
{
    window.location.replace("my_non_flash_page.htm");
}
```

If the Flash plug-in is not installed, `navigator.plugins["Shockwave Flash"]` will be undefined, which JavaScript considers to be false, thereby causing the `else` statement to execute. If Flash is installed, `navigator.plugins["Shockwave Flash"]` will return the Flash Plugin object, which JavaScript treats as true, and the main `if` statement will execute.

The problem with this method of detection is that the name given to a plug-in may vary from operating system to operating system. For example, the name of the Windows XP version of the plug-in may vary from the name of the Mac version, which in turn may vary from the name of the Linux version. Some plug-ins, such as RealPlayer, will not work reliably at all with this detection method, because the name is not simply RealPlayer but something that contains the word "RealPlayer."

An alternative method for determining whether a plug-in is installed is to loop through the `plugins[]` array and check each name for certain keywords. If you find them, you can assume that the control is installed. For example, to check for RealPlayer, you may use the following:

```
var plugInCounter;
for (plugInCounter = 0; plugInCounter < navigator.plugins.length;
    plugInCounter++)
{
    if (navigator.plugins[plugInCounter].name.indexOf("RealPlayer") >= 0)
    {
        alert("RealPlayer is installed");
        break;
    }
}
```

The `for` loop goes through the `navigator.plugins` array, starting from index 0 and continuing up to the last element. Each plug-in in the array has its `name` property checked to see if it contains the text `RealPlayer`. If it does, you know RealPlayer is installed, and you break out of the loop; if not, RealPlayer is clearly not installed.

An alternative to using `navigator` object's `plugins[]` array is using the `navigator` object's `mimeTypes[]` array, which contains an array of `MimeType` objects representing the MIME types supported by the browser. You can use this array to check whether the browser supports a specific type of media, such as Flash movies.

You have already come across MIME types before—the `type` attribute of the `<embed/>` element can be used to specify a MIME type so that the browser knows which plug-in to embed. Again, using the About Plug-ins option on the Help menu, you can find out what the MIME types are for a particular plug-in. In fact, one plug-in may well support more than one MIME type. When you check for a particular MIME type, you are checking that the browser supports a particular type of file format rather than necessarily a particular plug-in.

For example, you may use the `mimeTypes` array to check for the Flash plug-in as follows:

```
if (navigator.mimeTypes["application/x-shockwave-flash"] &&
    navigator.mimeTypes['application/x-shockwave-flash'].enabledPlugin)
{
```

```
        window.location.replace("my_flash_enabled_page.htm");
    }
    else
    {
        window.location.replace("my_non_flash_page.htm");
    }
}
```

The `if` statement is the important thing here. Its condition has two parts separated by the AND operator `&&`.

The first part checks that the specified MIME type is supported by trying to access a specific `mimeType` object in the `mimeTypes` array. If there is no such object, then `undefined` is returned, which, as far as an `if` statement goes, means the same thing as if `false` had been returned.

The second part checks to see not only that the MIME type is supported, but also that a plug-in to handle this MIME type is enabled. Although unusual, it is possible for a MIME type to be supported, or recognized, by the browser, but for no plug-in to be installed. For example, if the user has Microsoft Word installed, the MIME type `application/msword` would be valid, but that does not mean a plug-in exists to display it in the browser! The `enabledPlugin` property of the `mimeType` object actually returns a `Plugin` object, but again, if it does not exist, `null` will be returned, which will be considered as `false` by the `if` statement.

What happens if someone browses to your page with a browser that has no support at all for plug-ins?

Well, the `<embed/>` elements will be basically ignored, and if the browser does support script, errors will occur if you access the plug-in through your script. To get around this, use the `<noembed/>` element. Anything in between the opening `<noembed>` tag and the closing `</noembed>` tag is ignored by a browser that supports the `<embed/>` element, so you can put a message in there telling users that the page requires a browser that supports plug-ins.

```
<noembed>
  <h2> This page requires a browser that supports plug-ins </h2>
</noembed>
```

You can also use object checking to avoid errors with different browser versions.

```
if (document.embeds && document.embeds[0])
```

Checking for and Embedding ActiveX Controls on Internet Explorer

Although IE does support plug-ins to a certain extent, its support for ActiveX controls is more complete. The main difference between an ActiveX control and a plug-in is how they are embedded into a page and how they are installed. Once they are embedded and installed, their use, as far as scripting goes, will be very similar to that for plug-ins.

ActiveX controls are a little like mini-programs, usually created in languages like C++ or Visual Basic. Unlike normal programs, like Notepad or Microsoft Word, ActiveX controls cannot run on their own; they

Chapter 15: Using ActiveX and Plug-Ins with JavaScript

need to be sited in a container program. Not all programs can act as containers for ActiveX controls, only those specifically designed to do so, such as Microsoft Access and, of course, Internet Explorer. When the creator of the ActiveX control compiles his code, he also assigns it a unique identification string that enables programmers like you to specify exactly which control you want to embed in your IE ActiveX container.

Adding an ActiveX Control to the Page

Adding an ActiveX control to a page for an IE browser requires the use of the `<object/>` element. Two very important attributes of the `<object/>` element are common to all controls, namely `classid` and `codebase`. The `classid` attribute is the unique ID that the creator of the control gave to it when it was compiled. The `codebase` attribute gives a URL where the ActiveX control can be found — you'll look at this attribute in more detail in the next section.

How can you find out the `classid`? Well, one way to do this is by checking the documentation that came with the control or is available on the control creator's web site. If you have the control installed, another way to do this is via IE itself, which will tell you which controls are installed on the computer and available to IE. Also, IE gives you additional information such as `classid`, though it won't inform you about any controls that were installed with the operating system. For example, Flash 3 is installed with Windows 98 and therefore won't appear.

To get this information, open up IE and select Internet Options from the Tools menu, as shown in Figure 15-4.

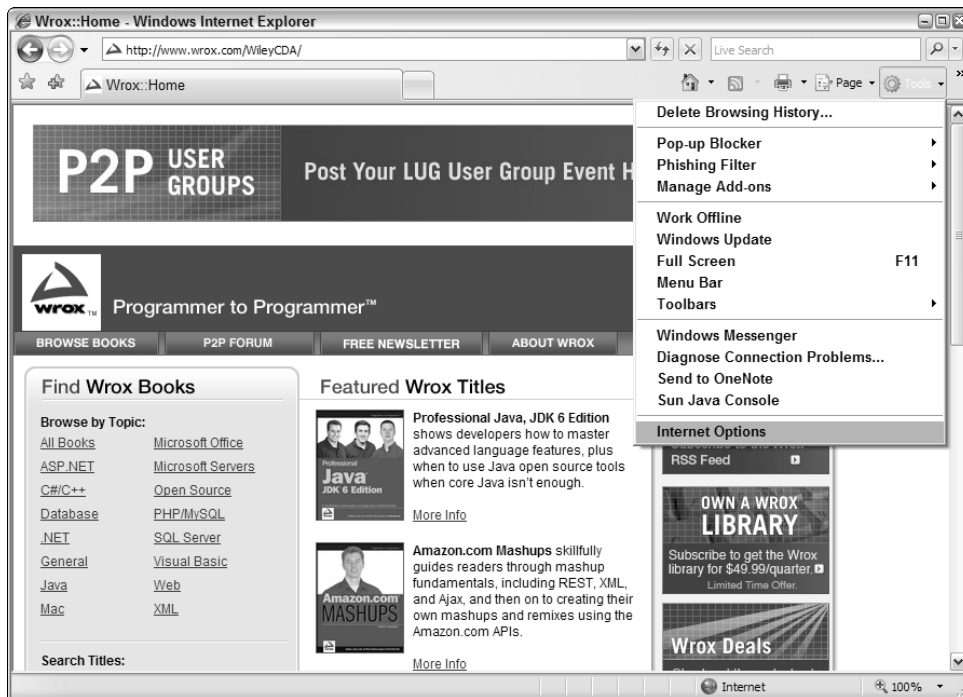


Figure 15-4

This opens up the console shown in Figure 15-5. In the Browsing history area, click the Settings button.

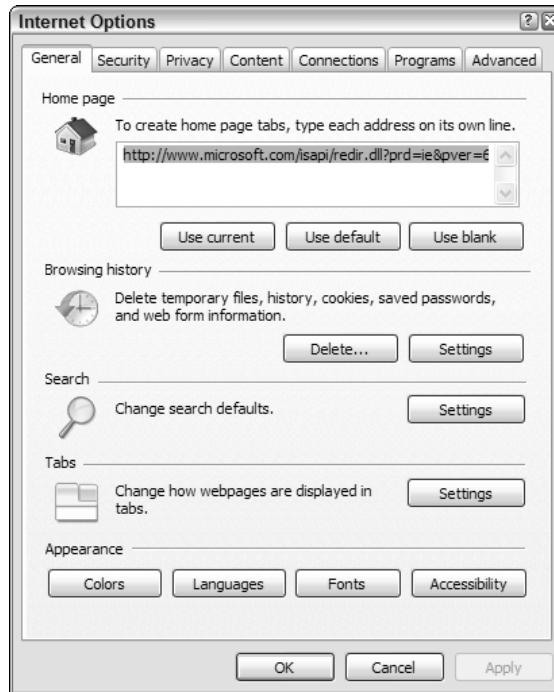


Figure 15-5

In the next console that opens, click the View objects button, shown in Figure 15-6.

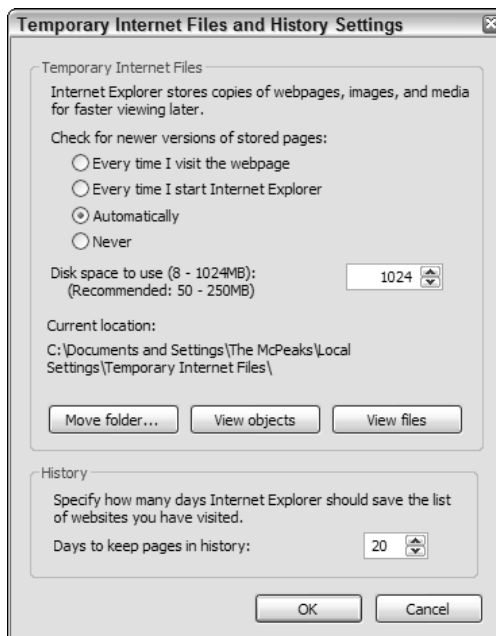


Figure 15-6

Chapter 15: Using ActiveX and Plug-Ins with JavaScript

This will display a list of all the ActiveX controls IE has installed from the Internet. The list shown in Figure 15-7 will most likely be different from that on your own computer.

You can see lots of information about each control, such as when it was created and its version number. However, to find out the `classid`, you need to right-click the name of the control you're interested in and select Properties from the menu that pops up.

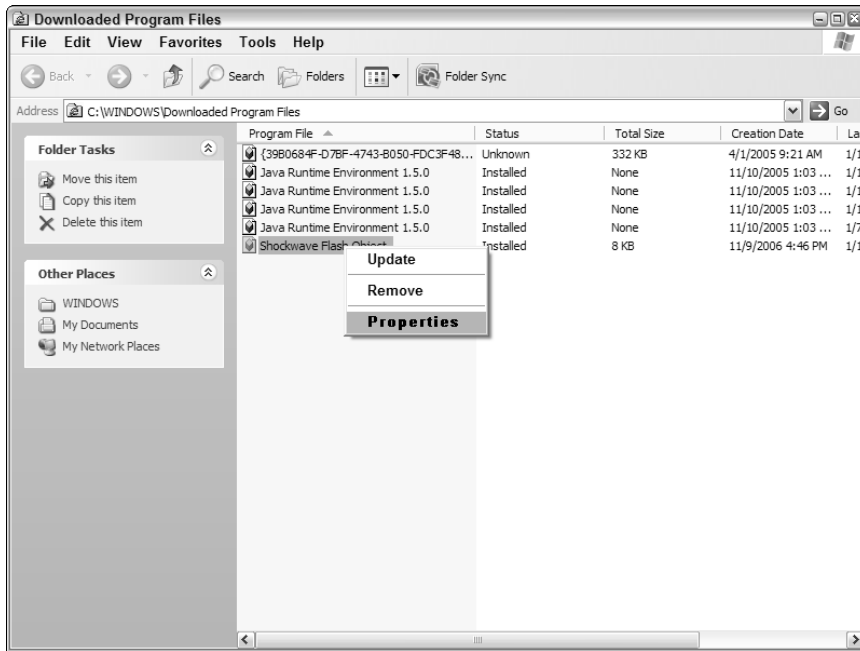


Figure 15-7

The information shown in Figure 15-8 is displayed, though this may be slightly different on your system.

You can see that the `classid` attribute, listed as just ID, and the `codebase` attribute, listed as CodeBase, are both displayed, although for `codebase` you may need to select the line and then scroll using the arrow keys to see all the information.

From this information, you see that to insert a Flash ActiveX control in your web page you need to add the following `<object/>` element:

```
<object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
  id=flashPlayer1
  width=500
  height=100>
</object>
```

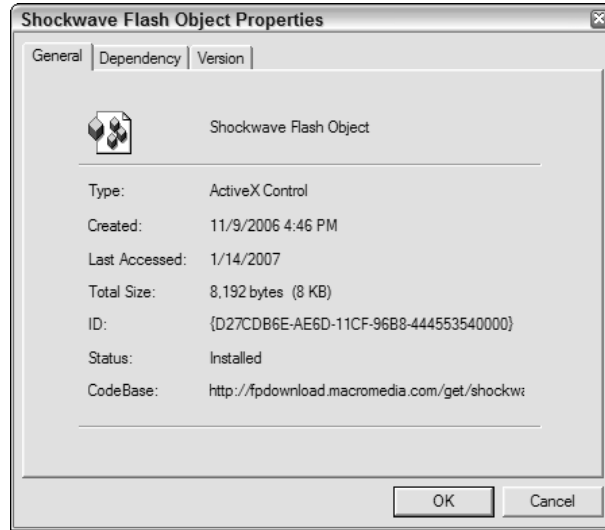



Figure 15-8

You can also set attribute or parameter values for the control itself. For example, with Flash you need to set the `src` attribute to point to the `.swf` file you want loaded, and you may also want to set the `quality` attribute, which determines the quality of appearance of the Flash movie. However, to set the parameters of the ActiveX control such as these (as opposed to the attributes of the `<object/>` element), you need to insert the `<param/>` element between the start `<object>` tag and the close `</object>` tag.

In each `<param/>` element you need to specify the name of the parameter you want to set and the value you want it set to. For example, if you want to set the `src` attribute to `myFlashMovie.swf`, you need to add a `<param/>` element like this:

```
<param name=src value="myFlashMovie.swf">
```

Let's add this to the full `<object/>` element definition and also define the `quality` attribute at the same time.

```
<object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
  id=flashPlayer1
  width=500
  height=100>
  <param name=src value="myFlashMovie.swf">
  <param name=quality value=high>
</object>
```

Alternatively, you can use the `movie` parameter instead of `src` parameter.

Installing an ActiveX Control

You've seen how to insert an ActiveX control into your page, but what happens if the user doesn't have that control installed on her computer?

This is where the `codebase` attribute of the `<object>` tag comes in. If the browser finds that the ActiveX control is not installed on the user's computer, it will try to download and install the control from the URL pointed to by the `codebase` attribute.

The creator of the ActiveX control will usually have a URL that you can use as a value for the `codebase` attribute. The information under the Internet Options option of the Tools menu you saw earlier provides the `codebase` for the control that was installed on your computer, though this may not necessarily be the best URL to use, particularly if it's not a link to the creator of the control.

For Flash, the `codebase` is `http://fpdownload.macromedia.com/get/shockwave/cabs/flash/swflash.cab`, so your `<object>` tag will look like this:

```
<object classid="clsid:D27CDB6E-AE6D-11CF-96B8-444553540000"
codebase="http://fpdownload.macromedia.com/get/shockwave/cabs/flash/swflash.cab "
id="flashPlayer1"
width=500
height=100 >
  <param name=src value="myFlashMovie.swf">
  <param name=quality value=high>
</object>
```

Subject to license agreements, you may be able to download the `.cab` file that installs the control to your own server and point the `codebase` attribute to that.

Unfortunately, there is no easy foolproof way of checking which ActiveX controls are installed on the user's computer. However, the Object object of the `<object/>` element does have the `readyState` property. This returns 0, 1, 2, 3, or 4, indicating the object's operational status. The possible values are as follows:

- ☐ 0 — Control is un-initialized and not ready for use
- ☐ 1 — Control is still loading
- ☐ 2 — Control has finished loading its data
- ☐ 3 — User can interact with control even though it is not fully loaded
- ☐ 4 — Control is loaded and ready for use

You need to give the control time to load before checking its `readyState` property, so any checking is best left until the window's `onload` event handler or even the document object's `onreadystatechange` event handler has fired.

To redirect the user to another page that doesn't need the control, you need to write this:

```
var flashPlayer1;

//code to retrieve ActiveX plug-in

if (flashPlayer1.readyState == 0)
```

```
{
    window.location.replace("NoControlPage.htm");
}
```

Attach this code to the window's onload event handler.

You saw that the `<noembed/>` element enables you to display a message for users with browsers that do not support plug-ins. This element works in exactly the same way with IE. Alternatively, you can place text between the `<object>` and `</object>` tags, in which case it will only be displayed if the browser is not able to display the ActiveX control.

Using Plug-ins and ActiveX Controls

When you have the plug-ins or ActiveX controls embedded into the page, their actual use is very uniform. To make life easier for you, most developers of plug-ins and controls make the properties, methods, and events supported by each plug-in and ActiveX control similar. However, it's important to check the developer's documentation because it's likely that there will be some idiosyncracies.

Inside the `<embed/>` or `<object/>` element, you give your plug-in or control a unique `id` value. You can then access the corresponding object's methods, properties, and events just as you would for any other tag. The actual properties, methods, and events supported by a plug-in or control will be specific to that control, but let's look at one of the more commonly available controls, RealNetworks RealPlayer, which comes in both plug-in form for Firefox and ActiveX control form for IE. You can find more information on this control at www.realnetworks.com/resources/index.html, and you can download a free version (RealOne Player) from www.realnetworks.com. Note that you can buy a version with more features, but if you search the web site, a no-charge basic version is available.

First you need to embed the control in a web page. Type the following into a text editor:

```
<html>
<head>
<object classid="clsid:CFCDA03-8BE4-11CF-B84B-0020AFBCCFA" id="real1"
width="0" height="0">
  <param name="height" value="0">
  <param name="width" value="0">
  <embed name="real1"
    id="real1"
    border="0"
    controls="play"
    height=0
    width=0
    type="audio/x-pn-realaudio-plugin">
</object>
</head>
<body>
<noembed>
  <h2>This Page requires a browser supporting Plug-ins or ActiveX controls</h2>
</noembed>
</body>
</html>
```

Chapter 15: Using ActiveX and Plug-Ins with JavaScript

Save this code as `realplayer.htm`.

The first thing to note is that the `<embed/>` element for the Firefox plug-in has been inserted between the opening `<object>` tag and the closing `</object>` tag. This is because Firefox will ignore the `<object/>` element and display only the plug-in defined in the `<embed/>` element. IE, on the other hand, will ignore the `<embed/>` element inside the `<object/>` element, but IE will display the ActiveX control.

IE does support the `<embed/>` element, though its use is discouraged. This means that if you placed the `<embed/>` element outside the `<object/>` element, IE would recognize both the `<object/>` and `<embed/>` elements and get confused — particularly over the `id` values, because both have the same `id` of `real1`.

You've placed the `<embed/>` and `<object/>` elements inside the `<head/>`. Why?

Well, for this example you don't want to display the graphical interface of the control itself; instead you want to use only its ability to play sounds. By placing it inside the head of the document, you make it invisible to the user.

Finally, you've put a `<noembed/>` element inside the body of the page for users with browsers that don't support plug-ins or ActiveX controls. This will display a message to such users, telling them why they are staring at a blank page!

You want to make sure that users without the RealPlayer plug-in or control don't see error messages when you start scripting the controls. So let's redirect them to another page if they do not have the right plug-in or control. You'll do this by adding a function to check for the availability of the plug-in or control and attaching this to the window object's `onload` event handler in the `<body>` tag.

```
<html>
<head>
<object classid="clsid:CFCDA03-8BE4-11CF-B84B-0020AFBBCCFA" id="real1"
width="0" height="0">
  <param name="height" value="0">
  <param name="width" value="0">
  <embed name="real1"
    id="real1"
    border="0"
    controls="play"
    height=0
    width=0
    type="audio/x-pn-realaudio-plugin">
</object>
<script type="text/javascript">
function window_onload()
{
  var plugInInstalled = false;
  if (navigator.appName.indexOf("Microsoft") == -1)
  {
    var plugInCounter;
    for (plugInCounter = 0; plugInCounter < navigator.plugins.length;
      plugInCounter++)
    {
```

```
        if (navigator.plugins[plugInCounter].name.indexOf("RealPlayer") >= 0)
        {
            plugInInstalled = true;
            break;
        }
    }
}
else
{
    if (real1.readyState == 4)
    {
        plugInInstalled = true;
    }
}
if (plugInInstalled == false) {
    window.location.replace("NoRealPlayerPage.htm");
}
}
</script>
</head>
<body onload="return window_onload()">
<noembed>
    <h2>This Page requires a browser supporting Plug-ins or ActiveX controls</h2>
</noembed>
</body>
</html>
```

In the `window_onload()` function, you first define a variable, `plugInInstalled`, and initialize it to `false`.

Next, since checking for plug-ins or controls is browser-dependent, you check to see if this is a Microsoft browser. If not, you assume it's Firefox, though for a real-life example you might want to do more detailed checks.

If the browser is Firefox, you use a `for` loop to go through the `navigator` object's `plugins` array, checking each installed plug-in's name for the word `RealPlayer`. If this word is found, you set the variable `plugInInstalled` to `true` and break out of the `for` loop.

If you find that this is a Microsoft browser, you use the `readyState` property of the `<object/>` element's `Object` object to see if the ActiveX control is loaded, initialized successfully, and now ready for action. If its value is 4, you know that all systems are ready to go, and you can use the control, so you set the variable `plugInInstalled` to `true`.

Finally, the last `if` statement in the function checks to see if `plugInInstalled` is `true` or `false`. If it is `false`, the user is redirected to another page, called `NoRealPlayerPage.htm`, where you can either provide alternative ways to display the content or provide a link to load the `RealPlayer` control. Let's create a simple page to do this.

```
<html>
<head>
    <title>No Real Player Installed</title>
</head>
```

```
<body>
  <h2>You don't have the required RealPlayer plug-in</h2>
  <p>
    You can download the plug-in from
    <a href="http://www.real.com">Real Player</a>
  </p>
</body>
</html>
```

Save this as `NoRealPlayerPage.htm`.

Finally, back in the `realplayer.htm` page, let's enable the user to select a sound file as well as to start and stop playing it. You add the following code to the top of the script block:

```
<html>
<head>
<object classid="clsid:CFCDA03-8BE4-11CF-B84B-0020AFBCCFA" id="real1"
width="0" height="0">
  <param name="height" value="0">
  <param name="width" value="0">
  <embed name="real1"
    id="real1"
    border="0"
    controls="play"
    height=0
    width=0
    type="audio/x-pn-realaudio-plugin">
</object>
<script type="text/javascript">
var fileName = "";

function butPlay_onclick()
{
  document.real1.SetSource("file:/// " + fileName);
  document.real1.DoPlay();
}

function butStop_onclick()
{
  document.real1.DoStop();
}

function file1_onblur()
{
  fileName = document.form1.file1.value;
}

function window_onload()
{
  var plugInInstalled = false;
  if (navigator.appName.indexOf("Microsoft") == -1)
  {
    var plugInCounter;
    for (plugInCounter = 0; plugInCounter < navigator.plugins.length;
```

```
        plugInCounter++)
    {
        if (navigator.plugins[plugInCounter].name.indexOf("RealPlayer") >= 0)
        {
            plugInInstalled = true;
            break;
        }
    }
}
else
{
    if (real1.readyState == 4)
    {
        plugInInstalled = true;
    }
}
if (plugInInstalled == false) {
    window.location.replace("NoRealPlayerPage.htm");
}
}
</script>
</head>
<body onload="return window_onload()">
<noembed>
    <h2>This Page requires a browser supporting Plug-ins or ActiveX controls</h2>
</noembed>
</body>
</html>
```

You also add a form with buttons for starting, stopping, and choosing a sound file to the body of the page.

```
<html>
<head>
<object classid="clsid:CFCDA03-8BE4-11CF-B84B-0020AFBCCFA" id="real1"
width="0" height="0">
    <param name="height" value="0">
    <param name="width" value="0">
    <embed name="real1"
        id="real1"
        border="0"
        controls="play"
        height=0
        width=0
        type="audio/x-pn-realaudio-plugin">
</object>
<script type="text/javascript">
var fileName = "";

function butPlay_onclick()
{
    document.real1.SetSource("file://" + fileName);
    document.real1.DoPlay();
}

function butStop_onclick()
```

Chapter 15: Using ActiveX and Plug-Ins with JavaScript

```
{
    document.reall.DoStop();
}

function file1_onblur()
{
    fileName = document.form1.file1.value;
}

function window_onload()
{
    var plugInInstalled = false;
    if (navigator.appName.indexOf("Microsoft") == -1)
    {
        var plugInCounter;
        for (plugInCounter = 0; plugInCounter < navigator.plugins.length;
            plugInCounter++)
        {
            if (navigator.plugins[plugInCounter].name.indexOf("RealPlayer") >= 0)
            {
                plugInInstalled = true;
                break;
            }
        }
    }
    else
    {
        if (reall.readyState == 4)
        {
            plugInInstalled = true;
        }
    }
    if (plugInInstalled == false) {
        window.location.replace("NoRealPlayerPage.htm");
    }
}
</script>
</head>
<body onload="return window_onload()">
<noembed>
    <h2>This Page requires a browser supporting Plug-ins or ActiveX controls</h2>
</noembed>
<form id=form1 name=form1>
    <input type="button" value="Play Sound" id="butPlay" name="butPlay"
        onclick="return butPlay_onclick()">
    <input type="button" value="Stop Sound" id="butStop" name="butStop"
        onclick="return butStop_onclick()">
    <input type="file" id="file1" name="file1"
        onblur="return file1_onblur()">
</form>

</body>
</html>
```


Chapter 15: Using ActiveX and Plug-Ins with JavaScript

You've completed your page, so let's now resave it. Load `realplayer.htm` into your browser and, as long as your browser supports plug-ins or ActiveX controls and the RealPlayer plug-in is installed, you should see something like what is shown in Figure 15-9.

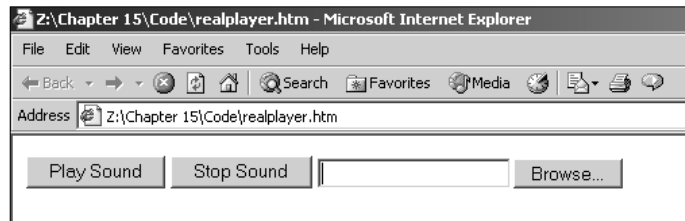


Figure 15-9

Click the Browse button and browse to an MP3 sound file (`Budd_Eno.mp3` is provided with the code download, or you can create your own with Real Producer Basic, also available at www.realnetworks.com). Click the Play Sound and Stop Sound buttons to play and stop the sound.

So how does this work?

The form in the body of the page contains three form elements. The first two of these are just standard buttons, but the last is an `<input />` element of `type="file"`. This means that a text box and a button are displayed. When the button is clicked, a Choose File dialog box opens, enabling you to choose the `.ra` file you want to hear. When chosen, this file's name appears in the text box.

You've connected the two buttons' `onclick` event handlers and the file control's `onblur` event handler to three functions, `butPlay_onclick()`, `butStop_onclick()`, and `file1_onblur()`, respectively, which are defined in the script block in the head of the page.

In the function `file1_onblur()`, you set a global variable, `fileName`, to the value of the file control. In other words, `fileName` will contain the name and path of the file the user has chosen to play. The `blur` event will fire whenever the user moves focus from the file control to another control or area of the page. In reality, you would perform checks to see whether the file type selected by the user is actually a valid sound file.

In the other two functions, you access the RealPlayer plug-in or control that you embedded in the page. You use one function to load the file the user selected and play it, and the other function to stop play.

In both functions, you access the RealPlayer control by using its name prefixed with `document`. The script will work with IE and Firefox, though under IE it's accessing the ActiveX control defined in `<object />`, and in Firefox it's accessing the plug-in defined in the `<embed />` tag.

In the `butPlay_onclick()` function, you use the `SetSource()` method of the RealPlayer object. This method takes one parameter—the file that you want the RealPlayer plug-in to load. So, in the line

```
document.real1.SetSource("file://" + fileName);
```

you load the file the user specified. Next you use the `DoPlay()` method of the `RealPlayer` object, which starts the playing of the source file.

```
document.real1.DoPlay();
```

With the function `butStop_onclick()`, you stop the playing of the clip using the `DoStop()` method of the `RealPlayer` object.

```
document.real1.DoStop();
```

Testing Your “No Plug-in or ActiveX Control” Redirection Script

It’s quite likely that if you plan to use an ActiveX control or plug-in, you’re going to make sure it’s installed on your computer. The problem is that while that’s great for testing pages to see if they work when there is a control installed, it does make it very difficult to check redirection scripts for users without that control. You have a number of possible options:

1. Get a second computer with a clean install of an operating system and browser, then load your pages on that computer. This is the only 100 percent sure way of checking your pages.
2. Uninstall the plug-in. Depending on how the plug-in or control was installed, there may be an uninstall program for it. Windows users can use the Add/Remove programs option in the Control Panel.
3. For Firefox, install a different version of the browser. For example, if you have Firefox 2 installed, try installing an older version, say Firefox 1 or even a beta version if you can find it. The plug-ins currently installed are not normally available to a browser installed later, though this may not be true all the time.
4. With IE, you can only have one version of the browser installed at once. However, IE does make it quite easy to remove ActiveX controls. In IE 5+, choose Internet Options from the Tools menu. Click the Settings button under Temporary Internet Files (Browsing History in IE7), followed by the View Objects button. From here you need to right-click the name of the control you want removed and select Remove from the pop-up menu.

Potential Problems

Plug-ins and ActiveX controls provide a great way to extend a browser’s functionality, but they do so at a price — compatibility problems. Some of the problems you may face are discussed in the following sections.

Similar but Not the Same — Differences Between Browsers

Although a plug-in for Firefox and the equivalent ActiveX control for IE may support many similar properties and methods, you will often find significant, and sometimes subtle, differences.

For example, both the plug-in and ActiveX control versions of `RealPlayer` support the `SetSource()` method. However, while

```
document.real1.SetSource("file:///D:\\MyDir\\MyFile.ra")
```

will work with IE, it will cause problems with the other browsers. To work with Firefox and the like, specify the protocol by which the file will be loaded. If it is a URL, specify `http://`, but for a file on a user's local hard drive, use `file:///`.

To make the code work across platforms, you must type this:

```
document.real1.SetSource("file:///D:\MyDir\MyFile.ra")
```

Differences in the Scripting of Plug-ins

When scripting the RealPlayer plug-in for Firefox, you embedded it like this:

```
<embed name="real1" id="real1"
  border="0"
  controls="play"
  height=0 width=0 type="audio/x-pn-realaudio-plugin">
```

You then accessed it via script just by typing this:

```
document.real1.DoPlay()
```

However, if you are scripting a Flash player, you need to add the following attribute to the `<embed/>` definition in the HTML:

```
swliveconnect="true"
```

Otherwise any attempts to access the plug-in will result in errors.

```
<embed name="map"
  swLiveConnect=true
  src="topmenu.swf"
  width=300 height=200

  pluginspage="http://fpdownload.macromedia.com/get/shockwave/cabs/flash/swflash.cab"
  Version=ShockwaveFlash">
```

It's very important to study any available documentation that comes with a plug-in to check that there are no subtle problems like this.

Differences Between Operating Systems

Support for ActiveX controls varies greatly between different operating systems. IE for the Mac supports it, but not as well as under Win32 operating systems, such as Windows 95, 98, 2000, and XP.

You also need to be aware that an ActiveX control written for Win32 will not work on the Mac; you need to make sure a Mac-specific control is downloaded.

IE on the Mac supports plug-ins as well as ActiveX controls; so, for example, Flash is a plug-in on the Mac and an ActiveX control on Win32. Clearly, if you want to support both Mac and Windows users, you need to write more complex code.

It's very important to check which operating system the user is running (for example, using the scripts given at the end of Chapter 5) and deal with any problems that may arise.

Differences Between Different Versions of the Same Plug-in or ActiveX Control

Creators of plug-ins and controls will often periodically release new versions with new features. If you make use of these new features, you need to make sure not only that the user has the right plug-in or ActiveX control loaded, but also that it is the right version.

ActiveX Controls

With ActiveX controls, you can add version information in the `codebase` attribute of the `<object/>` element.

```
<object classid=clsid:AAA03-8BE4-11CF-B84B-0020AFBCCFA
  id="myControl"
  codebase="http://myserver/mycontrol.cab#version=3,0,0,0">
</object>
```

Now, not only will the browser check that the control is installed on the user's system, but it'll also check that the installed version is version 3 or greater.

What if you want to check the version and then redirect to a different page if it's a version that is earlier than your page requires?

With ActiveX controls there's no easy way of using JavaScript code to check the ActiveX control version. One way is to find a property that the new control supports but that older versions don't, and then compare that to `null`. For example, imagine you have a control whose latest version introduces the property `BgColor`. To check if the installed version is the one you want, you type the following:

```
if (document.myControl.BgColor == null)
{
  alert("This is an old version");
}
```

It's also possible that the ActiveX creator has added to his control's object a `version` property of some sort that you can check against, but this will vary from control to control.

Plug-ins

With plug-ins you need to make use of the `Plugin` objects in the `navigator` object's `plugins[]` array property. Each `Plugin` object in the array has a `name`, `filename`, and `description` property, which may provide version information. However, this will vary between plug-ins.

For example, for Flash Player 4 on Win32, the description given by

```
navigator.plugins["Shockwave Flash"].description
```

is `Flash 4.0 r7`.

Using regular expressions, which were introduced in Chapter 8, you could extract the version number from this string:

```
var myRegExp = /\d{1,}.\d{1,}/;  
var flashVersion = navigator.plugins["Shockwave Flash"].description;  
flashVersion = parseFloat(flashVersion.match(myRegExp)[0]);
```

In the first line of code you define a regular expression that will match one or more digits, followed by a dot, and then one or more numbers. Next you store the description of the Flash plug-in in the variable `flashVersion`. Finally you search the variable for the regular expression, returning an array of all the matches made. You then use the `parseFloat()` function on the contents of the element in the array at index 0 (in other words, the first element in the array).

Changes to Internet Explorer 6 Service Pack 1b and ActiveX Controls

For mostly legal reasons, Microsoft is making changes to how ActiveX controls work in IE. Now whenever a user browses to a page with an ActiveX control, she gets a warning about the control, and by default it's blocked unless she chooses to unblock it. There are two ways around this:

1. Don't access any external data or have any `<param/>` elements in the definition, as the following example demonstrates:

```
<object classid="CLSID:6BF52A52-394A-11d3-B153-00C04F79FAA6"></object>
```

2. Use the new `noexternaldata` attribute to specify that no external access of data is used.

```
<object noexternaldata="true" classid="CLSID:6BF52A52-394A-11d3-B153-00C04F79FAA6">  
  <param  
    name="URL"  
  
    value="http://msdn.microsoft.com/workshop/samples/author/dhtml/media/drums.wav"/>  
</object>
```

The URL parameter will be ignored, and no external data from the URL, in this case a `.wav` file, will be accessed.

Summary

In this chapter you looked at how you can use plug-ins and ActiveX controls to extend a browser's functionality. You saw that:

- ❑ Internet Explorer supports ActiveX controls, and to some extent plug-ins, on Windows operating systems. Firefox has good support for plug-ins but does not support ActiveX controls.
- ❑ Most creators of plug-ins also provide an ActiveX control equivalent. Internet Explorer and Firefox are incompatible as far as the installation of plug-ins and ActiveX controls goes.
- ❑ Plug-ins are embedded in a web page by means of the `<embed/>` element. You let Firefox know which plug-in is to be embedded by specifying either a `source` file or a `MIME` type using the `src` and `type` attributes of the `<embed/>` element. If you define a value for the `<embed/>` element's `pluginspage` attribute, users who don't have that plug-in installed will be able to click a link and install it.
- ❑ You can find detailed information about what plug-ins are installed on your Firefox browser, as well as their descriptions and types, by using the About Plug-ins option on the Help menu.

- ❑ To use script to check if a user has a certain plug-in, you can use the navigator object's `plugins[]` array property. For each plug-in installed, there will be a `Plugin` object defined in this array. Each `Plugin` object has the properties `name`, `description`, `filename`, and `length`, which you can use to determine if a plug-in exists on the user's computer. You can also use the navigator object's `mimeTypes[]` array property to check if a certain type of file is supported.
- ❑ Internet Explorer supports ActiveX controls as an alternative to plug-ins. These are embedded into a web page using the `<object/>` element. You specify which ActiveX control you want by using the `classid` attribute. If you want to have controls automatically install for users who don't have a particular control already installed, you need to specify the `codebase` attribute.
- ❑ Any parameters particular to the control are specified by means of the `<param/>` element, which is inserted between the opening and closing `<object>` tags.
- ❑ You can check whether a control has loaded successfully using the `readyState` property of the `Object` object, which returns a number: 0 if the control is not installed, 1 if it's still loading, 2 if it has loaded, 3 if you can interact with it, and 4 if it's installed and ready for use.
- ❑ Virtually every different type of plug-in and ActiveX control has its own interface, for which the control's documentation will provide the details. You looked briefly at the `RealPlayer` control by `RealNetworks`.
- ❑ You also saw that while plug-ins and controls are great for extending functionality, they are subject to potential pitfalls. These include differences in the way plug-ins and ActiveX controls are scripted, differences in operating systems, and differences between versions of the same plug-in or control.

In the next chapter, you change direction to cover a “new” JavaScript technique that has rekindled web application development.

Exercise Question

A suggested solution to this question can be found in Appendix A.

Question 1

Using the `RealPlayer` plug-in/ActiveX control, create a page with three links, so that when the mouse pointer rolls over any of them a sound is played. The page should work in Firefox and IE. However, any other browsers should be able to view the page and roll over the links without errors appearing.

16

Ajax and Remote Scripting

Since its inception, the Internet has used a transaction-like communication model. A browser sends a request to a server, which sends a response to the browser, on which it (re)loads the page. This is typical HTTP communication, and it was designed to be this way, but this model is rather cumbersome for developers, as it requires our web applications to consist of several pages. The resulting user experience is disjointed and interrupted.

In the early 2000s, a movement began to look for and develop new techniques to enhance the user's experience, as well as make applications easier to build and maintain. These new techniques offered performance and usability usually associated with conventional desktop applications. It wasn't long before developers began to refine these processes to offer richer functionality.

At the heart of this movement was one language: JavaScript, and its remote scripting ability.

What Is Remote Scripting?

Essentially, *remote scripting* allows client-side JavaScript to request and receive data from a server without refreshing the web page. This technique enables the developer to create an application that is uninterrupted, making only portions of the page reload with new data.

The term “remote scripting” is quite large in scope, as a variety of JavaScript techniques can be used. These techniques incorporate the use of hidden frames/iframes, dynamically adding `<script/>` elements to the document, and/or using JavaScript to send HTTP requests to the server; the latter has become quite popular in the last couple of years. These new techniques refresh only portions of a page, both cutting the size of data sent to the browser and making the web page feel more like a conventional application.

What Can It Do?

The concept of remote scripting opens the doors for advanced web applications — ones that mimic desktop applications in form and in function.

Chapter 16: Ajax and Remote Scripting

A variety of commercial web sites employ the use of remote scripting. These sites look and behave more like desktop applications than like web sites, but that is the whole point of remote scripting. The most notable remote scripting-enabled web applications come from the search giant Google: Google Maps and Google Suggest.

Google Maps

Designed to compete with existing commercial mapping sites (and using images from its Google Earth), Google Maps (<http://maps.google.com>) uses remote scripting to dynamically add map images to the web page. When you enter a location, the main page does not reload at all; the images are dynamically loaded in the map area. Google Maps also enables you to drag the map to a new location, and once again, the map images are dynamically added to the map area (see Figure 16-1).

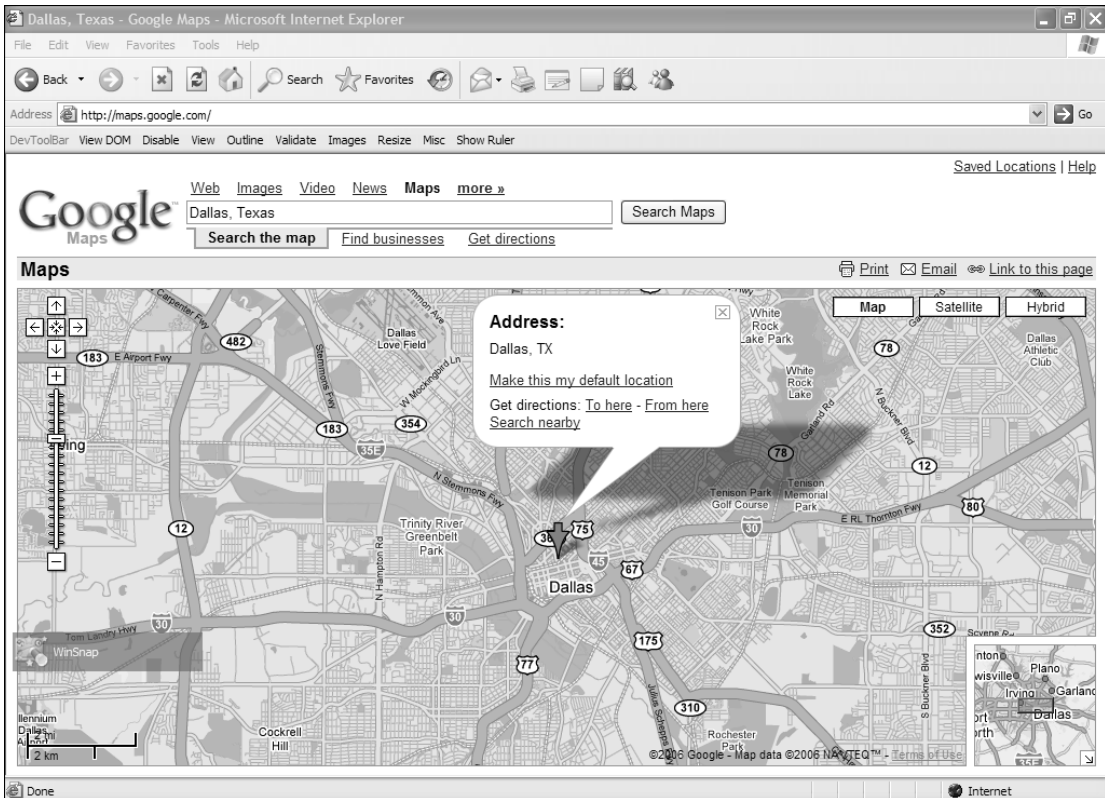


Figure 16-1

Google Suggest

Google Suggest (<http://labs.google.com/suggest/>) is another Google innovation that employs the use of remote scripting. Upon first glance, it appears to be a normal Google search page. When you start typing, however, a drop-down box displays suggestions for search terms that might interest you. To the right of the suggested word or phrase is the number of results the search term returns (see Figure 16-2).

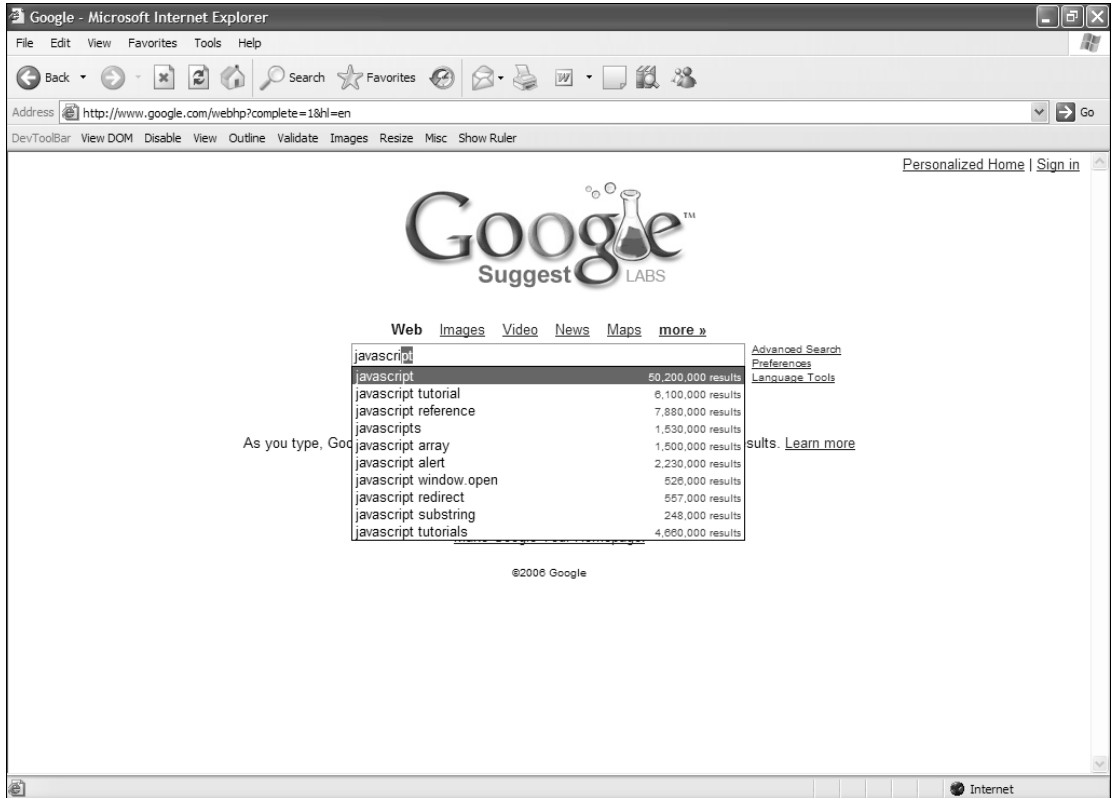


Figure 16-2

Ajax

In 2005, Jesse James Garrett wrote an article entitled “Ajax: A New Approach to Web Applications” (www.adaptivepath.com/publications/essays/archives/000385.php). In it, Garrett states that the interactivity gap between web and desktop applications is becoming smaller, and he cites Google Maps and Google Suggest as proof of this. He christened the new approach *Asynchronous JavaScript + XML*, or *Ajax*. Although the new term was coined in 2005, the underlying methodology of Ajax, which is remote scripting, has been around for many years. In fact, many developers use the terms *remote scripting* and *Ajax* interchangeably. The entire concept of Ajax can be boiled down to client-server communication — which is remote scripting.

For all intents and purposes, when Ajax is mentioned in this chapter, think of remote scripting.

Browser Support

Ajax is limited to the browser that runs the web application, and like every other advanced JavaScript concept we’ve covered in this book, Ajax capabilities differ from browser to browser. Thankfully, the most common forms of Ajax work in the following browsers:

- ☐ Internet Explorer 5+
- ☐ Firefox 1+
- ☐ Opera 9+
- ☐ Safari 2+

When using hidden frames, a popular Ajax approach, with these browsers, you'll notice few differences in the code, as each BOM handles frames the same way (we'll cover this approach later in the chapter). However, when you start using other forms of Ajax, the differences in code become apparent.

Ajax with JavaScript: The XMLHttpRequest Object

As stated before, there are a variety of ways that you can create Ajax-enabled applications. However, probably the most popular Ajax technique is using the JavaScript `XMLHttpRequest` object, which is present in IE 5+, Firefox, Opera, and Safari.

Despite its name, you can retrieve other types of data, like plain text, with `XMLHttpRequest`.

The `XMLHttpRequest` object originated as a Microsoft component, called `XmlHttp`, in the MSXML library first released with IE 5. It offered developers an easy way to open HTTP connections and retrieve XML data. Microsoft improved the component with each new version of MSXML, making it faster and more efficient.

As the popularity of the Microsoft `XMLHttpRequest` object grew, Mozilla decided to include its own version of the object with Firefox. The Mozilla version maintained the same properties and methods used in Microsoft's ActiveX component, making cross-browser usage possible. Soon after, Opera Software and Apple copied the Mozilla implementation, thus bringing the easy-to-use object to all modern browsers.

Cross-Browser Issues

The `XMLHttpRequest` object is no different from other web standards supported by the browsers, and the differences can be divided into two camps: the IE 5 and 6 method, and the IE 7, Firefox, Opera, and Safari method. Thankfully, the two browser types only differ when you need to create an `XMLHttpRequest` object. After the object's creation, the remainder of the code is compatible for every browser.

IE 5 and IE 6: Using ActiveX

Because the `XMLHttpRequest` object originated as a part of the MSXML library, an ActiveX XML parser, instantiating an `XMLHttpRequest` under these browsers, requires the creation of an ActiveX object. In Chapter 14, you created ActiveX objects to traverse the XML DOM. Creating an `XMLHttpRequest` object isn't much different.

```
var oHttp = new ActiveXObject("Microsoft.XMLHttp");
```

This line creates the first version of Microsoft's XMLHttpRequest. There are many other versions of Microsoft's XMLHttpRequest, but Microsoft recommends using one of the following versions:

- ❑ MSXML2.XmlHttp.6.0
- ❑ MSXML2.XmlHttp.3.0

You want to use the latest version possible when creating an XMLHttpRequest object; the latest version contains bug fixes and enhanced performance. The downside to this is that not everyone will have the same version installed on their computer. However, you can write a function that can do this.

With the previous version information, you'll write a function called `createXmlHttpRequest()` to create an XMLHttpRequest object with the latest version supported by the user's computer.

```
function createXmlHttpRequest()
{
    var versions =
    [
        "MSXML2.XmlHttp.6.0",
        "MSXML2.XmlHttp.3.0"
    ];
    //more code here
}
```

This code defines the `createXmlHttpRequest()` function. Inside it, an array called `versions` is created to contain the different version names recommended by Microsoft. Notice that the version names are listed starting with the newest first. This is done because you always want to check for the newest version first and continue down the list until you find the version installed on the computer.

To decide what version to use, use a `for` loop to loop through the elements in the array and then attempt to create an XMLHttpRequest object.

```
function createXmlHttpRequest()
{
    var versions =
    [
        "MSXML2.XmlHttp.6.0",
        "MSXML2.XmlHttp.3.0"
    ];

    for (var i = 0; i < versions.length; i++)
    {
        try
        {
            var oHttp = new ActiveXObject(versions[i]);
            return oHttp;
        }
        catch (error)
        {
            //do nothing here
        }
    }
    //more code here
}
```

Chapter 16: Ajax and Remote Scripting

A `try...catch` block is used inside the loop. Unfortunately, this is the only way to determine if a version is installed on the computer. If the following line fails:

```
var oHttp = new ActiveXObject(versions[i]);
```

the code execution drops to the `catch` block, and since nothing happens in this block, the loop iterates to the next element in the array until a version is found to work or the loop exits. If no version is found on the computer, then the function returns `null`, like this:

```
function createXmlHttpRequest()
{
    var versions =
    [
        "MSXML2.XmlHttp.6.0",
        "MSXML2.XmlHttp.3.0"
    ];

    for (var i = 0; i < versions.length; i++)
    {
        try
        {
            var oHttp = new ActiveXObject(versions[i]);
            return oHttp;
        }
        catch (error)
        {
            //do nothing here
        }
    }

    return null;
}
```

Now you don't have to worry about ActiveX objects to create an object. If you call this function, it'll do all the work for you.

```
var oHttp = createXmlHttpRequest();
```

The Other Browsers: Native Support

Unlike IE 5 and IE 6, the IE 7, Firefox, Opera, and Safari browsers boast a native implementation of the `XMLHttpRequest` object, in that it is an object located in the `window` object. Creating an `XMLHttpRequest` object is as simple as calling its constructor.

```
var oHttp = new XMLHttpRequest();
```

This line creates an `XMLHttpRequest` object, which you can use to connect to, and request and receive data from, a server. Unlike the ActiveX object in the previous section, `XMLHttpRequest` does not have different versions. Simply calling the constructor creates the object, and it is ready to use.

Playing Together

Just as with all other cross-browser problems, a solution can be found to create an `XMLHttpRequest` object in a cross-browser way. You already wrote the `createXmlHttpRequest()` function, so you can expand it to provide cross-browser functionality.

```
function createXmlHttpRequest()
{
    if (window.XMLHttpRequest)
    {
        var oHttp = new XMLHttpRequest();
        return oHttp;
    }
    else if (window.ActiveXObject)
    {
        var versions =
        [
            "MSXML2.XmlHttp.6.0",
            "MSXML2.XmlHttp.3.0"
        ];

        for (var i = 0; i < versions.length; i++)
        {
            try
            {
                var oHttp = new ActiveXObject(versions[i]);
                return oHttp;
            }
            catch (error)
            {
                //do nothing here
            }
        }
    }

    return null;
}
```

This new code first checks to see if `window.XMLHttpRequest` exists. If it does, then an `XMLHttpRequest` object can be created with the `XMLHttpRequest` constructor. If this first test fails, then the code checks for `window.ActiveXObject` for IE 5 and 6 and tries to create an object with the latest MSXML version. If no `XMLHttpRequest` object can be created (both for IE 5 and 6 and the other browsers), then the function returns `null`.

The order in which browsers are tested is important; test for `window.XMLHttpRequest` first because of IE 7, as the browser supports both `window.XMLHttpRequest` and `window.ActiveXObject`.

With this function, you can create an `XMLHttpRequest` object easily in both browsers. For example:

```
var oHttp = createXmlHttpRequest();
```

Regardless of the user's browser, if it supports `XMLHttpRequest`, an object will be created.

Using the XMLHttpRequest Object

Once you create the XMLHttpRequest object, you are ready to start requesting data with it. The first step in this process is to call the `open()` method to initialize the object.

```
oHttp.open(requestType, url, async);
```

This method accepts three arguments. The first, `requestType`, is a string value consisting of the type of request to make. The values can be either `GET` or `POST`. The second argument is the URL to send the request to, and the third is a `true` or `false` value indicating whether the request should be made in asynchronous mode. We discussed synchronous and asynchronous modes in Chapter 14, but as a refresher, requests made in synchronous mode halt all JavaScript code from executing until a response is received from the server.

The next step is to send the request; do this with the `send()` method. This method accepts one argument, which is a string that contains the request body to send along with the request. GET requests do not contain any information, so you must pass `null` as an argument.

```
var oHttp = createXmlHttpRequest();
oHttp.open("GET", "http://localhost/myTextFile.txt", false);
oHttp.send(null);
```

This code uses a GET request to retrieve a file called `myTextFile.txt` in synchronous mode. When the `send()` method is called, the request is sent to the server.

The `send()` method requires an argument to be passed; even if it is `null`.

Each XMLHttpRequest object has a `status` property. This property contains the HTTP status code sent with the server's response. If the requested file is not found, then `status` is 404. If the request was successful, `status` is 200. Consider the following example:

```
var oHttp = createXmlHttpRequest();
oHttp.open("GET", "http://localhost/myTextFile.txt", false);
oHttp.send(null);

if (oHttp.status == 200)
{
    alert("The text file was found!");
}
else if (oHttp.status == 404)
{
    alert("The text file could not be found!");
}
else
{
    alert("An error occurred while attempting to retrieve the file!");
}
```

If the request was successful (the status is 200), then an alert box tells the user that the file was found. If it could not be found (status is 404), an alert box tells the user that the file could not be found. Finally, an alert box tells the user that an error occurred if the status code happened to be something other than 200 or 404.

There are many different HTTP status codes, and checking for every code is not feasible. Most of the time, you should only be concerned with whether your request is successful. Therefore, you can cut the previous code down to this:

```
var oHttp = createXmlHttpRequest();
oHttp.open("GET", "http://localhost/myTextFile.txt", false);
oHttp.send(null);

if (oHttp.status == 200)
{
    alert("The text file was found!");
}
else
{
    alert("An error occurred while attempting to retrieve the file!");
}
```

This code performs the same basic function, but you check only for a status code of 200 and alert a generic message to the user if an error occurred.

Asynchronous Requests

The previous code samples have demonstrated synchronous requests, and the code is simple. Asynchronous requests, on the other hand, are a little more complex and require more code.

The key to asynchronous requests is the `onreadystatechange` event handler. In asynchronous requests, the `XMLHttpRequest` object exposes a `readyState` property, which holds a numeric value; each value refers to a specific ready state of the request.

- ☐ 0 — The object has been created, but the `open()` method hasn't been called
- ☐ 1 — The `open()` method has been called, but the request hasn't been sent
- ☐ 2 — The request has been sent
- ☐ 3 — A response has been received from the server
- ☐ 4 — The requested data have been received

Every time the `readyState` changes, the `readystatechange` event fires, calling the `onreadystatechange` event handler. Most of the time, you should be interested in only one state, 4, as this lets you know that the request is complete.

It is important to note that even if the request was successful, you may not have the information you wanted. An error may have occurred on the server's end of the request (a 404, 500, or some other error). Therefore, you still need to check the status code of the request.

Chapter 16: Ajax and Remote Scripting

Code to handle the `readystatechange` event could look like this:

```
var oHttp = createXmlHttpRequest();

function oHttp_readyStateChange()
{
    if (oHttp.readyState == 4)
    {
        if (oHttp.status == 200)
        {
            alert(oHttp.responseText);
        }
        else
        {
            alert("An error occurred while retrieving the text file!");
        }
    }
}

//more code here
```

This code first defines the `oHttp_readyStateChange()` function, which is assigned to the `onreadystatechange` event handler.

First check to see if the `readyState` is 4, or complete. If so, you know that you can use the received data. Next, check the `status` property to make sure that everything's okay, and then use the `responseText` property, which contains the requested file in plain text format, and alert it to the user.

To use the `oHttp_readyStateChange()` function to handle the `readystatechange` event, assign it to the `onreadystatechange` event handler.

```
var oHttp = createXmlHttpRequest();

function oHttp_readyStateChange()
{
    if (oHttp.readyState == 4)
    {
        if (oHttp.status == 200)
        {
            alert(oHttp.responseText);
        }
        else
        {
            alert("An error occurred while retrieving the text file!");
        }
    }
}

oHttp.open("GET", "http://localhost/myTextFile.txt", true);
oHttp.onreadystatechange = oHttp_readyStateChange;
oHttp.send(null);
```

In this new code, first call the `open()` method, set the request type to `GET`, and specify the desired URL to send the request to. The call to `open()` differs from earlier; the final argument is set to `true`, specifying

that you want to make an asynchronous request. Before calling `send()`, assign the `onreadystatechange` event handler to the `oHttpRequest.onreadystatechange()` function you previously wrote.

Asynchronous requests certainly do require more code; however, the benefits of using asynchronous communication are well worth the effort, as your other JavaScript code continues to run while the request is made. Perhaps a user-defined class that wraps an `XMLHttpRequest` object could make asynchronous requests easier to use and manage.

An `XMLHttpRequest` object also has a property called `responseXML`, which attempts to load the received data into an XML DOM (whereas `responseText` just returns plain text). This is the only way Safari 2 can load XML data into a DOM.

Creating a Remote Scripting Class

Code reuse is an important idea in programming; it is why we define functions to perform specific, common tasks. In Chapter 4, you learned that object-based and object-oriented languages have a different construct for code reuse: a class. A class contains properties that contain data, and/or methods that perform actions with those data.

In this section, you're going to wrap an `XMLHttpRequest` object into your own class, called `HttpRequest`, thereby making it easier for you to make asynchronous requests. Before getting into writing the class, we need to discuss a few properties and methods.

In the `HttpRequest` class, you need to keep track of only one piece of information: the underlying `XMLHttpRequest` object. Therefore, this class will have only one property:

- ❑ `request` — Contains the underlying `XMLHttpRequest` object

The methods for this class are equally easy to identify.

- ❑ `createXmlHttpRequest()` — Creates the `XMLHttpRequest` object in a cross-browser fashion. It is essentially a copy of the function of the same name written earlier in the chapter.
- ❑ `send()` — Sends the request to the server.

With the properties and methods identified, you can begin to write the class.

The `HttpRequest` Constructor

A class's constructor defines its properties and performs any logic needed in order for the class to function properly.

```
function HttpRequest(sUrl, fpCallback)
{
    this.request = this.createXmlHttpRequest();

    //more code here
}
```

Chapter 16: Ajax and Remote Scripting

The `HttpRequest` constructor accepts two arguments. The first, `sUrl`, is a string containing the URL the request should be sent to. The second, `fpCallback`, is a callback function. It will be called when the server's response is received (when the request's `readyState` is 4 and its `status` is 200). The first line of the constructor initializes the `request` property, assigning an `XMLHttpRequest` object to it.

With the `request` property created and ready to use, it's time to prepare the request for sending.

```
function HttpRequest(sUrl, fpCallback)
{
    this.request = this.createXmlHttpRequest();
    this.request.open("GET", sUrl, true);

    function request_readystatechange()
    {
        //more code here
    }

    this.request.onreadystatechange = request_readystatechange;
}
```

The first line of the new code uses the `XMLHttpRequest` object's `open()` method to initialize the request object. Set the request type to `GET`, use the `sUrl` parameter to specify the URL you want to request, and set the request object to use asynchronous mode. The next few lines define the `request_readystatechange()` function. Defining a function within a function may seem weird, but it is legal to do so. This function is called a *closure*. The `request_readystatechange()` function cannot be accessed outside the constructor. It, however, has access to the variables and parameters of the `HttpRequest` constructor. This function handles the request object's `readystatechange` event, and you bind it to do so by assigning it to the `onreadystatechange` event handler.

```
function HttpRequest(sUrl, fpCallback)
{
    this.request = this.createXmlHttpRequest();
    this.request.open("GET", sUrl, true);

    var tempRequest = this.request;
    function request_readystatechange()
    {
        if (tempRequest.readyState == 4)
        {
            if (tempRequest.status == 200)
            {
                fpCallback(tempRequest.responseText);
            }
            else
            {
                alert("An error occurred trying to contact the server.");
            }
        }
    }

    this.request.onreadystatechange = request_readystatechange;
}
```

These new lines of code may look strange. First is the `tempRequest` variable and its value. This variable is a pointer to the current object's `request` property, and it lets you get around scoping issues. Ideally, you would use `this.request` inside the `request_readystatechange()` function. However, in Firefox, this points to the `request_readystatechange()` function instead of to the `XMLHttpRequest` object, which would cause the code to not function properly. So when you see `tempRequest`, think `this.request`.

Inside the `request_readystatechange()` function, you see the following line:

```
fpCallback(tempRequest.responseText);
```

This line calls the callback function specified by the constructor's `fpCallback` parameter, and you pass the `responseText` property to this function. This will allow the callback function to use the information received from the server.

Creating the Methods

There are two methods in this class: one is used inside the constructor, and the other enables you to send the request to the server.

Cross-Browser XMLHttpRequest Creation...Again

The first method is `createXmlHttpRequest()`. The inner workings of cross-browser object creation were covered earlier in the chapter, so let's just see the method definition.

```
HttpRequest.prototype.createXmlHttpRequest = function ()
{
    if (window.XMLHttpRequest)
    {
        var oHttp = new XMLHttpRequest();
        return oHttp;
    }
    else if (window.ActiveXObject)
    {
        var versions =
        [
            "MSXML2.XmlHttp.6.0",
            "MSXML2.XmlHttp.3.0"
        ];

        for (var i = 0; i < versions.length; i++)
        {
            try
            {
                var oHttp = new ActiveXObject(versions[i]);
                return oHttp;
            }
            catch (error)
            {
                //do nothing here
            }
        }
    }
}
```

```
    }  
  
    alert("Your browser doesn't support XMLHttpRequest");  
}
```

In Chapter 4, you learned that class methods are assigned through the `prototype` object. This code follows that rule when writing the `createXmlHttpRequest()` method and the next method.

Sending the Request

Sending a request to the server involves the `XMLHttpRequest` object's `send()` method. This `send()` is similar, with the difference being that it doesn't accept arguments.

```
HttpRequest.prototype.send = function ()  
{  
    this.request.send(null);  
}
```

This version of `send()` is simple in that all you do is call the request object's `send()` method and pass it `null`.

The Full Code

Now that code's been covered, open up your text editor and type the following:

```
function HttpRequest(sUrl, fpCallback)  
{  
    this.request = this.createXmlHttpRequest();  
    this.request.open("GET", sUrl, true);  
  
    var tempRequest = this.request;  
    function request_readystatechange()  
    {  
        if (tempRequest.readyState == 4)  
        {  
            if (tempRequest.status == 200)  
            {  
                fpCallback(tempRequest.responseText);  
            }  
            else  
            {  
                alert("An error occurred trying to contact the server.");  
            }  
        }  
    }  
  
    this.request.onreadystatechange = request_readystatechange;  
}  
  
HttpRequest.prototype.createXmlHttpRequest = function ()  
{  
    if (window.XMLHttpRequest)  
    {
```

```

        var oHttp = new XMLHttpRequest();
        return oHttp;

    }
    else if (window.ActiveXObject)
    {
        var versions =
        [
            "MSXML2.XmlHttp.6.0",
            "MSXML2.XmlHttp.3.0"
        ];

        for (var i = 0; i < versions.length; i++)
        {
            try
            {
                var oHttp = new ActiveXObject(versions[i]);
                return oHttp;
            }
            catch (error)
            {
                //do nothing here
            }
        }

        return null;
    }

    HttpRequest.prototype.send = function ()
    {
        this.request.send(null);
    }

```

Save this file as `httprequest.js`. You'll use it later in the chapter.

The goal of this class was to make asynchronous requests easier to use, so let's look at a brief code-only example and see if that goal was accomplished.

The first thing you need is a function to handle the data received from the request. You'll pass this function to the `HttpRequest` constructor.

```

function handleData(sResponseText)
{
    alert(sResponseText);
}

```

Here is the definition of a function called `handleData()`: it accepts the value of `responseText` (of the `XMLHttpRequest` object) as its only argument. In this example, it merely alerts the data passed to it. Now create an `HttpRequest` object and send the request.

```

var request = new HttpRequest("http://localhost/myTextFile.txt", handleData);
request.send();

```

Chapter 16: Ajax and Remote Scripting

Pass the text file's location and the `handleData()` function to the constructor. Then send the request with the `send()` method. If the request was successful, `handleData()` is called.

This looks much easier to use (and reuse) than `XMLHttpRequest` code that you have to repeatedly rewrite. You don't have to worry about the `readyState` and `status` properties; the `HttpRequest` class does it all.

Creating a Smarter Form with XMLHttpRequest

You've probably seen it many times: registering as a user on a web site's forum or signing up for web-based e-mail, only to find that your desired user name is taken. Of course, you don't find this out until after you've filled out the entire form, submitted it, and watched the page reload with new data (not to mention that you've lost some of the data you entered). Thankfully, Ajax can soften this frustrating experience and allow the user to know if a user name is taken well in advance of the form's submission.

You can approach this solution in a variety of ways: The easiest to implement provides a link that initiates an HTTP request to the server application to check whether the user's desired information is available to use.

The form you'll build will resemble typical forms used today; it will contain the following fields:

- ☐ Username (validated)—The field where the user types her desired user name
- ☐ Email (validated)—The field where the user types her e-mail
- ☐ Password (not validated)—The field where the user types her password
- ☐ Verify Password (not validated)—The field where the user verifies her password

Note that the `Password` and `Verify Password` fields are just for show in this example. Verifying a typed password is certainly something the server application can do; however, it is far more efficient to let JavaScript perform that verification.

Next to the `Username` and `Email` fields will be a hyperlink that calls a JavaScript function to query the server with the `HttpRequest` class you built earlier in this chapter. The server application is a simple PHP file. PHP programming is beyond the scope of this book. However, we should discuss how to request data from the PHP application, as well as look at the response the application sends back to JavaScript.

Requesting Information

The PHP application looks for one of two arguments in the query string: `username` and `email`.

To check the availability of a user name, use the `username` argument. The URL to do this looks like the following:

```
http://localhost/formvalidator.php?username=[usernameToSearchFor]
```

When searching for a user name, replace [usernameToSearchFor] with the actual name.

Searching for an e-mail follows the same pattern. The e-mail URL looks like this:

```
http://localhost/formvalidator.php?email=[emailToSearchFor]
```

The Received Data

A successful request will result in one of two values:

- ☐ available—Means that the user name and/or e-mail is available for use
- ☐ not available—Signifies that the user name and/or e-mail is in use and therefore not available

These values are sent to the client in plain text format. A simple comparison will enable you to tell the user whether her user name or e-mail is already in use.

Before You Begin

This is a live-code Ajax example; therefore, your computer must meet a few requirements if you wish to run this example.

A Web Server

First, you need a web server. If you are using Windows 2000 (Server or Professional), Windows XP Professional, or Windows Server 2003, you have Microsoft's web server software, Internet Information Services, freely available to you. To install it, open Add/Remove Programs in the Control Panel and click Add/Remove Windows Components. Figure 16-3 shows the Windows Component Wizard in Windows XP Professional.

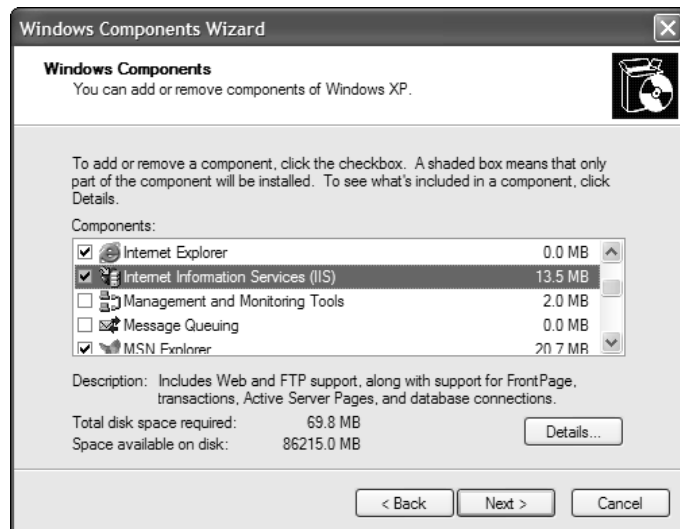


Figure 16-3

Chapter 16: Ajax and Remote Scripting

Check the box next to IIS and click Next to install. You may need your operating system's installation CD to complete the installation.

If your operating system isn't in the preceding list, or you wish to use another method, you can install Apache HTTP Server (www.apache.org). This is an open-source web server and can run on a variety of operating systems, such as Linux, Unix, and Windows, to list only a few.

PHP

PHP is a popular open source server-side scripting language and must be installed on your computer if you wish to run PHP scripts. You can download PHP in a variety of forms (binaries, Windows installation wizards, and source code) at www.php.net. The PHP code used in this example was written in PHP 4, but it should work just fine in version 5.

Try It Out XMLHttpRequest Smart Form

Open your text editor and type the following:

```
<html>
<head>
  <title>Form Field Validation</title>
  <style type="text/css">
    .fieldname
    {
      text-align: right;
    }

    .submit
    {
      text-align: right;
    }
  </style>
  <script type="text/javascript" src="HttpRequest.js"></script>
  <script type="text/javascript">
    function checkUsername()
    {
      var userValue = document.getElementById("username").value;

      if (userValue == "")
      {
        alert("Please enter a user name to check!");
        return;
      }

      var url = "formvalidator.php?username=" + userValue;

      var request = new XMLHttpRequest(url, checkUsername_callBack);
      request.send();
    }

    function checkUsername_callBack(sResponseText)
    {
```



```

        var userValue = document.getElementById("username").value;

        if (sResponseText == "available")
        {
            alert("The username " + userValue + " is available!");
        }
        else
        {
            alert("We're sorry, but " + userValue + " is not available.");
        }
    }

    function checkEmail()
    {
        var emailValue = document.getElementById("email").value;

        if (emailValue == "")
        {
            alert("Please enter an email address to check!");
            return;
        }

        var url = "formvalidator.php?email=" + emailValue;

        var request = new HttpRequest(url, checkEmail_callBack);
        request.send();
    }

    function checkEmail_callBack(sResponseText)
    {
        var emailValue = document.getElementById("email").value;

        if (sResponseText == "available")
        {
            alert("The email " + emailValue + " is currently not in use!");
        }
        else
        {
            alert("I'm sorry, but " + emailValue + " is in use by another
user.");
        }
    }
</script>
</head>
<body>
    <form>
        <table>
            <tr>
                <td class="fieldname">
                    Username:
                </td>
                <td>
                    <input type="text" id="username" />
                </td>
            </tr>
        </table>
    </form>
</body>
</html>

```

```
        <td>
            <a href="javascript: checkUsername()">Check Availability</a>
        </td>
    </tr>
    <tr>
        <td class="fieldname">
            Email:
        </td>
        <td>
            <input type="text" id="email" />
        </td>
        <td>
            <a href="javascript: checkEmail()">Check Availability</a>
        </td>
    </tr>
    <tr>
        <td class="fieldname">
            Password:
        </td>
        <td>
            <input type="text" id="password" />
        </td>
        <td />
    </tr>
    <tr>
        <td class="fieldname">
            Verify Password:
        </td>
        <td>
            <input type="text" id="password2" />
        </td>
        <td />
    </tr>
    <tr>
        <td colspan="2" class="submit">
            <input type="submit" value="Submit" />
        </td>
        <td />
    </tr>
</table>
</form>
</body>
</html>
```

Save this file in your web server's root directory. If you're using IIS for your web server, save it as `c:\inetpub\wwwroot\validate_form.htm`. If you're using Apache, you'll want to save it inside the `htdocs` folder: `pathTohtdocs\htdocs\validate_form.htm`.

You also need to place `httprequest.js` (the `HttpRequest` class) and the `formvalidator.php` file into the same directory as `validate_form.htm`.

Now open your browser and navigate to `http://localhost/formvalidator.php`. If everything is working properly, you should see the text “PHP is working correctly. Congratulations!” as in Figure 16-4.

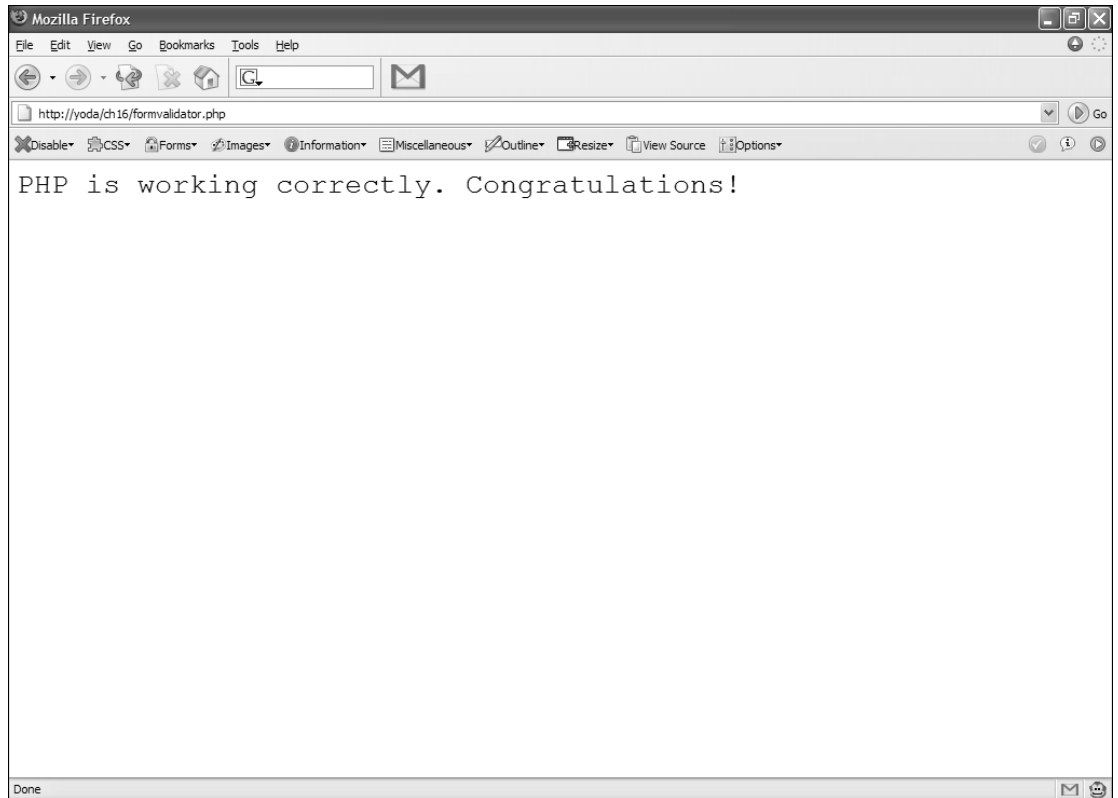


Figure 16-4

Now point your browser to `http://localhost/validate_form.htm`, and you should see something like Figure 16-5.

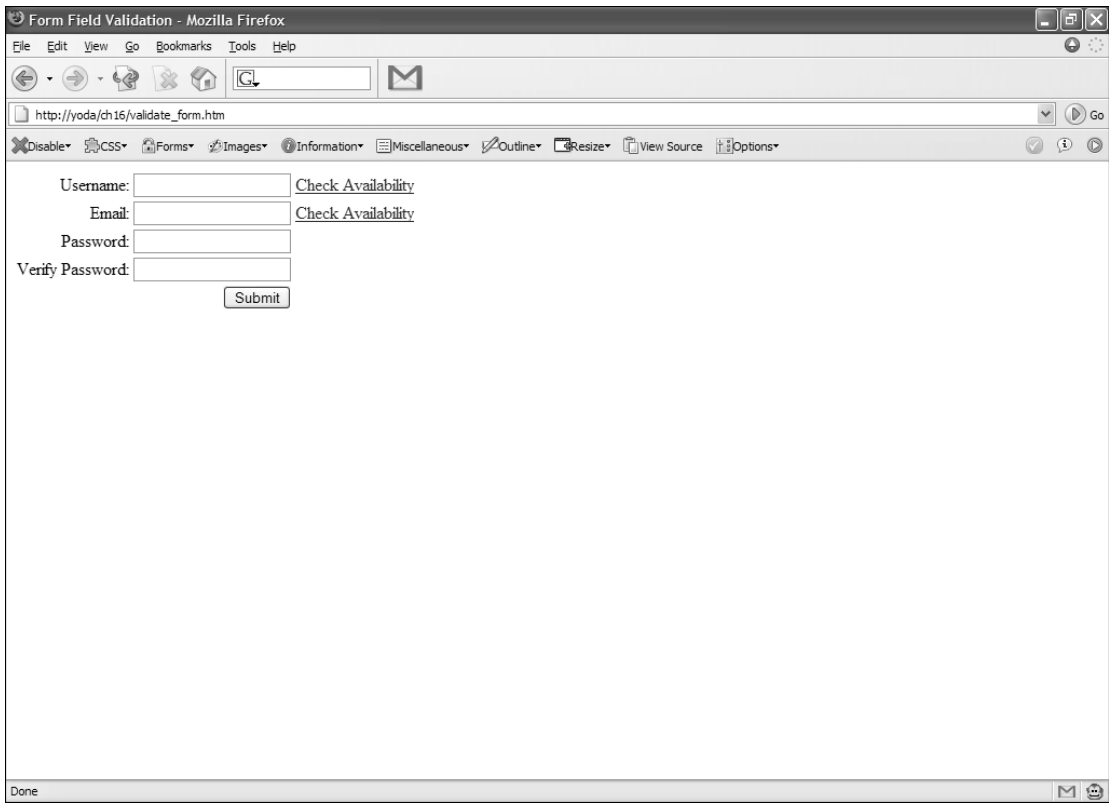


Figure 16-5

Type **jmcppeak** into the Username field and click the Check Availability link next to it. You'll see an alert box like the one in Figure 16-6.



Figure 16-6

Now type **someone@xyz.com** in the Email field and click the Check Availability link next to it. Again, you'll be greeted with an alert box stating that the e-mail's already in use. Now input your own user name and e-mail into these fields and click the appropriate links. Chances are an alert box will tell you that your user name and/or e-mail is available (the user names `jmcppeak` and `pwilton` and the e-mails `someone@xyz.com` and `someone@zyx.com` are the only ones used by the application).

How It Works

The body of this HTML page is a simple form whose fields are contained within a table. Each form field exists in its own row in the table. The first two rows contain the fields we're most interested in, the Username and Email fields.

```
<form>
  <table>
    <tr>
      <td class="fieldname">
        Username:
      </td>
      <td>
        <input type="text" id="username" />
      </td>
      <td>
        <a href="javascript: checkUsername()">Check Availability</a>
      </td>
    </tr>
    <tr>
      <td class="fieldname">
        Email:
      </td>
      <td>
        <input type="text" id="email" />
      </td>
      <td>
        <a href="javascript: checkEmail()">Check Availability</a>
      </td>
    </tr>
  </table>
<!-- HTML to be continued later -->
```

The first column contains text identifiers for the fields. The second column contains the `<input/>` elements themselves. Each of these tags has an `id` attribute, `username` for the Username field and `email` for the Email field. This enables you to easily find the `<input/>` elements and get the text entered into them. The third column contains an `<a/>` element. The hyperlinks use the `javascript:` protocol to call JavaScript code. In this case, the `checkUsername()` and `checkEmail()` functions are called when the user clicks the links. We'll examine these functions in a few moments.

The remaining three rows in the table contain two password fields and the Submit button (the smart form currently does not use these fields). The next two rows also contain three columns: the first is the text describing the field's function, the second contains the `<input/>` tag, and the third is empty.

```
<!-- HTML continued from earlier -->
<tr>
  <td class="fieldname">
    Password:
  </td>
  <td>
    <input type="text" id="password" />
  </td>
  <td />
</tr>
```

Chapter 16: Ajax and Remote Scripting

```
<tr>
  <td class="fieldname">
    Verify Password:
  </td>
  <td>
    <input type="text" id="password2" />
  </td>
</tr>
<tr>
  <td colspan="2" class="submit">
    <input type="submit" value="Submit" />
  </td>
</tr>
</table>
</form>
```

The final row contains two cells. The first cell spans two columns and contains the Submit button. The second cell is empty.

The CSS in this HTML page consists of only a couple of rules.

```
.fieldname
{
    text-align: right;
}

.submit
{
    text-align: right;
}
```

The style declarations used are to align the fields to give the form a clean and aligned look.

As stated earlier, the hyperlinks are key to the Ajax functionality, as they call JavaScript functions when clicked. The first function to discuss, `checkUsername()`, is responsible for retrieving the text the user entered into the Username field, and performing an HTTP request using that information.

```
function checkUsername()
{
    var userValue = document.getElementById("username").value;

    if (userValue == "")
    {
        alert("Please enter a user name to check!");
        return;
    }

    var url = "formvalidator.php?username=" + userValue;

    var request = new HttpRequest(url, checkUsername_callBack);
    request.send();
}
```

To retrieve the user name data, use the `document.getElementById()` method to retrieve the `<input/>` element and use the `value` property to retrieve the text typed into the text box. Next, check to see if the user typed any text by comparing the `userValue` variable to an empty string (`" "`). If the text box is empty, the function alerts the user to input a user name and stops the function from processing further. If we didn't do this, the application would make unnecessary requests to the server.

Next, construct the URL to make the request to the PHP application. The final steps in this function are to create an `HttpRequest` object, pass the URL and the callback function, and send the request.

The `checkUsername_callBack()` function executes when the `HttpRequest` object receives a complete response from the server. This function uses the requested information to tell the user whether the user name is available. Remember, there are two possible values sent from the server, `available` and `not available`; therefore, you need only to check for one of these values.

```
function checkUsername_callBack(sResponseText)
{
    var userValue = document.getElementById("username").value;

    if (sResponseText == "available")
    {
        alert("The username " + userValue + " is available!");
    }
    else
    {
        alert("We're sorry, but " + userValue + " is not available.");
    }
}
```

If the server's response is `available`, the function tells the user that his desired user name is okay to use. If not, the alert box says that his user name is taken.

Checking the e-mail's availability is a similar process. The `checkEmail()` function's purpose is to retrieve the text typed in the `Email` field, and to pass that information to the server application.

```
function checkEmail()
{
    var emailValue = document.getElementById("email").value;

    if (emailValue == "")
    {
        alert("Please enter an email address to check!");
        return;
    }

    var url = "formvalidator.php?email=" + emailValue;

    var request = new HttpRequest(url, checkEmail_callBack);
    request.send();
}
```

The `checkEmail_callBack()` function is similar to `checkUsername_callBack()`. The function uses the same logic, but it is based on the `Email` field's value.

```
function checkEmail_callBack(sResponseText)
{
    var emailValue = document.getElementById("email").value;

    if (sResponseText == "available")
    {
        alert("The email " + emailValue + " is currently not in use!");
    }
    else
    {
        alert("I'm sorry, but " + emailValue + " is in use by another user.");
    }
}
```

Once again, the function checks to see if the server's response is `available`, and if so, to let the user know that the e-mail address is currently not being used. If the address is not available, a different message tells the user his e-mail is not available.

Using `XMLHttpRequest` isn't the only way to tackle this form, and we'll look at implementing this a different way in a little bit.

Creating a Smarter Form with an IFrame

One of the advantages of `XMLHttpRequest` is its ease of use. You simply create the object, send the request, and await the server's response. Unfortunately, the JavaScript object does have a downside: The browser does not log a history of requests made with the object. Therefore, `XMLHttpRequest` essentially breaks the browser's Back button.

The solution to this problem lies in an older Ajax technique: using hidden frames/iframes to facilitate client-server communication. You must use two frames in order for this method to work properly. One must be hidden, and one must be visible.

Note that when you are using an iframe, the document that contains the iframe is the visible frame.

The hidden-frame technique consists of a four-step process. The first step is taken when the user, knowingly or unknowingly, initiates a JavaScript call to the hidden frame. This can be done by the user clicking a link in the visible frame, or some other form of user interaction. This call is usually nothing more complicated than the redirection of the frame to a different web page. This redirection automatically triggers the second step: the request is sent to the server.

After the server finishes processing the request, the third step in the process happens: The server sends the response to the hidden frame. The server's response is a web page, as this response is sent to a hidden frame. When the response is completely received, the web page in the hidden frame must contact the visible frame, telling it that the response is complete. This is the fourth step, and it is sometimes accomplished with the `window.onload` event handler of the hidden frame.

The example in this section is based upon the one built in the previous section. But this time, you'll use a hidden iframe to facilitate the communication between the browser and the server. Before getting into the code, we should first talk about the data received from the server.

The following example does not work in Safari, as it does not log the history of an iframe.

The Server Response

When we used `XMLHttpRequest` to get data from the server, we expected only a few words as the server's response. With this different approach, we know our response must consist of two things:

- ❑ The data, which must be in HTML format
- ❑ A mechanism to contact the parent document when the iframe receives the HTML response.

Keeping these two things in mind, you can begin to construct the response HTML page.

```
<html>
<head>
  <title>Returned Data</title>
</head>
<body>
  <script type="text/javascript">
    //more code here
  </script>
</body>
</html>
```

This simple HTML page contains a single `<script/>` element in the body of the document. The JavaScript code contained in this script block will be generated by the PHP application, calling either `checkUsername_callBack()` or `checkEmail_callBack()` in the visible frame and passing `available` or `not available` as their arguments. Therefore, the following HTML document is a valid response from the PHP application:

```
<html>
<head>
  <title>Returned Data</title>
</head>
<body>
  <script type="text/javascript">
    top.checkUsername_callBack("available", "some_username");
  </script>
</body>
</html>
```

In this sample response, the tested user name is available. Therefore, the HTML page calls the `checkUsername_callBack()` function in the parent window and passes the string `available`. Also, the searched user name (or e-mail) is sent back to the client. Do this, because then the client application will display the correct user name or e-mail when the Back or Forward button is pressed. With the response in this format, you can keep a good portion of our JavaScript code the same.

Try It Out Iframe Smart Form

The code for this revised smart form is very similar to the code you used previously with the XMLHttpRequest example. There are, however, a few slight changes. Open up your text editor and type the following:

```
<html>
<head>
  <title>Form Field Validation</title>
  <style type="text/css">
    .fieldname
    {
      text-align: right;
    }

    .submit
    {
      text-align: right;
    }

    #hiddenFrame
    {
      display: none;
    }
  </style>
  <script type="text/javascript">
    function checkUsername()
    {
      var userValue = document.getElementById("username").value;

      if (userValue == "")
      {
        alert("Please enter a user name to check!");
        return;
      }

      var url = "iframe_formvalidator.php?username=" + userValue;

      frames["hiddenFrame"].location = url;
    }

    function checkUsername_callBack(data, userValue)
    {
      if (data == "available")
      {
        alert("The username " + userValue + " is available!");
      }
      else
      {
        alert("We're sorry, but " + userValue + " is not available.");
      }
    }

    function checkEmail()
```

```

    {
        var emailValue = document.getElementById("email").value;

        if (emailValue == "")
        {
            alert("Please enter an email address to check!");
            return;
        }

        var url = "iframe_formvalidator.php?email=" + emailValue;

        frames["hiddenFrame"].location = url;
    }

function checkEmail_callBack(data, emailValue)
{
    if (data == "available")
    {
        alert("The email " + emailValue + " is currently not in use!");
    }
    else
    {
        alert("We're sorry, but " + emailValue
            + " is in use by another user.");
    }
}
</script>
</head>
<body>
    <form>
        <table>
            <tr>
                <td class="fieldname">
                    Username:
                </td>
                <td>
                    <input type="text" id="username" />
                </td>
                <td>
                    <a href="javascript: checkUsername()">Check Availability</a>
                </td>
            </tr>
            <tr>
                <td class="fieldname">
                    Email:
                </td>
                <td>
                    <input type="text" id="email" />
                </td>
                <td>
                    <a href="javascript: checkEmail()">Check Availability</a>
                </td>
            </tr>
            <tr>
                <td class="fieldname">

```

```
        Password:
    </td>
    <td>
        <input type="text" id="password" />
    </td>
    <td />
</tr>
<tr>
    <td class="fieldname">
        Verify Password:
    </td>
    <td>
        <input type="text" id="password2" />
    </td>
    <td />
</tr>
<tr>
    <td colspan="2" class="submit">
        <input type="submit" value="Submit" />
    </td>
    <td />
</tr>
</table>
</form>
<iframe src="about:blank" id="hiddenFrame" name="hiddenFrame" />
</body>
</html>
```

Save this file as `validate_iframe_form.htm`, and save it in your web server's root directory. Also locate the `iframe_formvalidator.php` file from the code download and place it in the same directory.

Open your web browser (not Safari) and navigate to `http://localhost/validate_iframe_form.htm`. You should see something like what is shown in Figure 16-7.

Check for three user names and e-mail addresses. After you clear the final alert box, press the browser's Back button a few times. You'll notice that it is cycling through the information you entered. The text in the text box will not change; however, the alert box will display the names and e-mails you entered. You can do the same thing with the Forward button.

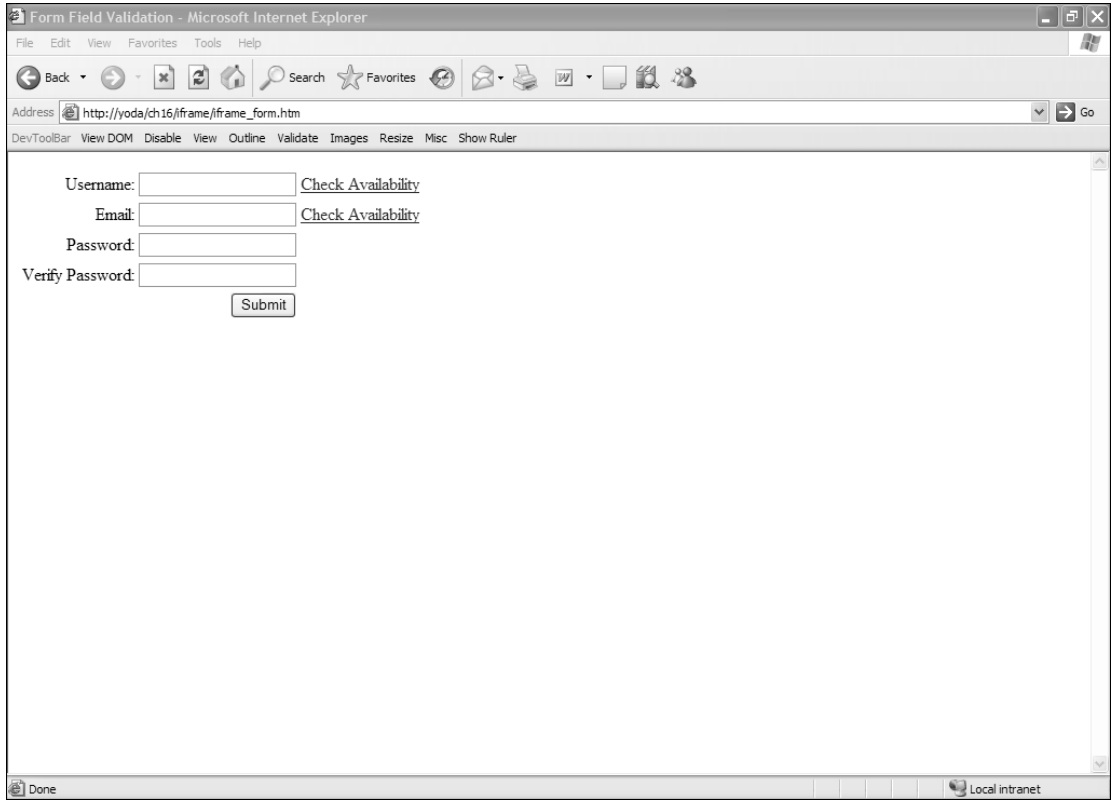


Figure 16-7

How It Works

The HTML in the body of the page remains unchanged except for the addition of the `<iframe/>` tag after the closing `<form/>` tag.

```
<iframe src="about:blank" id="hiddenFrame" name="hiddenFrame" />
```

This frame is initialized to have a blank HTML page loaded. Its `name` and `id` attributes contain the value of `hiddenFrame`. We'll use the value of the `name` attribute later to retrieve this frame from the `frames` collection in the BOM. Next, we set the CSS for the frame.

```
#hiddenFrame
{
    display: none;
}
```

Chapter 16: Ajax and Remote Scripting

This rule contains one style declaration to hide the iframe from view.

Hiding an iframe through CSS enables you to easily show it if you need to debug the server-side application.

Next up, the JavaScript.

```
function checkUsername()
{
    var userValue = document.getElementById("username").value;

    if (userValue == "")
    {
        alert("Please enter a user name to check!");
        return;
    }

    var url = "iframe_formvalidator.php?username=" + userValue;

    frames["hiddenFrame"].location = url;
}
```

The `checkUsername()` function has undergone a small change: It makes a request via the iframe instead of using `XMLHttpRequest`. It starts by retrieving the value of the Username text box. It then checks to see if the user typed anything into the box; if not, an alert box displays a message to the user telling her to enter a user name. If the value isn't an empty string, then the function continues and constructs the URL to send the request to. The final step is to load the URL into the hidden iframe by using the `frames` collection and the `location` property.

The second function, `checkUsername_callBack()`, has also slightly changed. It now accepts two arguments: the first will contain either `available` or `not available`, and the second will contain the user name sent in the request.

```
function checkUsername_callBack(data, userValue)
{
    if (data == "available")
    {
        alert("The username " + userValue + " is available!");
    }
    else
    {
        alert("We're sorry, but " + userValue + " is not available.");
    }
}
```

The function first checks to see if the user name is available. If so, an alert box tells the user that the user name is available. If not, the user sees an alert box stating that the user name is not available.

The functions for searching e-mail addresses are very similar to those for searching user names.

```
function checkEmail()
{
    var emailValue = document.getElementById("email").value;

    if (emailValue == "")
    {
        alert("Please enter an email address to check!");
        return;
    }

    var url = "iframe_formvalidator.php?email=" + emailValue;

    frames["hiddenFrame"].location = url;
}

function checkEmail_callBack(data, emailValue)
{
    if (data == "available")
    {
        alert("The email " + emailValue + " is currently not in use!");
    }
    else
    {
        alert("We're sorry, but " + emailValue + " is in use by another user.");
    }
}
```

The `checkEmail()` process follows the same pattern as `checkUsername()`. The function retrieves the text box's value, checks to see if the user entered data, constructs the URL, and loads the URL into the iframe.

The `checkEmail_callBack()` function contains changes similar to those made to `checkUsername_callBack()`. The function now accepts two arguments, checks to see if the e-mail is available, and displays a message accordingly.

These two examples are simple, and they were designed as such to focus on the Ajax techniques used to make them work. There are other approaches to smart forms (ones that do not include clicking a link), and while some are just as easy to implement as the ones in this chapter, others are not.

Also, keep in mind what Ajax really is: client-server communication that does not cause the page to refresh. The two key words in that definition are *client* and *server*. As these two examples have shown, a server component is required to handle data from and send data to the client. You'll need to pick up a server-side technology to perform this task in order to use Ajax to its fullest. Any server-side technology will do: PHP, ASP (classic or .NET), ColdFusion, PERL — anything that can output data in a text format.

Things to Watch Out For

Using JavaScript to communicate between server and client adds tremendous power to the language's abilities. However, this power does not come without its share of caveats.

The first two items discussed are more “for your information” than anything else. The third mini-section covers the usability issues: how Ajax breaks some aspects of web pages and how to fix them.

The Same-Origin Policy

Since the early days of Netscape Navigator 2.0, JavaScript cannot access scripts or documents from a different origin. This is a security measure that browser makers adhere to; otherwise, malicious coders could execute code wherever they wanted. The same-origin policy dictates that two pages are of the same origin only if the protocol (HTTP), port (the default is 80), and host are the same.

Consider the following two pages:

- ❑ Page 1 is located at `http://www.site.com/folder/mypage1.htm`
- ❑ Page 2 is located at `http://www.site.com/folder10/mypage2.htm`

According to the same-origin policy, these two pages are of the same origin. They share the same host (`www.site.com`), use the same protocol (HTTP), and are accessed on the same port (none is specified; therefore, they both use 80). Since they are of the same origin, JavaScript on one page can access the other page.

Now consider the next two pages:

- ❑ Page 1 is located at `http://www.site.com/folder/mypage1.htm`
- ❑ Page 2 is located at `https://www.site.com/folder/mypage2.htm`

These two pages are not of the same origin. The host is the same, and they use the same port. However, their protocols are different. Page 1 uses HTTP while Page 2 uses HTTPS. This difference, while slight, is enough to give the two pages two separate origins. Therefore, JavaScript on one of these pages cannot access the other page.

So what does this have to do with Ajax? Everything, because a large part of Ajax is JavaScript. For example, because of this policy, an `XMLHttpRequest` object cannot retrieve any file or document from a different origin. You can easily overcome this hurdle by using the server in the page's origin as a proxy to retrieve data from servers of a different origin. This policy also affects the hidden frame/iframe technique. JavaScript cannot interact with two pages of different origins, even if they are in the same frameset.

ActiveX

One of the downsides of `XMLHttpRequest` is in ActiveX, and only affects Internet Explorer on Windows; however, IE currently has the highest market share of all browsers, and it seems that isn't going to change anytime soon. Over the past few years, more security concerns have been raised with ActiveX, especially since many adware and spyware companies have used the technology to install their wares onto trusting user's computers.

Because of this rise in the awareness of security concerns, Microsoft (and users) is taking steps to make the browser more secure from hijacking attempts by restricting access to ActiveX plug-ins and objects. If a user turns off ActiveX completely, or your site is flagged for a certain security zone, ActiveX objects cannot be created, rendering your XMLHttpRequest-based Ajax applications dead in the water.

Usability Concerns

Ajax breaks the mold of traditional web applications and pages. It enables us to build applications that behave in a more conventional, non-“webbish” way. This, however, is also a drawback, as the Internet has been around for many, many years, and users are accustomed to traditional web pages.

Therefore, it is up to us to ensure that the user can use our web pages, and use them as they expect to, without causing frustration.

The Browser’s Back Button

The browser’s Back and Forward buttons are fundamental to how a user uses the Web. Earlier in the chapter we discussed how XMLHttpRequest breaks these buttons, and we built an Ajax-enabled form that uses a hidden iframe to ensure that the Back and Forward buttons’ functionality were preserved.

There are, however, a few issues with this approach. In Internet Explorer, using the hidden frame/iframe technique works with little problem. The browser logs each request made to the server, so you can go back and forward with no problem. The other browsers, however, do have a few quirks worth mentioning.

Firefox, for example, also logs each request, as the example showed. But we added the iframe to the HTML page with the <iframe> tag. In some early versions of Firefox, if you dynamically add an <iframe/> element to an HTML page by using DOM methods, Firefox will not log the history of the iframe.

Safari is another browser we must watch for; it doesn’t store the history of an iframe. A traditional frameset must be used for Safari to log the requests in the history. The hidden frame technique is almost exactly the same as the iframe technique; the only difference is that you use a frameset instead of an iframe.

Dealing with Delays

The web browser is just like any other conventional application in that user interface (UI) cues tell the user that something is going on. When a user clicks a link, the throbber animation runs, an hourglass appears next to the cursor (in Windows), and a status bar usually shows the browser’s progress in loading the page.

This is another area in which Ajax solutions, and XMLHttpRequest specifically, miss the mark. This problem, however, is simple to overcome: Simply add UI elements to tell the user something is going on and remove them when the action is completed. Consider the following code:

```
function requestComplete(sResponseText)
{
    //do something with the data here

    document.getElementById("divLoading").style.display = "none";
```

```
}  
  
var myRequest = new XMLHttpRequest("http://localhost/myfile.txt", requestComplete);  
document.getElementById("divLoading").style.display = "block";//show that we're  
loading  
myRequest.send();
```

This code uses the `HttpRequest` class built earlier to request a text file. Before sending the request, retrieve an HTML element in the document with an id of `divLoading`. This `<div/>` element tells the user that data is loading. When the request is completed, hide the loading `<div/>` element, which lets the user know that the loading process is completed.

Offering this information to your users lets them know the application is performing some operation that they requested. Otherwise, they may wonder if the application is working correctly when they click something and see nothing instantly happen.

Degrade Gracefully When Ajax Fails

In a perfect world, the code you write would work every time it runs. Unfortunately, you have to face the fact that many times Ajax-enabled web pages will not use the Ajax-enabled goodness, because a user turned off JavaScript in his browser.

The only real answer to this problem is to build an old-fashioned web page with old-fashioned forms, links, and other HTML elements. Then, using JavaScript, you can disable the default behavior of those HTML elements and add Ajax functionality. As an example, consider this hyperlink:

```
<a href="http://www.wrox.com" title="Wrox Publishing">Wrox Publishing</a>
```

This is a normal, run-of-the-mill hyperlink. When the user clicks it, it will take him to `http://www.wrox.com`. By using JavaScript, you can override this action and replace it with your own.

```
<a href="http://www.wrox.com" title="Wrox Publishing"  
onclick="return false;">Wrox Publishing</a>
```

The key to this functionality is the `onclick` event handler and returning a value of `false`. You can execute any code you wish with the event handler; just remember to `return false` at the end. This tells the browser to not do anything when the link is clicked. If the user's JavaScript is turned off, the `onclick` event handler is ignored, and the link behaves as it normally should.

As a rule of thumb, build your web page first and add Ajax later.

Summary

In this chapter, you were introduced to the concept of Ajax, a fancy-shmancy word for an old technique called remote scripting.

- ❑ We discussed the `XMLHttpRequest`, and how it differed between IE 5 & 6, and IE 7, Firefox, Opera, and Safari. You learned how to make both synchronous and asynchronous requests to the server and how to use the `onreadystatechange` event handler.

- ❑ We built our own Ajax class to make asynchronous HTTP requests easier for us to code.
- ❑ We used our new Ajax class in a smarter form, one that checks user names and e-mails to see if they are already in use.
- ❑ We discussed how `XMLHttpRequest` breaks the browser's Back and Forward buttons, and addressed this problem by rebuilding the same form by using a hidden iframe to make requests.
- ❑ We looked at some of the downsides to Ajax, the security issues and the gotchas.

Exercise Questions

Suggested solutions for these questions can be found in Appendix A.

Question 1

Extend the `HttpRequest` class to include synchronous requests in addition to the asynchronous requests the class already makes. You'll have to make some adjustments to your code to incorporate this functionality.

Hint: Create an `async` property for the class.

Question 2

It was mentioned earlier in the chapter that the smart forms could be modified to not use hyperlinks. Change the form that uses the `HttpRequest` class so that the user name and e-mail fields are checked at form submission. The only time you need to alert the user is when the user name or e-mail is taken.

Hint: Use the new version of `HttpRequest` and its new functionality.

A

Exercise Solutions

In this appendix you'll find some suggested solutions to the exercise questions that appear at the end of most of the chapters in this book.

Chapter 2

In this chapter you looked at how JavaScript stores and manipulates data such as numbers and text.

Question 1

Write a JavaScript program to convert degrees centigrade into degrees Fahrenheit, and to write the result to the page in a descriptive sentence. The JavaScript equation for Fahrenheit to centigrade is as follows:

```
degFahren = 9 / 5 * degCent + 32
```

Solution

```
<html>
<body>

<script language="JavaScript" type="text/javascript">

var degCent = prompt("Enter the degrees in centigrade",0);
var degFahren = 9 / 5 * degCent + 32;

document.write(degCent + " degrees centigrade is " + degFahren +
    " degrees Fahrenheit");

</script>

</body>
</html>
```

Appendix A: Exercise Solutions

Save this as `ch2_q1.htm`.

You get the degrees centigrade the user wants to convert by using the `prompt()` function, and store it inside the `degCent` variable.

You then do your calculation, which uses the data stored in `degCent` and converts them to Fahrenheit. The result is assigned to the `degFahren` variable.

Finally, you write the results to the web page, building it up in a sentence using the concatenation operator `+`. Note how JavaScript knows in the calculation that `degCent` is to be treated as a number, but in the `document.write()` it knows that it should be treated as text for concatenation. So how does it know? Simple, it looks at the context. In the calculation, `degCent` is surrounded by numbers and numerical-only operators, such as `*` and `/`. In the `document.write()`, `degCent` is surrounded by strings, hence JavaScript assumes the `+` means concatenate.

Question 2

The following code uses the `prompt()` function to get two numbers from the user. It then adds those two numbers together and writes the result to the page:

```
<html>
<body>
<script language="JavaScript" type="text/javascript">

var firstNumber = prompt("Enter the first number","");
var secondNumber = prompt("Enter the second number","");
var theTotal = firstNumber + secondNumber;
document.write(firstNumber + " added to " + secondNumber + " equals " +
    theTotal);

</script>
</body>
</html>
```

However, if you try the code out, you'll discover that it doesn't work. Why not?

Change the code so that it does work.

Solution

The data that the `prompt()` actually obtains is a string. So both `firstNumber` and `secondNumber` contain text that happens to be number characters. When you use the `+` symbol to add the two variables together, JavaScript assumes that since it's string data, you must want to concatenate the two together and not sum them.

To make it explicit to JavaScript that you want to add the numbers together, you need to convert the data to numbers using the `parseFloat()` function.

```

<html>
<body>
<script language="JavaScript" type="text/javascript">
var firstNumber = parseFloat(prompt("Enter the first number",""));
var secondNumber = parseFloat(prompt("Enter the second number",""));
var theTotal = firstNumber + secondNumber;
document.write(firstNumber + " added to " + secondNumber + " equals " +
    theTotal);

</script>
</body>
</html>

```

Save this as `ch2_q2.htm`.

Now the data returned by the `prompt()` function are converted to floating-point numbers before being stored in the `firstNumber` and `secondNumber` variables. Then, when we do the addition that is stored in `theTotal`, JavaScript makes the correct assumption that, because both the variables are numbers, we must mean to add them up and not concatenate them.

The general rule is that where we have expressions with only numerical data, the `+` operator means *do addition*. If there are any string data, the `+` will mean concatenate.

Chapter 3

In this chapter you looked at how JavaScript can make decisions based on conditions. You also looked at loops and functions.

Question 1

A junior programmer comes to you with some code that appears not to work. Can you spot where he went wrong? Give him a hand and correct the mistakes.

```

var userAge = prompt("Please enter your age");

if (userAge = 0);
{
    alert("So you're a baby!");
}
else if ( userAge < 0 | userAge > 200)
    alert("I think you may be lying about your age");
else
{
    alert("That's a good age");
}

```

Appendix A: Exercise Solutions

Solution

Oh dear, our junior programmer is having a bad day! There are two mistakes on the following line:

```
if (userAge = 0);
```

First, he has only one equals sign instead of two in the `if`'s condition, which means `userAge` will be assigned the value of 0 rather than `userAge` being *compared* to 0. The second fault is the semicolon at the end of the line — statements such as `if` and loops such as `for` and `while` don't require semicolons. The general rule is that if the statement has an associated block (that is, code in curly braces) then no semicolon is needed. So the line should be as follows:

```
if (userAge == 0)
```

The next fault is with these lines:

```
else if ( userAge < 0 | userAge > 200)
    alert("I think you may be lying about your age");
else
```

The junior programmer's condition is asking if `userAge` is less than 0 OR `userAge` is greater than 200. The correct operator for a Boolean OR is `||`, but the programmer has only used one `|`.

```
else if ( userAge < 0 || userAge > 200)
{
    alert("I think you may be lying about your age");
}
else
```

Question 2

Using `document.write()`, write code that displays the results of the 12 times table. Its output should be the results of the calculations.

```
12 * 1 = 12
12 * 2 = 24
12 * 3 = 36
.....
12 * 11 = 132
12 * 12 = 144
```

Solution

```
<html>
<body>
<script language="JavaScript" type="text/javascript">

var timesTable = 12;
var timesBy;

for (timesBy = 1; timesBy < 13; timesBy++)
```



```

{
    document.write(timesTable + " * " + timesBy + " = " + timesBy * timesTable +
        "<br>");
}
</script>
</body>
</html>

```

Save this as `ch3_q2.htm`.

You use a `for` loop to calculate from $1 * 12$ up to $12 * 12$. The results are written to the page with `document.write()`. What's important to note here is the effect of the order of precedence; the concatenation operator (the `+`) has a lower order of precedence than the multiplication operator, `*`. This means that the `timesBy * timesTable` is done before the concatenation, which is the result you want. If this were not the case, you'd have to put the calculation in parentheses to raise its order of precedence.

Question 3

Change the code of Question 2 so that it's a function that takes as parameters the times table required and the values at which it should start and end. For example, you might try the four times table displayed starting with $4 * 4$ and ending at $4 * 9$.

Solution

```

<html>
<body>
<script language="JavaScript" type="text/javascript">

function writeTimesTable(timesTable, timesByStart, timesByEnd)
{
    for (;timesByStart <= timesByEnd; timesByStart++)
    {
        document.write(timesTable + " * " + timesByStart + " = " +
            timesByStart * timesTable + "<br>");
    }
}

writeTimesTable(4,4,9);
</script>
</body>
</html>

```

Save this as `ch3_q3.htm`.

You've declared your function, calling it `writeTimesTable()`, and given it three parameters. The first is the times table you want to write, the second is the start point, and the third is the number it should go up to.

You've modified your `for` loop. First you don't need to initialize any variables, so the initialization part is left blank—you still need to put a semicolon in, but there's no code before it. The `for` loop continues while the `timesByStart` parameter is less than or equal to the `timesByEnd` parameter. You can see that,

Appendix A: Exercise Solutions

as with a variable, you can modify parameters—in this case, `timesByStart` is incremented by one for each iteration through the loop.

The code to display the times table is much the same. For the function's code to be executed, you now actually need to call it, which you do in the following line:

```
writeTimesTable(4,4,9);
```

This will write the four times table, starting at 4 times 4 and ending at 9 times 4.

Question 4

Modify the code of Question 3 to request the times table to be displayed from the user; the code should continue to request and display times tables until the user enters -1. Additionally, do a check to make sure that the user is entering a valid number; if the number is not valid, ask her to re-enter it.

Solution

```
<html>
<body>
<script language="JavaScript" type="text/javascript">

function writeTimesTable(timesTable, timesByStart, timesByEnd)
{
    for (;timesByStart <= timesByEnd; timesByStart++)
    {
        document.write(timesTable + " * " + timesByStart + " = " +
            timesByStart * timesTable + "<br>");
    }
}

var timesTable;

while ( (timesTable = prompt("Enter the times table",-1)) != -1)
{
    while (isNaN(timesTable) == true)
    {
        timesTable = prompt(timesTable + " is not a valid number, please retry",-1);
    }

    if (timesTable == -1)
    {
        break;
    }

    document.write("<br>The " + timesTable + " times table<br>");
    writeTimesTable(timesTable,1,12);
}
</script>
</body>
</html>
```

Save this as `ch3_q4.htm`.

The function remains the same, so let's look at the new code. The first change from Question 3 is that you declare a variable, `timesTable`, and then initialize it in the condition of the first `while` loop. This may seem like a strange thing to do at first, but it does work. The code in parentheses inside the `while` loop's condition

```
(timesTable = prompt("Enter the times table",-1))
```

is executed first because its order of precedence has been raised by the parentheses. This will return a value, and it is this value that is compared to `-1`. If it's not `-1`, then the `while` condition is `true`, and the body of the loop executes. Otherwise it's skipped over, and nothing else happens in this page.

In a second `while` loop nested inside the first, you check to see that the value the user has entered is actually a number using the function `isNaN()`. If it's not, then you prompt the user to try again, and this will continue until a valid number is entered.

If the user had entered an invalid value initially, then in the second `while` loop she may have entered `-1`, so following the `while` is an `if` statement that checks to see if `-1` has been entered. If it has, you `break` out of the `while` loop; otherwise the `writeTimesTable()` function is called.

Chapter 4

In this chapter you saw that JavaScript is an object-based language and you saw how to use some of the native JavaScript objects, such as the `Date` and `Math` objects.

Question 1

Using the `Date` object, calculate the date 12 months from now and write this into a web page.

Solution

```
<html>
<body>
<script language="JavaScript" type="text/javascript">
var months = new Array("Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug",
                        "Sep", "Oct", "Nov", "Dec");

var nowDate = new Date();

nowDate.setMonth(nowDate.getMonth() + 12);
document.write("Date 12 months ahead is " + nowDate.getDate());
document.write(" " + months[nowDate.getMonth()]);
document.write(" " + nowDate.getFullYear());

</script>
</body>
</html>
```

Appendix A: Exercise Solutions

Save this as `ch04_q1.htm`.

Because the `getMonth()` method returns a number between 0 and 11 for the month rather than its name, an array called `months` has been created that stores the name of each month. You can use `getMonth()` to get the array index for the correct month name.

The variable `nowDate` is initialized to a new `Date` object. Because no initial value is specified, the new `Date` object will contain today's date.

To add 12 months to the current date you simply use `setMonth()`. You get the current month value with `getMonth()`, and then add 12 to it.

Finally you write the result out to the page.

Question 2

Obtain a list of names from the user, storing each name entered in an array. Keep getting another name until the user enters nothing. Sort the names in ascending order, and then write them out to the page, with each name on its own line.

Solution

```
<html>
<body>
<script language="JavaScript" type="text/javascript">

var inputName = "";
var namesArray = new Array();

while ( (inputName = prompt("Enter a name", "")) != " " )
{
    namesArray[namesArray.length] = inputName;
}

namesArray.sort();

var namesList = namesArray.join("<br>");
document.write(namesList);

</script>
</body>
</html>
```

Save this as `ch04_q2.htm`.

First you declare two variables: `inputName`, which will hold the name entered by the user, and `namesArray`, which holds an `Array` object that stores each of the names entered.

You use a `while` loop to keep getting another name from the user as long as the user hasn't left the prompt box blank. Note that the use of parentheses in the `while` condition is essential. By placing the

following code inside parentheses, you ensure that this is executed first and that a name is obtained from the user and stored in the `inputName` variable.

```
(inputName = prompt("Enter a name", ""))
```

Then you compare the value returned inside the parentheses — whatever was entered by the user — with an empty string (denoted by `""`). If they are not equal — that is, if the user did enter a value, you loop around again.

Now, to sort the array into order, you use the `sort()` method of the `Array` object.

```
namesArray.sort();
```

Finally, to create a string containing all values contained in the array elements with each being on a new line, you use the HTML `
` tag and write the following:

```
var namesList = namesArray.join("<br>")
document.write(namesList);
```

The code `namesArray.join("
")` creates the string of array elements with a `
` between each. Finally, you write the string into the page with `document.write()`.

Question 3

You saw earlier in the chapter when looking at the `pow()` method how you could use it inventively to fix a number to a certain number of decimal places. However, there is a flaw in the function you created. A proper `fix()` function should return 2.1 fixed to three decimal places as

```
2.100
```

However, your `fix()` function instead returns it as

```
2.1
```

Change the `fix()` function so that the additional zeros are added where necessary.

Solution

```
<html>
<head>
<script language=JavaScript>

function fix(fixNumber, decimalPlaces)
{
    var div = Math.pow(10,decimalPlaces);
    fixNumber = new String(Math.round(fixNumber * div) / div);
    if (fixNumber.lastIndexOf(".")!=-1)
    {
        fixNumber = fixNumber + ".";
    }
}
```

Appendix A: Exercise Solutions

```
    }

    var zerosRequired = decimalPlaces -
        (fixNumber.length - fixNumber.lastIndexOf(".") - 1);

    for (; zerosRequired > 0; zerosRequired--)
    {
        fixNumber = fixNumber + "0";
    }

    return fixNumber;
}
</script>
</head>
<body>
<script language=JavaScript>

var number1 = prompt("Enter the number with decimal places you want to fix","");
var number2 = prompt("How many decimal places do you want?","");

document.write(number1 + " fixed to " + number2 + " decimal places is: ");
document.write(fix(number1,number2));

</script>
</body>
</html>
```

Save this as `ch04_q3.htm`.

The function declaration and the first line remain the same as in the `fix()` function you saw earlier in the chapter. However, things change after that.

You create the fixed number as before, using `Math.round(fixNumber * div) / div`. What is new is that you pass the result of this as the parameter to the `String()` constructor that creates a new `String` object, storing it back in `fixNumber`.

Now you have your number fixed to the number of decimal places required, but it will still be in the form `2.1` rather than `2.100`, as required. Your next task is therefore to add the extra zeros required. To do this you need to subtract the number of digits after the decimal point from the number of digits required after the decimal point as specified in `decimalPlaces`. First, to find out how many digits are after the decimal point, you write this:

```
(fixNumber.length - fixNumber.lastIndexOf(".") - 1)
```

For your number of `2.1`, `fixNumber.length` will be 3. `fixNumber.lastIndexOf(".")` will return 1; remember that the first character is 0, the second is 1, and so on. So `fixNumber.length - fixNumber.lastIndexOf(".")` will be 2. Then you subtract 1 at the end, leaving a result of 1, which is the number of digits after the decimal place.

The full line is as follows:

```
var zerosRequired = decimalPlaces -
    (fixNumber.length - fixNumber.lastIndexOf(".") - 1);
```

You know the last bit (`fixNumber.length - fixNumber.lastIndexOf(".") - 1`) is 1 and that the `decimalPlaces` parameter passed is 3. Three minus one leaves two zeros that must be added.

Now that you know how many extra zeros are required, let's add them.

```
for (; zerosRequired > 0; zerosRequired--)
{
    fixNumber = fixNumber + "0";
}
```

Now you just need to return the result from the function to the calling code.

```
return fixNumber;
```

Chapter 5

This chapter dealt with how JavaScript uses the objects, methods, properties, and events made available by the BOM (Browser Object Model) of the user's browser.

Question 1

Create a page with a number of links. Then write code that fires on the window `onload` event, displaying the `href` of each of the links on the page.

Solution

```
<html>
<head>
<script language="JavaScript" type="text/javascript">
function displayLinks()
{
    var linksCounter;

    for (linksCounter = 0; linksCounter < document.links.length; linksCounter++)
    {
        alert(document.links[linksCounter].href);
    }
}
</script>
</head>
<body onload="displayLinks()">

<A href="link0.htm" >Link 0</A>
<A href="link1.htm">Link 2</A>
<A href="link2.htm">Link 2</A>

</body>
</html>
```

Appendix A: Exercise Solutions

Save this as `ch05_q1.htm`.

You connect to the `window` object's `onload` event handler by adding an attribute to the `<body>` tag.

```
<body onload="displayLinks()">
```

On the `onload` event firing, this will run the script in quotes calling the `displayLinks()` function.

In this function you use a `for` loop to cycle through each `A` object in the `document` object's `links` array.

```
function displayLinks()
{
    var linksCounter

    for (linksCounter = 0; linksCounter < document.links.length; linksCounter++)
    {
        alert(document.links[linksCounter].href);
    }
}
```

You used the `length` property of the `links` array in your condition to determine how many times you need to loop. Then, using an `alert` box, you display each `A` object's `href` property. You can't use `document.write()` in the `onload` event, because it occurs when the page has finished loading.

Question 2

Create two pages, one called `IEOnly.htm` and the other called `FFOnly.htm`. Each page should have a heading telling you what page is loaded, for example:

```
<H2>Welcome to the Internet Explorer only page</H2>
```

Using the functions for checking browser type, connect to the `window` object's `onload` event handler and detect what browser the user has. Then if it's the wrong page for that browser, redirect to the other page.

Solution

The `FFOnly.htm` page is as follows:

```
<html>
<head>
<script language="JavaScript" type="text/javascript">

function getBrowserName()
{
    var lsBrowser = navigator.userAgent;
    if (lsBrowser.indexOf("MSIE") >= 0)
    {
        lsBrowser = "MSIE";
    }
    else if (lsBrowser.indexOf("Netscape") >= 0)
```



```

    {
        lsBrowser = "Netscape";
    }
    else if (lsBrowser.indexOf("Firefox") >= 0)
    {
        lsBrowser = "Firefox";
    }
    else if (lsBrowser.indexOf("Safari") >= 0)
    {
        lsBrowser = "Safari";
    }
    else if (lsBrowser.indexOf("Opera") >= 0)
    {
        lsBrowser = "Opera";
    }
    else
    {
        lsBrowser = "UNKNOWN";
    }
    return lsBrowser;
}

function checkBrowser()
{
    if (getBrowserName() == "MSIE")
    {
        window.location.replace("IEOnly.htm");
    }
}

</script>
</head>
<body onload="checkBrowser()">

<H2>Welcome to the Firefox only page</H2>

</body>
</html>

```

The IEOnly.htm page is very similar:

```

<html>
<head>
<script language="JavaScript" type="text/javascript">

function getBrowserName()
{
    var lsBrowser = navigator.userAgent;
    if (lsBrowser.indexOf("MSIE") >= 0)
    {
        lsBrowser = "MSIE";
    }
    else if (lsBrowser.indexOf("Netscape") >= 0)

```

Appendix A: Exercise Solutions

```
{
    lsBrowser = "Netscape";
}
else if (lsBrowser.indexOf("Firefox") >= 0)
{
    lsBrowser = "Firefox";
}
else if (lsBrowser.indexOf("Safari") >= 0)
{
    lsBrowser = "Safari";
}
else if (lsBrowser.indexOf("Opera") >= 0)
{
    lsBrowser = "Opera"
}
else
{
    lsBrowser = "UNKNOWN"
}
return lsBrowser;
}
```

```
function checkBrowser()
```

```
{
    if (getBrowserName() == "Firefox")
    {
        window.location.replace("FFOnly.htm");
    }
}
```

```
</script>
```

```
</head>
```

```
<body onload="checkBrowser()">
```

```
<H2>Welcome to the Internet Explorer only page</H2>
```

```
</body>
```

```
</html>
```

Starting with the `IEOnly.htm` page, first you add an `onload` event handler, so that on loading of the page, your `checkBrowser()` function is called.

```
<body onload="checkBrowser()">
```

Then, in `checkBrowser()`, you use your `getBrowserName()` function to tell you which browser the user has. If it's Firefox, you replace the page loaded with the `FFOnly.htm` page. Note that you use `replace()` rather than `href`, because you don't want the user to be able to hit the browser's Back button. This way it's less easy to spot that a new page is being loaded.

```
function checkBrowser()
{
    if (getBrowserName() == "Firefox")
    {
```

```
        window.location.replace("FFOnly.htm");
    }
}
```

The `FFOnly.htm` page is identical, except that in your `if` statement you check for `MSIE` and redirect to `IEOnly.htm` if it is `MSIE`.

```
function checkBrowser()
{
    if (getBrowserName() == "MSIE")
    {
        window.location.replace("IEOnly.htm");
    }
}
```

Question 3

Insert an image in the page with the `` tag. When the mouse pointer rolls over the image, it should switch to a different image. When the mouse pointer rolls out (leaves the image), it should swap back again.

Solution

```
<html>
<head>
<script language="JavaScript" type="text/javascript">

function mouseOver()
{
    document.images["myImage"].src = "Img2.jpg";
}

function mouseOut()
{
    document.images["myImage"].src = "Img1.jpg";
}
</script>
</head>
<body>

</body>
</html>
```

Save this as `ch05_q3.htm`.

Appendix A: Exercise Solutions

At the top of the page you define your two functions to handle the `onmouseover` and `onmouseout` events.

```
function mouseOver()
{
    document.images["myImage"].src = "Img2.jpg";
}

function mouseOut()
{
    document.images["myImage"].src = "Img1.jpg";
}
```

The function names tell you what events they will be handling. You access the `img` object for your `` tag using the `document.images` array and putting the name in square brackets. In the `onmouseover` event you change the `src` property of the image to `Img2.jpg`, and in the `onmouseout` event you change it back to `img1.jpg`, the image you specified when the page was loaded.

In the page itself you have your `` tag.

```

```

Chapter 6

In this chapter you looked at how to add a user interface onto your JavaScript so that you can interact with your users and acquire information from them.

Question 1

Using the code from the temperature converter example you saw in Chapter 2, create a user interface for it and connect it to the existing code so that the user can enter a value in degrees Fahrenheit and convert it to centigrade.

Solution

```
<html>
<head>
<script language="JavaScript" type="text/javascript">
function convertToCentigrade(degFahren)
{
    var degCent;
    degCent = 5/9 * (degFahren - 32);

    return degCent;
}

function butToCent_onclick()
```

```

{
    var CalcBox = document.form1.txtCalcBox;

    if (isNaN(CalcBox.value) == true || CalcBox.value == "")
    {
        CalcBox.value = "Error Invalid Value";
    }
    else
    {
        CalcBox.value = convertToCentigrade(CalcBox.value);
    }
}
</script>
</head>
<body>

<form name=form1>
<P>
    <input type="text" name=txtCalcBox value="0.0">
</P>
    <input type="button"
        value="Convert to centigrade"
        name=butToCent
        onclick="butToCent_onclick()">
</form>
</body>
</html>

```

Save this as `ch06_q1.htm`.

The interface part is simply a form containing a text box into which users enter the Fahrenheit value and a button they click to convert that value to centigrade. The button has its `onclick` event handler set to call a function called `butToCent_onclick()`.

The first line of `butToCent_onclick()` declares a variable and sets it to reference the object representing the text box.

```
var CalcBox = document.form1.txtCalcBox;
```

Why do this? Well, in your code when you want to use `document.form1.txtCalcBox`, you can now just use the much shorter `CalcBox`; it saves typing and keeps your code shorter and easier to read.

So

```
alert(document.form1.txtCalcBox.value);
```

is the same as

```
alert(CalcBox.value);
```

In the remaining part of the function you do a sanity check—if what the user has entered is a number (that is, it is not `NaN`) and the text box does contain a value, you use the Fahrenheit-to-centigrade

Appendix A: Exercise Solutions

conversion function you saw in Chapter 2 to do the conversion, the results of which are used to set the text box's value.

Question 2

Create a user interface that allows the user to pick the computer system of her dreams, similar in principle to the e-commerce sites selling computers over the Internet. For example, she could be given a choice of processor type, speed, memory, and hard drive size, and the option to add additional components like a DVD-ROM drive, a sound card, and so on. As the user changes her selections, the price of the system should update automatically and notify her of the cost of the system as she has specified it, either by using an alert box or by updating the contents of a text box.

Solution

```
<html>
<head>

<script language="JavaScript" type="text/javascript">
var CompItems = new Array();
CompItems[100] = 1000;
CompItems[101] = 1250;
CompItems[102] = 1500;

CompItems[200] = 35;
CompItems[201] = 65;
CompItems[202] = 95;

CompItems[300] = 50;
CompItems[301] = 75;
CompItems[302] = 100;

CompItems[400] = 10;
CompItems[401] = 15;
CompItems[402] = 25;
function updateOrderDetails()
{
    var total = 0;
    var orderDetails = "";
    var formElement;
    formElement =
        document.form1.cboProcessor[document.form1.cboProcessor.selectedIndex];
    total = parseFloat(CompItems[formElement.value]);
    orderDetails = "Processor : " + formElement.text;
    orderDetails = orderDetails + " $" + CompItems[formElement.value] + "\n";

    formElement =
        document.form1.cboHardDrive[document.form1.cboHardDrive.selectedIndex];
    total = total + parseFloat(CompItems[formElement.value]);
    orderDetails = orderDetails + "Hard Drive : " + formElement.text;
    orderDetails = orderDetails + " $" + CompItems[formElement.value] + "\n";

    formElement = document.form1.chkCDROM
```

```

    if (formElement.checked == true)
    {
        orderDetails = orderDetails + "CD-ROM : $" +
            CompItems[formElement.value] + "\n";
        total = total + parseFloat(CompItems[formElement.value]);
    }

    formElement = document.form1.chkDVD
    if (formElement.checked == true)
    {
        orderDetails = orderDetails + "DVD-ROM : $" +
            CompItems[formElement.value] + "\n";
        total = total + parseFloat(CompItems[formElement.value]);
    }

    formElement = document.form1.chkScanner
    if (formElement.checked == true)
    {
        orderDetails = orderDetails + "Scanner : $" +
            CompItems[formElement.value] + "\n";
        total = total + parseFloat(CompItems[formElement.value]);
    }

    formElement = document.form1.radCase
    if (formElement[0].checked == true)
    {
        orderDetails = orderDetails + "Desktop Case : $" +
            CompItems[formElement[0].value];
        total = total + parseFloat(CompItems[formElement[0].value]);
    }
    else if (formElement[1].checked == true)
    {
        orderDetails = orderDetails + "Mini Tower Case : $" +
            CompItems[formElement[1].value];
        total = total + parseFloat(CompItems[formElement[1].value]);
    }
    else
    {
        orderDetails = orderDetails + "Full Tower Case : $" +
            CompItems[formElement[2].value];
        total = total + parseFloat(CompItems[formElement[2].value]);
    }

    orderDetails = orderDetails + "\n\nTotal Order Cost is $" + total;

    document.form1.txtOrder.value = orderDetails;
}

</script>
</head>

<body>
<form name="form1">
<table>
<TR>

```

Appendix A: Exercise Solutions

```
<TD width="300">
Processor
<br>
<select name=cboProcessor>
  <option value="100">MegaPro 10ghz</option>
  <option value="101">MegaPro 12</option>
  <option value="102">MegaPro 15ghz</option>
</select>
<br><br>
Hard drive
<br>
<select name=cboHardDrive>
  <option value="200">30tb</option>
  <option value="201">40tb</option>
  <option value="202">60tb</option>
</select>
<br><br>
CD-ROM
<input type="checkbox" name=chkCDROM value="300">
<br>
DVD-ROM
<input type="checkbox" name=chkDVD value="301">
<br>
Scanner
<input type="checkbox" name=chkScanner value="302">
<br><br>
Desktop Case
<input type="radio" name=radCase checked value="400">
<br>
Mini Tower
<input type="radio" name=radCase value="401">
<br>
Full Tower
<input type="radio" name=radCase value="402">
<P>
<input type="button" value="Update" name=butUpdate onclick="updateOrderDetails()">
</P>
</TD>
<TD>
<textarea rows="20" cols="35" id=txtOrder name="txtOrder">
</textarea>
</TD>
</TR>
</table>
</form>
</body>
</html>
```

Save this as `ch06_q2.htm`.

This is just one of many ways to tackle this question — you may well have thought of a better way.

Here we are displaying the results of the user's selection as text in a `textarea` box, with each item and its cost displayed on separate lines and a final total at the end.

Each form element has a value set to hold a stock ID number. For example, a full tower case is stock ID 402. The actual cost of the item is held in arrays defined at the beginning of the page. Why not just store the price in the `value` attribute of each form element? Well, this way is more flexible. Currently your array just holds price details for each item, but we could modify it that so it holds more data—for example price, description, number in stock, and so on. Also, if this form is posted to a server the values passed will be stock IDs, which we could then use for a lookup in a stock database. If the values were set to prices and the form were posted, we'd have no way of telling what the customer ordered—all we'd know is how much it all cost.

This solution includes an Update button which, when clicked, updates the order details in the `textarea` box. However, you may want to add event handlers to each form element and update when anything changes.

Turning to the function that actually displays the order summary, `updateOrderDetails()`, we can see that there is a lot of code, and although it looks complex, it's actually fairly simple. A lot of it is repeated with slight modification.

To save on typing and make the code a little more readable, this solution declares a variable, `formElement`, which will be set to each element on the form in turn and used to extract the stock ID and, from that, the price. After the variable's declaration, we then find out which processor has been selected, calculate the cost, and add the details to the `textarea`.

```
formElement =
    document.form1.cboProcessor[document.form1.cboProcessor.selectedIndex];
total = parseFloat(CompItems[formElement.value]);
orderDetails = "Processor : " + formElement.text;
orderDetails = orderDetails + " $" + CompItems[formElement.value] + "\n";
```

The `selectedIndex` property tells us which `Option` object inside the select control has been selected by the user, and we set our `formElement` variable to reference that.

The same principle applies when we find the hard drive size selected, so let's turn next to the check boxes for the optional extra items, looking first at the CD-ROM check box.

```
formElement = document.form1.chkCDROM
if (formElement.checked == true)
{
    orderDetails = orderDetails + "CD-ROM : $" +
        CompItems[formElement.value] + "\n";
    total = total + parseFloat(CompItems[formElement.value]);
}
```

Again, we set the `formElement` variable to now reference the `chkCDROM` check box object. Then, if the check box is checked, we add a CD-ROM to the order details and update the running total. The same principle applies for the DVD and scanner check boxes.

Finally, we have the case type. Because only one case type out of the options can be selected, we've used a radio button group. Unfortunately, there is no `selectedIndex` for radio buttons as there is for check boxes, so we have to go through each radio button in turn and find out if it has been selected.

```
formElement = document.form1.radCase
if (formElement[0].checked == true)
```

Appendix A: Exercise Solutions

```
{
    orderDetails = orderDetails + "Desktop Case : $" +
        CompItems[formElement[0].value];
    total = total + parseFloat(CompItems[formElement[0].value]);
}
else if (formElement[1].checked == true)
{
    orderDetails = orderDetails + "Mini Tower Case : $" +
        CompItems[formElement[1].value];
    total = total + parseFloat(CompItems[formElement[1].value]);
}
else
{
    orderDetails = orderDetails + "Full Tower Case : $" +
        CompItems[formElement[2].value];
    total = total + parseFloat(CompItems[formElement[2].value]);
}
```

We check to see which radio button has been selected and add its details to the `textarea` and its price to the total. If our array of stock defined at the beginning of the code block had further details, such as description as well as price, we could have looped through the radio button array and added the details based on the `CompItems` array.

Finally, we set the `textarea` to the details of the system the user has selected.

```
orderDetails = orderDetails + "\n\nTotal Order Cost is " + total;
document.form1.txtOrder.value = orderDetails;
```

Chapter 7

In this chapter you looked at how you could put frames into your web pages and how to write JavaScript code that scripted between frames. You also learned how to open and modify new browser windows using script.

Question 1

In the previous chapter's exercise questions, you created a form that allowed the user to pick a computer system. He could view the details of his system and its total cost by clicking a button that wrote the details to a `textarea`. Change the example so it's a frames-based web page; instead of writing to a text area, the user should write the details to another frame.

Solution

The solution shown here involves a frameset that divides the page into two frames, left and right. In the left frame you have the form that allows the user to pick his system. The system chosen is summarized in the right frame when the user clicks an Update button.

The first page is the top frameset-defining page and is the one that needs to be loaded into the browser first.

```
<html>
<head>
</head>
<frameset cols="55%,*">
  <frame src="PickSystem.htm" name="pickSystem">
  <frame src="blank.htm" name="systemSummary">
</frameset>

</html>
```

Save this as `ch7Q1TopFrame.htm`.

Finally, you come to the page that's loaded into the left frame and that allows the user to choose his computer system and its components. This is very similar to the solution to Question 2 in the previous chapter, so this example will show only what's been changed. All the changes are within the `updateOrderDetails()` function.

```
function updateOrderDetails()
{
  var total = 0;
  var orderDetails = "<H3>Your selected system</H3>";
  var formElement;
  formElement =
    document.form1.cboProcessor[document.form1.cboProcessor.selectedIndex];
  total = parseFloat(CompItems[formElement.value]);
  orderDetails = orderDetails + "Processor : " + formElement.text
  orderDetails = orderDetails + " $" + CompItems[formElement.value] + "<br>";

  formElement =
    document.form1.cboHardDrive[document.form1.cboHardDrive.selectedIndex];
  total = total + parseFloat(CompItems[formElement.value]);
  orderDetails = orderDetails + "Hard Drive : " + formElement.text
  orderDetails = orderDetails + " $" + CompItems[formElement.value] + "<br>";

  formElement = document.form1.chkCDROM
  if (formElement.checked == true)
  {
    orderDetails = orderDetails + "CD-ROM : $" +
      CompItems[formElement.value] + "<br>";
    total = total + parseFloat(CompItems[formElement.value]);
  }

  formElement = document.form1.chkDVD
  if (formElement.checked == true)
  {
    orderDetails = orderDetails + "DVD-ROM : $" +
      CompItems[formElement.value] + "<br>";
    total = total + parseFloat(CompItems[formElement.value]);
  }

  formElement = document.form1.chkScanner
```

Appendix A: Exercise Solutions

```
if (formElement.checked == true)
{
    orderDetails = orderDetails + "Scanner : $" +
        CompItems[formElement.value] + "<br>";
    total = total + parseFloat(CompItems[formElement.value]);
}

formElement = document.form1.radCase
if (formElement[0].checked == true)
{
    orderDetails = orderDetails + "Desktop Case : $" +
        CompItems[formElement[0].value] + "<br>";
    total = total + parseFloat(CompItems[formElement[0].value]);
}
else if (formElement[1].checked == true)
{
    orderDetails = orderDetails + "Mini Tower Case : $" +
        CompItems[formElement[1].value] + "<br>";
    total = total + parseFloat(CompItems[formElement[1].value]);
}
else
{
    orderDetails = orderDetails + "Full Tower Case : $" +
        CompItems[formElement[2].value] + "<br>";
    total = total + parseFloat(CompItems[formElement[2].value]);
}

orderDetails = orderDetails + "<P>Total Order Cost is $" + total + "</P>";

window.parent.systemSummary.document.open();
window.parent.systemSummary.document.write(orderDetails);
window.parent.systemSummary.document.close();
}
```

One final optional change is to remove the text area control, as it's no longer needed.

Save this as `PickSystem.htm`, and load `ch7Q1TopFrame.htm` into your browser to try out the code.

The first difference between this and the code from Question 2 in the last chapter is that when creating the text summarizing the system, you are creating HTML rather than plain text, so rather than `\n` for new lines you use the `
` tag.

The main change, however, is the following three lines:

```
window.parent.systemSummary.document.open();
window.parent.systemSummary.document.write(orderDetails);
window.parent.systemSummary.close();
```

Instead of setting the value of a text area box as you did in the solution to Question 2 in the last chapter, this time you are writing the order summary to an HTML page, the page contained in the right-hand frame, `systemSummary`. First you open the document for writing, then write out your string, and finally close the document, indicating that you have completed your writing to the page.

Question 2

The first example in this chapter was a page with images of books, in which clicking on a book's image brought up information about that book in a pop-up window. Amend this so that the pop-up window also has a button or link that, when clicked, adds the item to the user's shopping basket. Also, on the main page, give the user some way of opening up a shopping basket window with details of all the items he has purchased so far, and give him a way of deleting items from this basket.

Solution

This is the most challenging exercise so far, but by the end you'll see how a more complex application can be created using JavaScript. The solution to this exercise involves four pages: two that display the book's details (very similar to the pages you created in the example), a third that displays the book's images and opens the new windows, and a fourth, totally new page, which holds the shopping basket.

Let's look at the main page to be loaded, called `online_books.htm`.

```
<html>
<head>
<title>Online Books</title>
<script language="JavaScript" type="text/javascript">
var detailsWindow;
var basketWindow;

var stockItems = new Array();

stockItems[100] = new Array();
stockItems[100][0] = "Beginning ASP.net 2";
stockItems[100][1] = "$39.99";
stockItems[100][2] = 0;

stockItems[101] = new Array();
stockItems[101][0] = "Professional JavaScript";
stockItems[101][1] = "$46.99";
stockItems[101][2] = 0;

function removeItem(stockId)
{
    stockItems[stockId][2] = 0;
    alert("Item Removed");
    showBasket();
    return false;
}

function showDetails(bookURL)
{
    detailsWindow = window.open(bookURL, "bookDetails", "width=400,height=500");
    detailsWindow.focus();
    return false;
}

function addBookToBasket(stockId)
```

Appendix A: Exercise Solutions

```

{
    stockItems[stockId][2] = 1;
    alert("Item added successfully");
    detailsWindow.close();
}

function showBasket()
{
    basketWindow =
        window.open('ShoppingBasket.htm', 'shoppingBasket', 'width=400,height=350');
    basketWindow.document.open();
    var basketItem;
    var containsItems = false;
    basketWindow.document.write("<H4>Your shopping basket contains :</H4>");

    for (basketItem in stockItems)
    {
        if (stockItems[basketItem][2] > 0)
        {
            basketWindow.document.write(stockItems[basketItem][0] + " at ");
            basketWindow.document.write(stockItems[basketItem][1]);
            basketWindow.document.write(" &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;");
            basketWindow.document.write("<A href='\"' onclick='return \" +
\"window.opener.removeItem(\" + basketItem + \"')>");
            basketWindow.document.write("Remove Item</A><br>");
            containsItems = true;
        }
    }

    if (containsItems == false)
    {
        basketWindow.document.write("<H4>No items</H4>");
    }
    basketWindow.document.close();
    basketWindow.focus();
}

</script>
</head>
<body>
<H2 align=center>Online Book Buyer</H2>
<form name=form1>
<input type="button" value="Show Shopping Basket" onclick="showBasket()"
name=butShowBasket>
</form>
<P>
Click any of the images below for more details
</P>
<strong>Beginning ASP.net 2</strong>
<br>
<A name="begASPLink" href="" onclick="return showDetails('beg_asp2_details.htm')">
</A>
<br>
<br>
<strong>Professional JavaScript</strong>

```

```
<br>
<A name="profJSLink" href=""
onclick="return showDetails('prof_js_details.htm')">

</A>
</body>
</html>
```

*Please note the line `basketWindow.document.write("<A href=" onclick='return " + "`
`window.opener.removeItem(" + basketItem + ")>");` must be on one line. Because of the size of the page it has been split in the preceding code listing.*

The details of the books are stored in the `stockItems` array, which you've made a multi-dimensional array. The second dimension stores the book's title, its price, and finally the quantity the customer has in his basket.

So in the first element, you have this:

```
stockItems[100] = new Array();
stockItems[100][0] = "Beginning ASP.Net 2";
stockItems[100][1] = "$39.99";
stockItems[100][2] = 0;
```

`[100][0]` is the title, `[100][1]` is the price, and finally `[100][2]` is the quantity required, which starts as 0. In fact, though a customer may order more than one of a certain book, the code does not facilitate that.

The first function defined in the code is `removeItem()`.

```
function removeItem(stockId)
{
    stockItems[stockId][2] = 0;
    alert("Item Removed");
    showBasket();
    return false;
}
```

This removes a book from the shopping basket. The parameter `stockId` is simply the array element index of that book, which you then use to set the quantity element of the second dimension to 0.

Next, you have the function that adds a book to the shopping basket.

```
function addBookToBasket(stockId)
{
    stockItems[stockId][2] = 1;
    alert("Item added successfully");
    detailsWindow.close();
}
```

The final function displays the contents of the shopping basket in a new window.

Appendix A: Exercise Solutions

```
function showBasket()
{
    basketWindow =
        window.open('ShoppingBasket.htm','shoppingBasket','width=400,height=350');
    basketWindow.document.open();
    var basketItem;
    var containsItems = false;
    basketWindow.document.write("<H4>Your shopping basket contains :</H4>");

    for (basketItem in stockItems)
    {
        if (stockItems[basketItem][2] > 0)
        {
            basketWindow.document.write(stockItems[basketItem][0] + " at ");
            basketWindow.document.write(stockItems[basketItem][1]);
            basketWindow.document.write(" &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;");
            basketWindow.document.write("<A href='' onclick='return " +
                " window.opener.removeItem(" + basketItem + ")'>" );
            basketWindow.document.write("Remove Item</A><br>");
            containsItems = true;
        }
    }

    if (containsItems == false)
    {
        basketWindow.document.write("<H4>No items</H4>");
    }
    basketWindow.document.close();
    basketWindow.focus();
}
```

A new window is opened up and its window object reference is stored in `basketWindow`. You then write to the new window's document. First you write a heading, and then you loop through each item in the `stockItems` array and check the quantity element of the second dimension, which is `stockItems[basketItem][2]`. If it is greater than zero, you write the book's details to the shopping list window. You also write out a link to the shopping basket that when clicked calls your `removeItem()` function.

Let's create the shopping basket page.

```
<html>
<head>
<title>Shopping Basket</title>
</head>
<body>
</body>
</html>
```

Save this as `ShoppingBasket.htm`. There's no code, but if you don't create the page and load it into the shopping basket window, you won't be able to `document.write()` to it.

Finally, you need to create the book description pages. First you have `prof_js_details.htm`. This is identical to the version you created for the example, except for the addition of the form and button

inside. When clicked, the button calls the `addToBasket()` function in the window that opened this window — that is, your `online_books.htm` page.

```
<html>
<head>
<title>Professional JavaScript</title>
</head>
<body><strong>Professional JavaScript</strong>
<br>
<form name=form1>
<input type="button" value="Add to basket" name=butAddBook
      onclick="window.opener.addBookToBasket(101)">
</form>

<strong>Subjects</strong>
  ECMAScript<br>
  Internet<br>JavaScript
  <br>XML and Scripting<BR>

<HR color=#cc3333>
<P>This book covers the broad spectrum of programming JavaScript - from the core
language to browser applications and server-side use to stand-alone and embedded
JavaScript.
</P>
<P>
It includes a guide to the language - when where and how to get the
most out of JavaScript - together with practical case studies demonstrating
JavaScript in action. Coverage is bang up-to-date, with discussion of
compatibility issues and version differences, and the book concludes with a
comprehensive reference section. </P>

</body>
</html>
```

Finally, you have your `beg_asp2_details.htm` page. Again, it is identical to the version created in the example, with a form and button to add the book to the shopping basket, as in the preceding page.

```
<html>
<head>
<title>Beginning ASP.Net 2</title>
</head>
<body>
<strong>Beginning ASP.Net 2</strong>
<form name=form1>
<input type="button" value="Add to basket" name=butAddBook
      onclick="window.opener.addBookToBasket(100)">
</form>

<br>
Subjects
<br>
ASP
<br>
Internet
```

Appendix A: Exercise Solutions

```
<br>

<HR color=#cc3333>

<P>ASP.Net 2 is the most recent version of ASP and this book covers it in-depth in
a clear and easy to read manner.
</P>

</body>
</html>
```

Chapter 8

In this chapter you looked at string manipulation using the `string` object, and the use of the `Regex` object to match patterns of characters within strings.

Question 1

What problem does the code below solve?

```
var myString = "This sentence has has a fault and and we need to fix it."
var myRegex = /(\b\w+\b) \1/g;
myString = myString.replace(myRegex, "$1");
```

Now imagine that you change that code, so that you create the `Regex` object like this:

```
var myRegex = new Regex("(\b\w+\b) \1");
```

Why would this not work, and how could you rectify the problem?

Solution

The problem is that the sentence has “has has” and “and and” inside it, clearly a mistake. A lot of word processors have an autocorrect feature that fixes common mistakes like this, and what your regular expression does is mimic this feature.

So the erroneous `myString`

“This sentence has has a fault and and we need to fix it.”

will become

“This sentence has a fault and we need to fix it.”

Let’s look at how the code works, starting with the regular expression.

```
/(\b\w+\b) \1/g;
```

By using parentheses you have defined a group, so `(\b\w+\b)` is group one. This group matches the pattern of a word boundary followed by one or more alphanumeric characters — that is, characters a–z, A–Z, 0–9, and `_` — followed by a word boundary. Following the group you have a space, then `\1`. What `\1` means is “Match exactly the same characters as were matched in pattern group one.” So, for example, if group one matched “has,” then `\1` will match “has” as well. It’s important to note that `\1` will match the exact previous match by group one. So when group one matches the “and,” the `\1` now also matches “and” and not the “has” that was previously matched.

You use the group again in your `replace()` method; this time the group is specified by means of the `$` symbol, so `$1` matches group one. It’s this that causes the two matched duplicated words, “has” and “and,” to be replaced by just one word in each instance.

Turning to the second part of the question, how do you change the following code so that it works?

```
var myRegExp = new RegExp("(\b\w+\b) \1");
```

Easy: Now you are using a string passed to the `RegExp` object’s constructor, and you need to use two slashes rather than one when you mean a regular expression syntax character, like this:

```
var myRegExp = new RegExp("(\\b\\w+\\b) \\1", "g");
```

Notice you’ve also passed a `g` to the second parameter to make it a global match.

Question 2

Write a regular expression that finds all of the occurrences of the word “a” in the following sentence and replaces them with “the”:

“a dog walked in off a street and ordered a finest beer”

Solution

```
<html>
<body>
<script language="JavaScript" type="text/javascript">
var myString = "a dog walked in off a street and ordered a finest beer";
var myRegExp = /\ba\b/gi;
myString = myString.replace(myRegExp, "the");
alert(myString);
</script>
</body>
</html>
```

Save this as `ch08_q2.htm`.

With regular expressions, it’s often not just what you want to match, but also what you don’t want to match that is a problem. Here you want to match the letter *a*, so why not just write this?

```
var myRegExp = /a/gi;
```

Appendix A: Exercise Solutions

Well, that would work, but it would also replace the *a* in “walked,” which you don’t want. You want to replace the letter *a*, but only where it’s a word on its own and not inside another word. So when does a letter become a word? The answer is when it’s between two word boundaries. The word boundary is represented by the regular expression special character `\b`, so the regular expression becomes this:

```
var myRegExp = /\ba\b/gi;
```

The `gi` at the end ensures a global, case-insensitive search.

Now, with your regular expression created, you can use it in the `replace()` method’s first parameter.

```
myString = myString.replace(myRegExp, "the");
```

Question 3

Imagine you have a website with a message board. Write a regular expression that would remove barred words. (You can make up your own words.)

Solution

```
<html>
<body>
<script language="JavaScript" type="text/javascript">
  var myRegExp = /(sugar )?candy|choc(olate|oholic)?/gi;
  var myString = "Mmm, I love chocolate, I'm a chocoholic. " +
    "I love candy too, sweet, sugar candy";
  myString = myString.replace(myRegExp, "salad");
  alert(myString)
</script>
</body>
</html>
```

Save this as `ch08_q3.htm`.

For this example, pretend you’re creating script for a board on a dieting site where text relating to candy is barred and will be replaced with references to a much healthier option, salad.

The barred words are:

- ☐ chocolate
- ☐ choc
- ☐ chocoholic
- ☐ sugar candy
- ☐ candy

Let’s see how you can build up the regular expression to remove the offending words.

Start with the two basic words, so to match *choc* or *candy*, use this:

```
candy|choc
```

Next, add the matching for *sugar candy*. Since the *sugar* bit is optional, you group it by placing it in parentheses and adding the question mark after it. This means “Match the group zero times or one time.”

```
(sugar )?candy|choc
```

Finally you need to add the optional *olate* and *oholic* end bits. You add these as a group after the “choc” word and again make the group optional. You can match either of the endings in the group by using the `|` character.

```
(sugar )?candy|choc(olate|oholic)?/gi
```

Finally, you declare it as follows:

```
var myRegExp = /(sugar )?candy|choc(olate|oholic)?/gi
```

The `gi` at the end means that the regular expression will find and replace words on a global, case-insensitive basis.

So, to sum up:

```
/(sugar )?candy|choc(olate|oholic)?/gi
```

reads as:

Either match zero or one occurrences of “sugar” followed by “candy.” Or alternatively match “choc” followed by either one or zero occurrences of “olate” or match “choc” followed by zero or one occurrence of “oholic.”

Finally, the following:

```
myString = myString.replace(myRegExp, "salad");
```

replaces the offending words with “salad” and sets `myString` to the new clean version:

“Mmm, I love salad, I’m a salad. I love salad too, sweet, salad.”

Chapter 9

In this chapter you looked in more detail at the `Date` object, particularly with respect to world time and local time. You also looked at how to create timers to trigger code on a web page.

Appendix A: Exercise Solutions

Question 1

Create a web page with an advertisement image at the top. When the page loads, select a random image for that advertisement. Every four seconds, make the image change to a different one, making sure a different advertisement is selected until all the advertisement images have been seen.

Solution

```
<html>
<head>
<script language="JavaScript" type="text/javascript">

var imagesSelected = new Array(false,false,false);
var noOfImages = 3;
var totalImagesSelected = 0;

function window_onload()
{
    setInterval("switchImage()",4000);
}

function switchImage()
{
    var imageIndex;

    if (totalImagesSelected == noOfImages)
    {
        for (imageIndex = 0; imageIndex < noOfImages; imageIndex++)
        {
            imagesSelected[imageIndex] = false;
        }

        totalImagesSelected = 0;
    }

    var selectedImage = Math.floor(Math.random() * noOfImages) + 1;
    while (imagesSelected[selectedImage - 1] == true)
    {
        selectedImage = Math.floor(Math.random() * noOfImages) + 1;
    }
    totalImagesSelected++;
    imagesSelected[selectedImage - 1] = true;
    document.imgAdvert.src = "AdvertImage" + selectedImage + ".jpg";
}

</script>
</head>
<body onload="window_onload()">

</body>
</html>
```

Save this as `ch09_q1.htm`.

This solution is based on the example in the chapter, `Adverts.htm`, in which you displayed three images at set intervals, one after the other. The first difference is that you select a random image each time, rather than the images in sequence. Secondly, you make sure you don't select the same image twice in one sequence by having an array, `imagesSelected`, with each element of that array being `true` or `false` depending on whether the image has been selected before. Once you've shown each image, you reset the array and start the sequence of selecting images randomly again.

The final difference between this solution and the example in the chapter is that you set the timer going continuously with `setInterval()`. So until the user moves to another page, your random display of images will continue.

Question 2

Create a form that gets the user's date of birth. Then, using that information, tell her on what day of the week she was born.

Solution

```
<html>
<head>
<script language="JavaScript" type="text/javascript">

var days = new Array();
days[0] = "Sunday";
days[1] = "Monday";
days[2] = "Tuesday";
days[3] = "Wednesday";
days[4] = "Thursday";
days[5] = "Friday";
days[6] = "Saturday";

function dayOfWeek()
{
    var form = document.form1;
    var date = parseInt(form.txtDate.value)
    var year = parseInt(form.txtYear.value)

    if (isNaN(date) || isNaN(year))
    {
        alert("Please enter a valid whole number");
    }
    else
    {
        if (date < 1 || date > 31)
        {
            alert("Day of the month must be between 1 and 31");
        }
        else
```

Appendix A: Exercise Solutions

```
        {
            userDate = date + " ";
            userDate = userDate +
                form.selMonth.options[form.selMonth.selectedIndex].value;
            userDate = userDate + " " + year;
            var dateThen = new Date(userDate);
            alert(days[dateThen.getDay()]);
        }
    }
}
</script>
</head>
<body>
<P>Find the day of your birth</P>
<P>
<form name="form1">
<input type="text" name="txtDate" size="2" maxlength="2">
<select name="selMonth">
    <option selected value="Jan">Jan</option>
    <option selected value="Feb">Feb</option>
    <option selected value="Mar">Mar</option>
    <option selected value="Apr">Apr</option>
    <option selected value="May">May</option>
    <option selected value="Jun">Jun</option>
    <option selected value="Jul">Jul</option>
    <option selected value="Aug">Aug</option>
    <option selected value="Sept">Sept</option>
    <option selected value="Oct">Oct</option>
    <option selected value="Nov">Nov</option>
    <option selected value="Dec">Dec</option>
</select>
<input type="text" name="txtYear" size="4" maxlength="4">
<br>
<input type="button" value="Day of the week"
    onclick="dayOfWeek()" name="button1">
</form>
</P>

</body>
</html>
```

Save this as `ch09_q2.htm`.

The solution is surprisingly simple. You create a new `Date` object based on the date entered by the user. Then you get the day of the week using the `Date` object's `getDay()` method. This returns a number, but by defining an array of days of the week to match this number, you can use the value of `getDay()` as the index to your days array.

You also do some basic sanity checking to make sure that the user has entered numbers and that in the case of the date, the number is between 1 and 31. You could have defined a select element as the method of getting the date and only have numbers from 1 to 31. Of course, for either way, you don't check

whether invalid dates are entered (for example, the 31st of February). You might want to try this as an additional exercise.

Hint: To get the last day of the month, get the first day of the next month and then subtract one.

Chapter 10

In this chapter you looked at some common mistakes in JavaScript code, debugging code using the Microsoft script debugger, and ways of handling errors using the `try...catch` clause and the `throw` statement.

Question 1

The example `debug_timestable2.htm` has a deliberate bug. For each times table it creates only multipliers with values from 1 to 11.

Use the script debugger to work out why this is happening, and then correct the bug.

Solution

The problem is with your code's logic rather than its syntax. Logic errors are much harder to spot and deal with because, unlike with syntax errors, the browser won't inform you that there's such and such error at line so and so but instead just fails to work as expected. The error is with this line:

```
for (counter = 1; counter < 12; counter++)
```

You want your loop to go from 1 to 12 inclusive. Your `counter < 12` statement will be `true` up to and including 11 but will be `false` when the counter reaches 12; hence 12 gets left off. To correct this, you could change your code to the following:

```
for (counter = 1; counter <= 12; counter++)
```

Question 2

The following code contains a number of common errors. See if you can spot them:

```
<html>
<head>
</head>
<body>
<script language=JavaScript>
function checkForm(theForm)
{
    var formValid = true;
    var elementCount = 0;
    while(elementCount =< theForm.length)
    {
```

Appendix A: Exercise Solutions

```
        if (theForm.elements[elementCount].type == "text")
        {
            if (theForm.elements[elementCount].value() == "")
                alert("Please complete all form elements");
            theForm.elements[elementCount].focus();
            formValid = false;
            break;
        }
    }
    return formValid;
}
</script>
<form name=form1 onsubmit="return checkForm(document.form1)">
    <input type="text" ID=text1 name=text1>
    <br>
    CheckBox 1<input type="checkbox" ID=checkbox1 name=checkbox1>
    <br>
    CheckBox 2<input type="checkbox" ID=checkbox2 name=checkbox2>
    <br>
    <input type="text" ID=text2 name=text2>
    <p>
    <input type="submit" value="Submit" ID=submit1 name=submit1>
    </p>
</form>
</body>
</html>
```

Solution

The bug-free version looks like this:

```
<html>
<head>
</head>
<body>

<script language="JavaScript">
function checkForm(theForm)
{
    var formValid = true;
    var elementCount = 0;

    while(elementCount < theForm.length)
    {
        if (theForm.elements[elementCount].type == "text")
        {
            if (theForm.elements[elementCount].value == "")
            {
                alert("Please complete all form elements");
                theForm.elements[elementCount].focus();
                formValid = false;
                break;
            }
        }
    }
}
```

```

    }
}

    elementCount++;
}

    return formValid;
}

</script>

<form name="form1" onsubmit="return checkForm(document.form1)">
    <input type="text" id="text1" name="text1">
    <br>
    CheckBox 1<input type="checkbox" id="checkbox2" name="checkbox2">
    <br>
    CheckBox 1<input type="checkbox" id="checkbox1" name="checkbox1">
    <br>
    <input type="text" id="text2" name="text2">
    <P>
    <input type="submit" value="Submit" id="submit1" name="submit1">
    </P>
</form>

</body>
</html>

```

Let's look at each error in turn.

The first error is a logic error.

```
while(elementCount <= theForm.length)
```

Arrays start at 0 so the first `Form` object is at index array 0, the second at 1, and so on. The last `Form` object has an index value of 4. However, `theForm.length` will return 5 because there are five elements in the form. So the `while` loop will continue until `elementCount` is less than or equal to 5, but as the last element has an index of 4, this is one past the limit. You should write either this:

```
while(elementCount < theForm.length)
```

or this:

```
while(elementCount <= theForm.length - 1)
```

Either is fine, though the first is shorter.

You come to your second error in the following line:

```
if (theForm.elements[elementcount].type == "text")
```

Appendix A: Exercise Solutions

On a quick glance it looks fine, but it's JavaScript's strictness on case sensitivity that has caused your downfall. The variable name is `elementCount`, not `elementcount` with a lowercase *c*. So this line should read as follows:

```
if (theForm.elements[elementCount].type == "text")
```

The next line with an error is this:

```
if (theForm.elements[elementCount].value() = "")
```

This has two errors. First, `value` is a property and not a method, so there is no need for parentheses after it. Second, you have the all-time classic error of one equals sign instead of two. Remember that one equals sign means "Make it equal to," and two equals signs mean "Check if it is equal to." So with the changes, the line is:

```
if (theForm.elements[elementCount].value == "")
```

The next error is your failure to put your block of `if` code in curly braces. Even though JavaScript won't throw an error since the syntax is fine, the logic is not so fine, and you won't get the results you expect. With the braces, the `if` statement should be as follows:

```
if (theForm.elements[elementCount].value == "")
{
    alert("Please complete all form elements")
    theForm.elements[elementCount].focus;
    formValid = false;
    break;
}
```

The penultimate error is in this line:

```
theForm.elements[elementCount].focus;
```

This time you have a method but with no parentheses after it. Even methods that have no parameters must have the empty parentheses after them. So, corrected, the line is as follows:

```
theForm.elements[elementCount].focus();
```

Now you're almost done; there is just one more error. This time it's not something wrong with what's there, but rather something very important that should be there but is missing. What is it? It's this:

```
elementCount++;
```

This line should be in your `while` loop, otherwise `elementCount` will never go above 0 and the `while` loop's condition will always be `true`, resulting in the loop continuing forever: a classic infinite loop.

Chapter 11

In this chapter you looked at storing small amounts of information, called cookies, on the user's computer and using that information to customize your web site for the user.

Question 1

Create a page that keeps track of how many times the page has been visited by the user in the last month.

Solution

```
<html>
<head>
<script language="JavaScript" type="text/javascript">

function getCookieValue(cookieName)
{
    var cookieValue = document.cookie;
    var cookieStartsAt = cookieValue.indexOf(" " + cookieName + "=");

    if (cookieStartsAt == -1)
    {
        cookieStartsAt = cookieValue.indexOf(cookieName + "=");
    }

    if (cookieStartsAt == -1)
    {
        cookieValue = null;
    }
    else
    {
        cookieStartsAt = cookieValue.indexOf("=", cookieStartsAt) + 1;
        var cookieEndsAt = cookieValue.indexOf(";", cookieStartsAt);
        if (cookieEndsAt == -1)
        {
            cookieEndsAt = cookieValue.length;
        }
        cookieValue = unescape(cookieValue.substring(cookieStartsAt,
            cookieEndsAt));
    }

    return cookieValue;
}

function setCookie(cookieName,cookieValue, cookiePath, cookieExpires)
{
    cookieValue = escape(cookieValue);
    if (cookieExpires == "")
    {
        var nowDate = new Date();
        nowDate.setMonth(nowDate.getMonth() + 6);
        cookieExpires = nowDate.toGMTString();
    }

    if (cookiePath != "")
    {
        cookiePath = ";Path=" + cookiePath;
    }
}
```

Appendix A: Exercise Solutions

```
        document.cookie = cookieName + "=" + cookieValue + ";Expires=" +
            cookieExpires + cookiePath;
    }

    var pageViewCount = getCookieValue("pageViewCount");
    var pageFirstVisited = getCookieValue("pageFirstVisited");

    if (pageViewCount == null)
    {
        pageViewCount = 1;
        pageFirstVisited = new Date();
        pageFirstVisited.setMonth(pageFirstVisited.getMonth());
        pageFirstVisited = pageFirstVisited.toGMTString();
        setCookie("pageFirstVisited",pageFirstVisited,"","")
    }
    else
    {
        pageViewCount = Math.floor(pageViewCount) + 1;
    }

    setCookie("pageViewCount",pageViewCount,"","")

</script>
</head>
<body>
<script>
var pageHTML = "You've visited this page " + pageViewCount;
pageHTML = pageHTML + " times since " + pageFirstVisited;
document.write(pageHTML);
</script>
</body>
</html>
```

Save this as `ch11_q1.htm`.

We discussed the cookie functions in Chapter 11, so let's turn straight to the new code.

In the first two lines we get two cookies and store them in variables. The first cookie holds the number of visits, the second the date the page was first visited.

```
var pageViewCount = getCookieValue("pageViewCount");
var pageFirstVisited = getCookieValue("pageFirstVisited");
```

If the `pageViewCount` cookie does not exist, it's either because the cookie expired (remember that we are counting visits for the last month) or because the user has never visited our site before. Either way we need to set the `pageViewCount` to 1 and store the date the page was first visited plus one month in the `pageFirstVisited` variable. We'll need this value later when we want to set the `expires` value for the `pageViewCount` cookie we create because there is no way of using code to find out an existing cookie's expiration date.

```
if (pageViewCount == null)
{
    pageViewCount = 1;
```

```

    pageFirstVisited = new Date();
    pageFirstVisited.setMonth(pageFirstVisited.getMonth() + 1)
    pageFirstVisited = pageFirstVisited.toGMTString();
    setCookie("pageFirstVisited",pageFirstVisited,"","")
}

```

In the else statement we increase the value of pageViewCount.

```

else
{
    pageViewCount = Math.floor(pageViewCount) + 1;
}

```

We then set the cookie keeping track of the number of page visits by the user.

```

setCookie("pageViewCount",pageViewCount,"","")

```

Finally, we write out the number of page visits and the date since the counter was reset.

```

var pageHTML = "You've visited this page " + pageViewCount;
pageHTML = pageHTML + " times since " + pageFirstVisited;
document.write(pageHTML);

```

Question 2

Use cookies to load a different advertisement every time a user visits a web page.

Solution

```

<html>
<head>
<script language="JavaScript" type="text/javascript">

function getCookieValue(cookieName)
{
    var cookieValue = document.cookie;
    var cookieStartsAt = cookieValue.indexOf(" " + cookieName + "=");

    if (cookieStartsAt == -1)
    {
        cookieStartsAt = cookieValue.indexOf(cookieName + "=");
    }

    if (cookieStartsAt == -1)
    {
        cookieValue = null;
    }
    else
    {
        cookieStartsAt = cookieValue.indexOf("=", cookieStartsAt) + 1;
    }
}

```

Appendix A: Exercise Solutions

```
        var cookieEndsAt = cookieValue.indexOf(";", cookieStartsAt);
        if (cookieEndsAt == -1)
        {
            cookieEndsAt = cookieValue.length;
        }
        cookieValue = unescape(cookieValue.substring(cookieStartsAt, cookieEndsAt));
    }

    return cookieValue;
}

function setCookie(cookieName,cookieValue, cookiePath, cookieExpires)
{
    cookieValue = escape(cookieValue);
    if (cookieExpires == "")
    {
        var nowDate = new Date();
        nowDate.setMonth(nowDate.getMonth() + 6);
        cookieExpires = nowDate.toGMTString();
    }

    if (cookiePath != "")
    {
        cookiePath = ";Path=" + cookiePath;
    }
    document.cookie = cookieName + "=" + cookieValue + ";Expires=" +
        cookieExpires + cookiePath;
}

</script>
</head>
<body>

<script>

var imageNumber = getCookieValue("displayedImages");
var totalImages = 3;

if (imageNumber == null)
{
    imageNumber = "1";
}
else
{
    imageNumber = Math.floor(imageNumber) + 1;
}

if (totalImages == imageNumber)
{
    setCookie("displayedImages","", "", "Mon, 1 Jan 1970 00:00:00");
}
else
{

```



```

        setCookie("displayedImages",imageNumber,"","");
    }

    document.imgAdvert.src = "AdvertImage" + imageNumber + ".jpg";
</script>
</body>
</html>

```

Save this as `ch11_q2.htm`.

This solution is based on similar questions in previous chapters, such as Chapter 9 where we displayed a randomly selected image. In this case we display a different image in the page each time the user visits it, as far as our selection of images allows.

We've seen the cookie setting and reading functions earlier in the chapter, so let's look at the new code.

We store the number of the previously displayed images in a cookie named `displayedImages`. The next image we display is that image number plus one. Once all of our images have been displayed, we start again at 1. If the user has never been to the web site, no cookie will exist so `null` will be returned from `getCookieValue()`, in which case we set `imageNumber` to 1.

Most of the code is fairly self-explanatory, except perhaps these lines:

```

    if (totalImages == imageNumber)
    {
        setCookie("displayedImages","", "", "Mon, 1 Jan 1970 00:00:00")
    }

```

What this bit of code does is delete the cookie by setting its expiration date to a date that has already passed.

Chapter 12

In this chapter you were introduced to Dynamic HTML (DHTML), in which you used JavaScript to manipulate web pages after they were loaded into the browser to enhance user interaction.

Question 1

Create a web page that contains two links. The first link should say `Show First Box` and the second `Show Second Box`. Then add two `<div/>` elements and set their `id` attributes to `boxOne` and `boxTwo`. Give them a height, width, background color, and position, and then hide them. Next, set up the links so that when you click the first one, only the first box shows, and when you click the second one, only the second box shows.

Solution

```
<html>
<head>
  <title>Chapter 12, Question 1</title>
  <style type="text/css">
    #boxOne {
      position: absolute;
      top: 125px;
      left: 231px;
      width: 100px;
      height: 100px;
      background-color: navy;
      visibility: hidden;
    }

    #boxTwo {
      position: absolute;
      top: 100px;
      left: 400px;
      width: 200px;
      height: 200px;
      background-color: red;
      visibility: hidden;
    }
  </style>
  <script type="text/javascript">
    function showBoxOne() {
      var boxOne = document.getElementById("boxOne");
      var boxTwo = document.getElementById("boxTwo");

      boxOne.style.visibility = "visible";
      boxTwo.style.visibility = "hidden";
    }

    function showBoxTwo() {
      var boxOne = document.getElementById("boxOne");
      var boxTwo = document.getElementById("boxTwo");

      boxOne.style.visibility = "hidden";
      boxTwo.style.visibility = "visible";
    }
  </script>
</head>
<body>
  <a href="#" onclick="showBoxOne(); return false;">Show First Box</a>
  <a href="#" onclick="showBoxTwo(); return false;">Show Second Box</a>

  <div id="boxOne"></div>
  <div id="boxTwo"></div>
</body>
</html>
```

Let's look at this page in more detail, starting with the style sheet. There are two rules in this style sheet, one for each `<div/>` element in the body of the page. These rules position the elements and assign their height, width, background-color, and visibility properties.

In the body of the page, you find two `<a/>` elements and two `<div/>` elements. The `<a/>` elements are set to handle the `click` event with the `onclick` attribute. The first link calls the `showBoxOne()` JavaScript function, and the second link calls `showBoxTwo()`. Both `onclick` event handlers return a value of `false`, which tells the browser not to navigate to the URL specified in the `href` attribute.

```
<a href="#" onclick="showBoxOne(); return false;">Show First Box</a>
<a href="#" onclick="showBoxTwo(); return false;">Show Second Box</a>
```

Next are the `<div/>` elements, which contain no content. The only attribute specified is their `id` attributes.

```
<div id="boxOne"></div>
<div id="boxTwo"></div>
```

In the `<script/>` element, two functions can be found: `showBoxOne()` and `showBoxTwo()`. Their purpose is to show one `<div/>` element while hiding the other. The first, `showBoxOne()`, shows the first box.

```
function showBoxOne() {
    var boxOne = document.getElementById("boxOne");
    var boxTwo = document.getElementById("boxTwo");

    boxOne.style.visibility = "visible";
    boxTwo.style.visibility = "hidden";
}
```

The first step in this process is to retrieve the two `<div/>` elements by using `document.getElementById()`. Next, the `visibility` property for `boxOne` is set to `visible`, while the same property for `boxTwo` is set to `hidden`. This shows the first box while hiding the other.

The `showBoxTwo()` function follows the same idea, and for the most part, uses the same code as `showBoxOne()`.

```
function showBoxTwo() {
    var boxOne = document.getElementById("boxOne");
    var boxTwo = document.getElementById("boxTwo");

    boxOne.style.visibility = "hidden";
    boxTwo.style.visibility = "visible";
}
```

The only difference in this function is the values assigned to the two element's `visibility` properties. In this case, `boxOne` is set to `hidden` while `boxTwo` is set to `visible`.

Question 2

Create a `<div/>` element that floats around the page. Use the edges of the browser's viewport as a boundary.

Solution

```
<html>
<head>
<style type="text/css">
    #floatingDiv {
        position: absolute;
        left: 0px;
        top: 0px;
        width: 50px;
        height: 50px;
        background-color: navy;
    }
</style>
<script type="text/javascript">

var floatingDiv;
var screenWidth;
var screenHeight;

var horizontalMovement = Math.ceil(Math.random() * 5);
var verticalMovement = Math.ceil(Math.random() * 5);

function startTimer() {
    floatingDiv = document.getElementById("floatingDiv");
    screenWidth = document.body.clientWidth;
    screenHeight = document.body.clientHeight;

    window.setInterval("moveDiv()", 10);
}

function moveDiv() {
    var currentLeft = floatingDiv.offsetLeft;
    var currentTop = floatingDiv.offsetTop;

    if (currentTop < 0) {
        verticalMovement = Math.ceil(Math.random() * 5);
    } else if ( ( currentTop + floatingDiv.offsetHeight ) > screenHeight ) {
        verticalMovement = -(Math.ceil(Math.random() * 5));
    }

    if (currentLeft < 0) {
        horizontalMovement = Math.ceil(Math.random() * 5);
    } else if ( ( currentLeft + floatingDiv.offsetWidth ) > screenWidth ) {
        horizontalMovement = -(Math.ceil(Math.random() * 5));
    }

    floatingDiv.style.left = currentLeft + horizontalMovement + "px";
    floatingDiv.style.top = currentTop + verticalMovement + "px";
}
</script>

</head>
<body onload="startTimer()">
```

```
<div id="floatingDiv"></div>
</body>
</html>
```

Let's see how the page works. When the page is loaded, the window's `onload` event handler calls the function `startTimer()`. This starts a timer going at regular 10-millisecond intervals. Each time the timer fires, it calls the function `moveDiv()`, which moves a `<div/>` element about the page.

The style sheet in this page consists of only one rule, which is for the `floatingDiv <div/>` element.

```
#floatingDiv {
  position: absolute;
  left: 0px;
  top: 0px;
  width: 50px;
  height: 50px;
  background-color: navy;
}
```

This rule sets the element to be positioned absolutely at 0, 0. It also sets the height and width to 50 pixels, and gives the element a navy background color.

Now let's look at the code. At the top of the script block, five global, page-level variables are defined. These variables are accessible from any JavaScript function used in the web page.

```
var floatingDiv;
var screenWidth;
var screenHeight;

var horizontalMovement = Math.ceil(Math.random() * 5);
var verticalMovement = Math.ceil(Math.random() * 5);
```

The `floatingDiv` variable will reference the `<div/>` element. The `screenHeight` and `screenWidth` variables will contain the height and width of the browser's viewport, respectively. These values are used to determine if the element has reached the edge of the viewport. The final two global variables are `horizontalMovement` and `verticalMovement`. These contain random numbers and are between 1 and 5. These hold the amount that the `<div/>` element should be moved each time the timer calls `moveDiv()`.

The first function, `startTimer()`, is called when the page loads. Its job is to populate the `floatingDiv`, `screenHeight`, and `screenWidth` variables with their values. It also starts the animation.

```
function startTimer() {
  floatingDiv = document.getElementById("floatingDiv");
  screenWidth = document.body.clientWidth;
  screenHeight = document.body.clientHeight;

  window.setInterval("moveDiv()", 10);
}
```

Appendix A: Exercise Solutions

The `floatingDiv` variable gets its value by using the `document.getElementById()` method to retrieve the `<div/>` element in the page's body. By using the `clientWidth` and `clientHeight` properties of the `document.body` object, the `screenWidth` and `screenHeight` variables get their values. The final line of this function uses `window.setInterval()` to repeatedly call `moveDiv()` every 10 milliseconds.

The real workhorse of this example is the `moveDiv()` function, which moves the element around on the page. The first task is to find the element's current top and left positions.

```
function moveDiv() {
    var currentLeft = floatingDiv.offsetLeft;
    var currentTop  = floatingDiv.offsetTop;

    //more code here
}
```

You do this by using the `offsetLeft` and `offsetTop` properties of `floatingDiv`. Next, the function needs to determine where the element should be moved. This is done in two sections.

The first section decides whether or not the element has reached the top or bottom of the page.

```
function moveDiv() {
    var currentLeft = floatingDiv.offsetLeft;
    var currentTop  = floatingDiv.offsetTop;

    if (currentTop < 0) {
        verticalMovement = Math.ceil(Math.random() * 5);
    } else if ( ( currentTop + floatingDiv.offsetHeight ) > screenHeight ) {
        verticalMovement = -(Math.ceil(Math.random() * 5));
    }

    //more code here
}
```

If the top of the element has reached the top of the page, then the `<div/>` needs to move downward. Therefore, the `verticalMovement` variable is set to a random positive number between 1 and 5 that moves the element toward the bottom of the page. If the bottom of the page has been reached by the element's bottom, then the element needs to start moving back toward the top. Therefore, `verticalMovement` is assigned a negative random number between 1 and 5.

The second section determines whether or not the element has reached the right or left edge of the browser's viewport.

```
function moveDiv() {
    var currentLeft = floatingDiv.offsetLeft;
    var currentTop  = floatingDiv.offsetTop;

    if (currentTop < 0) {
        verticalMovement = Math.ceil(Math.random() * 5);
    } else if ( ( currentTop + floatingDiv.offsetHeight ) > screenHeight ) {
        verticalMovement = -(Math.ceil(Math.random() * 5));
    }
}
```

```

    }

    if (currentLeft < 0) {
        horizontalMovement = Math.ceil(Math.random() * 5);
    } else if ( ( currentLeft + floatingDiv.offsetWidth ) > screenWidth ) {
        horizontalMovement = -(Math.ceil(Math.random() * 5));
    }

    //more code here
}

```

This new code follows the same principles as the top/bottom code. First, it checks to see whether the element has reached the left edge of the viewport. If so, `horizontalMovement` is assigned a random positive number to move the element to the right. When the right edge of the element reaches the right edge of the page, `horizontalMovement` is set to a random negative number, which moves the element back to the left.

The final step is to move the element to its new location.

```

function moveDiv() {
    var currentLeft = floatingDiv.offsetLeft;
    var currentTop  = floatingDiv.offsetTop;

    if (currentTop < 0) {
        verticalMovement = Math.ceil(Math.random() * 5);
    } else if ( ( currentTop + floatingDiv.offsetHeight ) > screenHeight ) {
        verticalMovement = -(Math.ceil(Math.random() * 5));
    }

    if (currentLeft < 0) {
        horizontalMovement = Math.ceil(Math.random() * 5);
    } else if ( ( currentLeft + floatingDiv.offsetWidth ) > screenWidth ) {
        horizontalMovement = -(Math.ceil(Math.random() * 5));
    }

    floatingDiv.style.left = currentLeft + horizontalMovement + "px";
    floatingDiv.style.top  = currentTop  + verticalMovement + "px";
}

```

This code uses the `style` object and `left` and `top` properties to set the new left and top positions for the `<div/>` element. The new `left` position is set to the element's current left plus the number contained in `horizontalMovement`, and the new `top` position is set to the element's current top plus `verticalMovement`. At the end of both statements, the string `px` is appended to the value, making sure that the browser will position the element correctly.

Chapter 13

In this chapter you looked at the DOM and how the standard method for accessing objects on the HTML document can be applied in JavaScript and used to create web pages that will work in both major browsers.

Appendix A: Exercise Solutions

Question 1

Here's some HTML code that creates a web page. Re-create this page, using JavaScript to generate the HTML using only DOM objects, properties, and methods. Test your code in IE, Firefox, Opera, and Safari (if you have it) to make sure it works in them.

Hint: Comment each line as you write it to keep track of where you are in the tree structure, and create a new variable for every element on the page (for example, not just one for each of the TD cells, but nine variables).

```
<html>
<head>
</head>
<body>
  <table>
    <thead>
      <tr>
        <td>Car</td>
        <td>Top Speed</td>
        <td>Price</td>
      </tr>
    </thead>
    <tbody>
      <tr>
        <td>Chevrolet</td>
        <td>120mph</td>
        <td>$10,000</td>
      </tr>
      <tr>
        <td>Pontiac</td>
        <td>140mph</td>
        <td>$20,000</td>
      </tr>
    </tbody>
  </table>
</body>
</html>
```

Solution

It seems a rather daunting example, but rather than being difficult, it is just a conjunction of two areas, one building a tree structure and the other navigating the tree structure. You start by navigating to the `<body/>` element and creating a `<table/>` element. Now you can navigate to the new `<table/>` element you've created and create a new `<thead/>` element and carry on from there. It's a lengthy and repetitious process, so that's why it's a good idea to comment your code to keep track of where you are.

```
<html>
<head>
</head>
<body>
<script language="JavaScript">
var TableElem = document.createElement("table")
```



```

var THElem = document.createElement("thead")
var TRElem1 = document.createElement("TR")
var TRElem2 = document.createElement("TR")
var TRElem3 = document.createElement("TR")
var TDElem1 = document.createElement("TD")
var TDElem2 = document.createElement("TD")
var TDElem3 = document.createElement("TD")
var TDElem4 = document.createElement("TD")
var TDElem5 = document.createElement("TD")
var TDElem6 = document.createElement("TD")
var TDElem7 = document.createElement("TD")
var TDElem8 = document.createElement("TD")
var TDElem9 = document.createElement("TD")
var TBODYElem = document.createElement("tbody")
var TextNodeA1 = document.createTextNode("Car")
var TextNodeA2 = document.createTextNode("Top Speed")
var TextNodeA3 = document.createTextNode("Price")
var TextNodeB1 = document.createTextNode("Chevrolet")
var TextNodeB2 = document.createTextNode("120mph")
var TextNodeB3 = document.createTextNode("$10,000")
var TextNodeC1 = document.createTextNode("Pontiac")
var TextNodeC2 = document.createTextNode("140mph")
var TextNodeC3 = document.createTextNode("$14,000")

docNavigate = document.documentElement; //Starts with HTML document
docNavigate = docNavigate.lastChild;    //Moves to body element
docNavigate.appendChild(TableElem);    //Adds the table element
docNavigate = docNavigate.lastChild;    //Moves to the table element
docNavigate.appendChild(THElem);        //Adds the thead element
docNavigate = docNavigate.firstChild;    //Moves to the thead element
docNavigate.appendChild(TRElem1);        //Adds the TR element
docNavigate = docNavigate.firstChild;    //Moves the TR element
docNavigate.appendChild(TDElem1);        //Adds the first TD element in the
// heading
docNavigate.appendChild(TDElem2);        //Adds the second TD element in the
// heading
docNavigate.appendChild(TDElem3);        //Adds the third TD element in the
// heading

docNavigate = docNavigate.firstChild;    //Moves to the first TD element
docNavigate.appendChild(TextNodeA1);    //Adds the second text node
docNavigate = docNavigate.nextSibling;    //Moves to the next TD element
docNavigate.appendChild(TextNodeA2);    //Adds the second text node
docNavigate = docNavigate.nextSibling;    //Moves to the next TD element
docNavigate.appendChild(TextNodeA3);    //Adds the third text node

docNavigate = docNavigate.parentNode;    //Moves back to the TR element
docNavigate = docNavigate.parentNode;    //Moves back to the thead element
docNavigate = docNavigate.parentNode;    //Moves back to the table element
docNavigate.appendChild(TBODYElem);    //Adds the tbody element
docNavigate = docNavigate.lastChild;    //Moves to the tbody element
docNavigate.appendChild(TRElem2);        //Adds the second TR element
docNavigate = docNavigate.lastChild;    //Moves to the second TR element
docNavigate.appendChild(TDElem4);        //Adds the TD element
docNavigate.appendChild(TDElem5);        //Adds the TD element
docNavigate.appendChild(TDElem6);        //Adds the TD element

```

Appendix A: Exercise Solutions

```
docNavigate = docNavigate.firstChild;    //Moves to the first TD element
docNavigate.appendChild(TextNodeB1);    //Adds the first text node
docNavigate = docNavigate.nextSibling;    //Moves to the next TD element
docNavigate.appendChild(TextNodeB2);    //Adds the second text node
docNavigate = docNavigate.nextSibling;    //Moves to the next TD element
docNavigate.appendChild(TextNodeB3);    //Adds the third text node
docNavigate = docNavigate.parentNode;    //Moves back to the TR element
docNavigate = docNavigate.parentNode;    //Moves back to the tbody element
docNavigate.appendChild(TRElem3);        //Adds the TR element
docNavigate = docNavigate.lastChild;     //Moves to the TR element
docNavigate.appendChild(TDElem7);        //Adds the TD element
docNavigate.appendChild(TDElem8);        //Adds the TD element
docNavigate.appendChild(TDElem9);        //Adds the TD element
docNavigate = docNavigate.firstChild;    //Moves to the TD element
docNavigate.appendChild(TextNodeC1);    //Adds the first text node
docNavigate = docNavigate.nextSibling;    //Moves to the next TD element
docNavigate.appendChild(TextNodeC2);    //Adds the second text node
docNavigate = docNavigate.nextSibling;    //Moves to the next TD element
docNavigate.appendChild(TextNodeC3);    //Adds the third text node
</script>
</body>
</html>
```

Question 2

Augment your DOM web page so that the table has a border and only the headings of the table (that is, not the column headings) are center-aligned. Again, test your code in both IE, Firefox, Opera, and Safari (if you have it).

Hint: Add any extra code to the end of the script code you have already written.

Solution

Add these lines to the bottom of the script code to add a border:

```
docAttr = document.getElementsByTagName("table").item(0);
docAttr.setAttribute("border", "1");
```

Add these lines to the bottom of the script code to center-align headings:

```
docNewAttr = document.getElementsByTagName("thead").item(0);
docNewAttr.setAttribute("align", "center");
```

Chapter 14

In this chapter you took a very brief look at XML, created your first XML document (a DTD), and then formatted the XML document using CSS and XSL.

Question 1

Create an XML document that logically orders the following data for a school:

- ☐ Child's name
- ☐ Child's age
- ☐ Class the child is in

Use the following data:

Bibby Jones	13	1B
Beci Smith	12	1B
Jack Wilson	14	2C

Solution

```
<?xml version="1.0" encoding="iso-8859-1"?>

<school>
  <child>
    <name>Bibby Jones</name>
    <age>13</age>
    <class>1B</class>
  </child>
  <child>
    <name>Beci Smith</name>
    <age>12</age>
    <class>1B</class>
  </child>
  <child>
    <name>Jack Wilson</name>
    <age>14</age>
    <class>2C</class>
  </child>
</school>
```

Save this file as `school.xml`.

Question 2

Using JavaScript, load the information from the XML file into a page and display it when the user clicks a link.

Solution

```
<html>
<head>
  <title>Chapter 14 Question 2</title>
```

Appendix A: Exercise Solutions

```
<script type="text/javascript" language="javascript">
function createDocument()
{
    //Temporary DOM object.
    var xmlDoc;

    //Create the DOM object for IE
    if (window.ActiveXObject)
    {
        var versions =
        [
            "Msxml2.DOMDocument.6.0",
            "Msxml2.DOMDocument.3.0"
        ];

        for (var i = 0; i < versions.length; i++)
        {
            try
            {
                xmlDoc = new ActiveXObject(versions[i]);
                return xmlDoc;
            }
            catch (error)
            {
                //do nothing here
            }
        }
    }
    //Create the DOM for Firefox and Opera
    else if (document.implementation && document.implementation.createDocument)
    {
        xmlDoc = document.implementation.createDocument("", "", null);
        return xmlDoc;
    }

    return null;
}

var xmlDoc = createDocument();
xmlDoc.load("school.xml");

function writeTableOfSchoolChildren()
{
    var xmlNode = xmlDoc.getElementsByTagName('child');
    var newTableElement = document.createElement('table');
    newTableElement.setAttribute('cellPadding', 5);

    var tempElement = document.createElement('tbody');
    newTableElement.appendChild(tempElement);
    var tableRow = document.createElement('TR');

    tempElement.appendChild(tableRow );

    for (var i = 0; i < xmlNode.length; i++)
    {
```

```

        var tableRow = document.createElement('TR');
        for (var iRow= 0; iRow < xmlNode[i].childNodes.length; iRow++)
        {
            if (xmlNode[i].childNodes[iRow].nodeType != 1)
            {
                continue;
            }
            var tdElement = document.createElement('TD');
            var textData = document.createTextNode(xmlNode[i].childNodes[iRow]
.firstChild.nodeValue);
            tdElement.appendChild(textData);
            tableRow.appendChild(tdElement);
        }

        tempElement.appendChild(tableRow);
    }

    document.getElementById('displaySchoolInfo').appendChild(newTableElement);
}
</script>

</head>

<body>

<p>
    <a href="javascript: writeTableOfSchoolChildren()">
        Show Table of Children At The School
    </a>
</p>
<p id="displaySchoolInfo"></p>

</body>
</html>

```

Chapter 15

In this chapter you saw how you could extend the functionality of the browser by using plug-ins and ActiveX controls, and how you could use these plug-ins within web pages.

Question 1

Using the RealPlayer plug-in/ActiveX control, create a page with three links, so that when the mouse pointer rolls over any of them a sound is played. The page should work in Firefox and IE. However, any other browsers should be able to view the page and roll over the links without errors appearing.


```

        <a onmouseover="play('Explosion.ra')" onmouseout="document.real1.DoStop();"
href="#">
            Kaboom!
        </a>
</body>
</html>

```

Save this as `ch15_q1.htm`.

This solution is based on the `RealPlayer` example in the chapter. Note that the three sound files, `Evil_Laugh.ra`, `Whoosh.ra`, and `Explosion.ra`, can be found in the code download for this book.

Verify that the user has the ability to play `RealAudio` files in the window `onload` event handler, which calls the `window.onload()` function.

Just as the IE and Firefox support for plug-ins is different, so therefore are the means of checking for plug-ins. For Firefox, go through the `navigator` object's `plugins` array and check each installed plug-in for the name `RealPlayer`; if it's found, you know the user has the `RealAudio` player installed.

With IE, simply use the `real1` ActiveX control's `readyState` property to see if it's installed and initialized correctly.

To play the sounds, a function called `play()` is defined whose parameter is the name of the `.ra` (or `.rm`) sound file to be played.

```

function play(fileName)
{
    document.real1.SetSource(fileName);
    document.real1.DoPlay();
}

```

The function makes use of the `RealAudio` player's `setSource()` method to set the sound file to be played and the `DoPlay()` method to actually start playing the clip. You have used different sounds for each link by simply specifying a different file name each time as the parameter for the `play()` function.

Use the `onmouseover` and `onmouseout` event handlers to start playing the sound when the mouse pointer is over the link and to stop it when the mouse pointer moves out of the link, respectively. The `mouseout` event starts playing the audio clip by calling the `play()` function, and the `mouseout` event stops playing it by calling the `RealPlayer`'s `DoStop()` method.

```

<a onmouseover="play('audiosig.ra')" onmouseout="document.real1.DoStop()" href="#">
    Evil Laugh
</a>

```

Chapter 16

This chapter introduced you to the concept of remote scripting. You wrote a JavaScript class to easily perform asynchronous HTTP requests and created two forms that used Ajax to validate their fields.

Appendix A: Exercise Solutions

Question 1

Extend the `HttpRequest` class to include synchronous requests in addition to the asynchronous requests the class already makes.

Solution

```
function HttpRequest(sUrl, fpCallback)
{
    this.url = sUrl;
    this.callBack = fpCallback;
    this.async = true;
    this.request = this.createXmlHttpRequest();
}

HttpRequest.prototype.createXmlHttpRequest = function ()
{
    if (window.XMLHttpRequest)
    {
        var oHttp = new XMLHttpRequest();
        return oHttp;
    }
    else if (window.ActiveXObject)
    {
        var versions =
        [
            "MSXML2.XmlHttp.6.0",
            "MSXML2.XmlHttp.3.0"
        ];

        for (var i = 0; i < versions.length; i++)
        {
            try
            {
                oHttp = new ActiveXObject(versions[i]);
                return oHttp;
            }
            catch (error)
            {
                //do nothing here
            }
        }
    }
    return null;
}

HttpRequest.prototype.send = function()
{
    this.request.open("GET", this.url, this.async);

    if (this.async)
```



```

    {
        var tempRequest = this.request;
        var fpCallback = this.callBack;

        function request_readystatechange()
        {
            if (tempRequest.readyState == 4)
            {
                if (tempRequest.status == 200)
                {
                    fpCallback(tempRequest.responseText);
                }
                else
                {
                    alert("An error occurred while attempting to contact the
server.");
                }
            }
        }

        this.request.onreadystatechange = request_readystatechange;
    }

    this.request.send(null);

    if (!this.async)
    {
        this.callBack(this.request.responseText);
    }
}

```

It's possible to add synchronous communication to your `HttpRequest` class in a variety of ways. The approach in this solution refactors the code to accommodate a new property called `async`, which contains either `true` or `false`. If it contains `true`, then the class uses asynchronous communication to retrieve the file, and if it contains `false`, the class uses synchronous communication. In short, this property resembles an XML DOM's `async` property for determining how an XML document is loaded.

The first change made to the class is in the constructor itself. The original constructor initializes and readies the `XMLHttpRequest` (XHR) object to send data. This will not do for this new version, however. Instead, the constructor merely initializes all the properties.

```

function HttpRequest(sUrl, fpCallback)
{
    this.url = sUrl;
    this.callBack = fpCallback;
    this.async = true;
    this.request = this.createXmlHttpRequest();
}

```

There are three new properties to the class. The first, `url`, contains the URL that the XHR object should attempt to request from the server. The `callBack` property contains a reference to the callback function, and the `async` property determines the type of communication the XHR uses. Setting `async` to `true` in the constructor gives the property a default value. Therefore, you can send the request without setting the property externally.

Appendix A: Exercise Solutions

The new constructor and class properties are actually desirable, as they enable you to reuse the same `HttpRequest` object for multiple requests. If you wanted to make a request to a different URL, all you would need to do is assign the `url` property a new value. The same can be said for the callback function as well.

The `createXmlHttpRequest()` method remains untouched. This is a helper method and doesn't really have anything to do with sending the request. For completeness, however, here is its definition:

```
HttpRequest.prototype.createXmlHttpRequest = function ()
{
    if (window.XMLHttpRequest)
    {
        var oHttp = new XMLHttpRequest();
        return oHttp;
    }
    else if (window.ActiveXObject)
    {
        var versions =
        [
            "MSXML2.XmlHttp.6.0",
            "MSXML2.XmlHttp.3.0"
        ];

        for (var i = 0; i < versions.length; i++)
        {
            try
            {
                oHttp = new ActiveXObject(versions[i]);
                return oHttp;
            }
            catch (error)
            {
                //do nothing here
            }
        }
    }
    return null;
}
```

The majority of changes to the class are in the `send()` method. It is here that the class decides whether to use asynchronous or synchronous communication. Both types of communication have very little in common when it comes to making a request; asynchronous communication uses the `onreadystatechange` event handler, and synchronous communication allows access to the XHR object's properties when the request is complete. Therefore, code branching is required.

```
HttpRequest.prototype.send = function()
{
    this.request.open("GET", this.url, this.async);

    if (this.async)
    {
```

```

        //more code here
    }

    this.request.send(null);

    if (!this.async)
    {
        //more code here
    }
}

```

The first line of this method uses the `open()` method of the XHR object. The `async` property is used as the final parameter of the method. This determines whether or not the XHR object uses asynchronous communication. Next comes an `if` statement, which tests to see if `this.async` is `true`; if it is, the asynchronous code will be placed in this `if` block. Next, the XHR object's `send()` method is called, sending the request to the server. The final `if` statement checks to see whether `this.async` is `false`. If it is, synchronous code is placed here to execute.

```

HttpRequest.prototype.send = function()
{
    this.request.open("GET", this.url, this.async);

    if (this.async)
    {
        var tempRequest = this.request;
        var fpCallback = this.callBack;

        function request_readystatechange()
        {
            if (tempRequest.readyState == 4)
            {
                if (tempRequest.status == 200)
                {
                    fpCallback(tempRequest.responseText);
                }
                else
                {
                    alert("An error occurred while attempting to contact the
server.");
                }
            }
        }

        this.request.onreadystatechange = request_readystatechange;
    }

    this.request.send(null);

    if (!this.async)
    {
        this.callBack(this.request.responseText);
    }
}

```

Appendix A: Exercise Solutions

This new code finishes off the method. Let's start with the first `if` block. A new variable called `fpCallback` is assigned the value of `this.callBack`. This is done for the same reasons as with the `tempRequest` variable—scoping issues—as `this` points to the `request_readystatechange()` function instead of the `HttpRequest` object. Other than this change, the asynchronous code remains the same. The `request_readystatechange()` function handles the `readystatechange` event and calls the callback function when the request is successful.

The second `if` block is much simpler. Because this code executes only if synchronous communication is desired, all you have to do is call the callback function and pass the XHR's `responseText` property.

Using this newly refactored class is quite simple. The following code makes an asynchronous request for a fictitious text file called `test.txt`.

```
function request_callback(sResponseText)
{
    alert(sResponseText);
}

var oHttp = new HttpRequest("test.txt", request_callback);

oHttp.send();
```

Nothing has really changed for asynchronous requests. This is the exact same code used earlier in the chapter. If you want to use synchronous communication, simply set `async` to `false`, like this:

```
function request_callback(sResponseText)
{
    alert(sResponseText);
}

var oHttp = new HttpRequest("test.txt", request_callback);

oHttp.async = false;

oHttp.send();
```

You now have a class that requests information in both asynchronous and synchronous communication!

Question 2

It was mentioned earlier in the chapter that the smart forms could be modified to not use hyperlinks. Change the form that uses the `HttpRequest` class so that the user name and e-mail fields are checked at form submission. The only time you need to alert the user is when the user name or e-mail is taken.

Solution

```
<html>
<head>
    <title>Form Field Validation</title>
    <style type="text/css">
```

```
.fieldname
{
    text-align: right;
}

.submit
{
    text-align: right;
}
</style>
<script type="text/javascript" src="HttpRequest.js"></script>
<script type="text/javascript">
    var isUsernameTaken;
    var isEmailTaken;

    function checkUsername_callBack(sResponseText)
    {
        if (sResponseText == "available")
        {
            isUsernameTaken = false;
        }
        else
        {
            isUsernameTaken = true;
        }
    }

    function checkEmail_callBack(sResponseText)
    {
        if (sResponseText == "available")
        {
            isEmailTaken = false;
        }
        else
        {
            isEmailTaken = true;
        }
    }

    function form_submit()
    {
        var request = new HttpRequest();
        request.async = false;

        //First check the username
        var userValue = document.getElementById("username").value;

        if (userValue == "")
        {
            alert("Please enter a user name to check!");
            return false;
        }

        request.url = "formvalidator.php?username=" + userValue;
```

Appendix A: Exercise Solutions

```
request.callBack = checkUsername_callBack;
request.send();

if (isUsernameTaken)
{
    alert("The username " + userValue + " is not available!");
    return false;
}

//Now check the email
var emailValue = document.getElementById("email").value;

if (emailValue == "")
{
    alert("Please enter an email address to check!");
    return false;
}

request.url = "formvalidator.php?email=" + emailValue;
request.callBack = checkEmail_callBack;
request.send();

if (isEmailTaken)
{
    alert("I'm sorry, but " + emailValue + " is in use by " +
        "another user.");
    return false;
}

//If the code's made it this far, everything's good
return true;
}
</script>
</head>
<body>
    <form onsubmit="return form_submit()">
        <table>
            <tr>
                <td class="fieldname">
                    Username:
                </td>
                <td>
                    <input type="text" id="username" />
                </td>
            </tr>
            <tr>
                <td class="fieldname">
                    Email:
                </td>
                <td>
                    <input type="text" id="email" />
                </td>
            </tr>
            <tr>
                <td colspan="2">
                    <input type="button" value="Submit" />
                </td>
            </tr>
        </table>
    </form>
</body>
</html>
```

```

        <td class="fieldname">
            Password:
        </td>
        <td>
            <input type="text" id="password" />
        </td>

    </tr>
    <tr>
        <td class="fieldname">
            Verify Password:
        </td>
        <td>
            <input type="text" id="password2" />
        </td>

    </tr>
    <tr>
        <td colspan="2" class="submit">
            <input type="submit" value="Submit" />
        </td>
    </tr>
</table>
</form>
</body>
</html>

```

Let's begin with the HTML. The links were removed, as well as the third column of the table. The key difference in this new HTML is the `onsubmit` event handler in the opening `<form>` tag. Ideally, the form should submit its data only when the form fields have been validated. Therefore, the `onsubmit` event handler is set to return `form_submit()`. The `form_submit()` function returns either `true` or `false`, making the browser submit the form's data if everything is okay and not submit if a field is not validated.

The JavaScript code holds the most changes, so let's get started there. In this new implementation, two global variables, called `isUsernameTaken` and `isEmailTaken`, are declared. These variables hold `true` or `false` values: `true` if the user name or e-mail is taken, and `false` if it is not.

```

var isUsernameTaken;
var isEmailTaken;

function checkUsername_callBack(sResponseText)
{
    if (sResponseText == "available")
    {
        isUsernameTaken = false;
    }
    else
    {
        isUsernameTaken = true;
    }
}

function checkEmail_callBack(sResponseText)

```

Appendix A: Exercise Solutions

```
{
    if (sResponseText == "available")
    {
        isEmailTaken = false;
    }
    else
    {
        isEmailTaken = true;
    }
}
```

The first two functions, `checkUsername_callBack()` and `checkEmail_callBack()`, are somewhat similar to their original versions. Instead of alerting information to the user, however, they simply assign the `isUsernameTaken` and `isEmailTaken` variables their values.

The function that performs most of the work is `form_submit()`. It is responsible for making the requests to the server and determines if the data in the form fields are ready for submission.

```
function form_submit()
{
    var request = new HttpRequest();
    request.async = false;

    //more code here
}
```

This code creates the `HttpRequest` object and sets it to synchronous communication. There are times when synchronous communication is appropriate to use, and during form validation is one of those times. Validating fields in a form is a sequential process, and its submission depends upon the outcome of the `onsubmit` event handler. Using synchronous communication forces the function to wait for information to be retrieved from the server before attempting to validate the field. If you used asynchronous communication, `form_submit()` would attempt to validate the `Username` and `Email` fields before it had the appropriate information for each field. Also note that the `HttpRequest` constructor received no arguments. This is because you can explicitly set the `url` and `callback` properties with the new version.

The first field to check is the `Username` field.

```
function form_submit()
{
    var request = new HttpRequest();
    request.async = false;

    //First check the username
    var userValue = document.getElementById("username").value;

    if (userValue == "")
    {
        alert("Please enter a user name to check!");
        return false;
    }

    request.url = "formvalidator.php?username=" + userValue;
```



```

    request.callBack = checkUsername_callBack;
    request.send();

    if (isUsernameTaken)
    {
        alert("The username " + userValue + " is not available!");
        return false;
    }

    //more code here
}

```

This code retrieves the value of the Username field and checks to see whether any information was entered. If none was entered, a message is alerted to the user informing her to enter data. If the user entered information in the Username field, then code execution continues. The url and callBack properties are assigned their values and the request is sent to the server. If it turns out that the user's desired user name is taken, an alert box tells her so. Otherwise, the code continues to execute and checks the e-mail information:

```

function form_submit()
{
    var request = new HttpRequest();
    request.async = false;

    //First check the username
    var userValue = document.getElementById("username").value;

    if (userValue == "")
    {
        alert("Please enter a user name to check!");
        return false;
    }

    request.url = "formvalidator.php?username=" + userValue;
    request.callBack = checkUsername_callBack;
    request.send();

    if (isUsernameTaken)
    {
        alert("The username " + userValue + " is not available!");
        return false;
    }

    //Now check the email
    var emailValue = document.getElementById("email").value;

    if (emailValue == "")
    {
        alert("Please enter an email address to check!");
        return false;
    }

    request.url = "formvalidator.php?email=" + emailValue;
    request.callBack = checkEmail_callBack;
}

```

Appendix A: Exercise Solutions

```
request.send();

if (isEmailTaken)
{
    alert("I'm sorry, but " + emailValue + " is in use by another user.");
    return false;
}

//If the code's made it this far, everything's good
return true;
}
```

The e-mail-checking code goes through the same process that was used to check the user name. The value of the `Email` field is retrieved and checked to determine whether the user typed anything into the text box. Then that value is used to make another request to the server. Notice again that the `url` and `callback` properties are explicitly set. If `isEmailTaken` is true, an alert box shows the user that another user has taken the e-mail address and the function returns `false`. If the address is available, the function returns `true`, thus making the browser submit the form.

Index

SYMBOLS AND NUMERICS

& (ampersand), XML (Extensible Markup Language) character data, 555–556

&& (AND), logical operator, 70–72

' (apostrophe), XML (Extensible Markup Language) character data, 555–556

***** (asterisk), repetition character, 304–305

&#x26;, entity reference, 555–556

', entity reference, 555–556

>, entity reference, 555–556

<, entity reference, 555–556

** **, space symbol, 23

<A> tag, event handlers and the, 170–175

**** (backslash), escape character, 23

\' (backslash apostrophe), escape sequence, 23

\" (backslash quote), escape sequence, 23

**** (double backslash), escape sequence, 23

\b

escape sequence, 23

position character, 305–308

\\b, regular expression, 324–329

^ (caret), position character, 305–308

[^...] (caret and periods in brackets), character class, 300–304

© (copyright symbol), inserting in a string, 23

{ } (curly braces), marking out a block of code with, 67

— (decrement), operators, 37–39

\d, character class, 300–304, 300–304

\$ (dollar sign), position character, 305–308

== (double equal to sign), operator symbol, 65

!= (exclamation point equal to), operator symbol, 65

\f, escape sequence, 23

> (greater than)

operator symbol, 65

XML (Extensible Markup Language) character data, 555–556

>= (greater than or equal to), operator symbol, 65

++ (increment), operators, 37–39

**** tag, exercise questions, 186, 671

< (less than)

operator symbol, 65

XML (Extensible Markup Language) character data, 555–556

<= (less than or equal to), operator symbol, 65

! (NOT), logical operator, 70–72

\n, escape sequence, 23

|| (OR), logical operator, 70–72

. (period), character class, 300–304

[...] (periods in brackets), character class, 300–304

+ (plus sign), repetition character, 304–305

(pound sign), CSS (Cascading Style Sheets) and the, 471

? (question mark), repetition character

? (question mark), repetition character, 304–305

“ (quote marks)

escape sequence, 23

strings and, 22

\r, **escape sequence**, 23

\S, **character class**, 300–304

;(semicolon)

requirements for the, 660

using the, 8

\t, **escape sequence**, 23

| (**vertical bar**), **alternation character**, 309–310

\W, **character class**, 300–304

\xNN, **escape sequence**, 23

A

<A> **tag**, **event handlers and the**, 170–175

abs() **method**, **overview**, 122

absolute positioning, **defined**, 467

ActiveX controls

adding to a page in Internet Explorer, 602–605

browser differences with, 614–615

changes to Internet Explorer 6 Service Pack 1b and, 617

creating a page with, 713–715

defined, 601–602

exercise questions, 618, 713–715

exercise solutions, 713–715

general use of, 607–617

installing, 606–607

plug-ins versus, 601–602

problems with, 614–617, 652–653

summary, 617–618

using in Internet Explorer 5/6, 622–624

version differences, 616–617

ActiveX objects, overview, 577–579

ActiveXObject() **object**, **using the**, 577–579

add() **method**, **options[] array**, 220–221

addBookToBasket(), **function**, 683

adding

an ActiveX control to the page in Internet Explorer, 602–605

borders to a Web page, 710

buttons to a pop-up window, 681–686

data to an XML (Extensible Markup Language)

document, 561–566

elements to a document, 516–518

features to a window, 262–264

HTML to a new window, 261–262

links to a pop-up window, 681–686

new options, 216–220

new options with Internet Explorer, 220–221

a plug-in to a page in Firefox, 596–598

style to the DHTML (Dynamic HTML) toolbar, 529–532

style to HTML pages, 459–470

addPage(), **function**, 245–247

addToBasket(), **function**, 685–686

Ajax: A New Approach to Web Applications (Garrett), 621

Ajax (Asynchronous JavaScript + XML)

delays, 653–654

remote scripting and, 621

Same-Origin Policy, 652

summary, 655

XMLHttpRequest object, 622

alert()

function, 11–12

method in HTML Forms, 214

align, **attribute**, 521

alphanumeric characters, **checking for**, 301–304

alternation characters, | (**vertical bar**), 309–310

altKey **property**, **mouseEvent object**, 522–526

& (ampersand), **XML (Extensible Markup Language)**, **character data**, 555–556

& , **entity reference**, 555–556

AND (&&), **logical operator**, 71

animation, **requisites of**, 483

answerCorrect() **function**

Answer Radio Buttons, 229–233

Trivia Quiz, 99–100, 287, 323–329

Apache HTTP Server, **Web site**, 636

' (**apostrophe**), **XML (Extensible Markup Language)**, **character data**, 555–556

' , **entity reference**, 555–556

appearances, **changing**, 470–478

appendChild(newNode) **method**, **Node object**, 516–518

Apple Safari. See Safari`apply_templates`, **elements**, 572–574`appName` **property**, `navigator` **object** and the, 179–184**array**

alphabetizing an, 134–135

converting into a single string, 133–134

copying part of an, 132–133

creating an, 50–54

defined, 18, 49

`document.images`, 672

finding the number of elements in an, 130–131

`for...in` loop and, 87–88`forms[]`, 188–190`frames[]`, 249, 288`images[]`, 167–169

joining an, 131–132

`links[]`, 169, 668`mimeType`s, 600–601`navigator.plugins`, 598`options[]`, 220–221

putting in numerical order, 134–135

putting in reverse order, 135–137

sorting an, 665

sorting example, 136–137

using an, 151–156

value attribute storage versus, 677

Array object`concat()` method, 131–132

creating an, 107–109

HTML Forms, 213

`join()` method, 133–134`length` property, 130–131

overview, 107, 130, 156

`reverse()` method, 135–137`slice()` method, 132–133`sort()` method, 134–135`AskQuestion.htm`, **creating for the trivia quiz**, 276–278, 283–284**assignment operator**

comparison operator versus, 66

defined, 26, 36

*** (asterisk)**, repetition character, 304–305`async` **property**, using the, 579–580, 717–726**asynchronous**

communication, 720

requests, 627–629

Asynchronous JavaScript + XML (Ajax)

delays, 653–654

remote scripting and, 621

Same-Origin Policy, 652

summary, 655

`XMLHttpRequest` object, 622`Attr` **object**, high-level DOM object, 502**attribute**`align`, 521`bgColor`, 7`CHECKED`, 208`class`, 475–478`classid`, 602–605`codebase`, 602–605`COLS`, 204–207

creating an, 519–521

event handler as an, 170–172

`font-style`, 461`g`, 299

getting an, 518–521

`i`, 299`ID`, 240–241`m`, 299`name`, 233`onclick`, 170–172`pluginspage`, 598–601`rows`, 204–207, 240–241

setting an, 518–521

`src`, 151`style`, 461`target`, 188–189`type`, 7`value`, 677`wrap`, 204–207**B**

\ (backslash), escape character, 23

\’ (backslash apostrophe), escape sequence, 23

\” (backslash quote), escape sequence, 23

\\ (double backslash), escape sequence

\\ (double backslash), escape sequence, 23

\\b

escape sequence, 23

position character, 305–308

\\b, **regular expression**, 324–329

\\B, **position character**, 305–308

back button, browser, 653

back() **method**, **history object**, 163

background color

changing, 463–464

example for changing in the browser, 7–8

background-color, **property**, 463–464

backslash, escape sequence, 23

backspace, escape sequence, 23

backward compatible, defined, 13

base DOM objects, overview, 501

Berners-Lee, Tim, HTML creator, 490–492

bgColor, **attribute**, 7

binary numbers, defined, 2

bitwise operators, defined, 70

blocking, pop-ups, 257

blur **event**

Firefox/Netscape and the, 203

troubleshooting the, 203

blur() **method**

<form> elements, 193–194

summary, 233–234

BOM (Browser Object Model)

<A> tag, 170–175

browser properties checking, 176–178

command window and the, 383

document object, 165–169

DOM (Document Object Model) versus, 497

exercise questions, 667–672

exercise solutions, 667–672

history object, 163

images[] Array, 167–169

links[] Array, 169

location object, 163–164

navigator object, 164, 179–184

<noscript> tag, 178

overview, 159–160, 160–161, 489–490

screen object, 164

Boolean data, data type, 24

border, adding to a Web page, 710

, **tag**, 665

break **statement**

looping and the, 91

summary, 102

switch statement, 80

breakpoints, Microsoft Script Debugger and,
380–381

Browser Object Model (BOM)

<A> tag, 170–175

browser properties checking, 176–178

command window and, 383

document object, 165–169

Document Object Model (DOM) versus, 497

exercise questions, 667–672

exercise solutions, 667–672

history object, 163

images[] Array, 167–169

links[] Array, 169

location object, 163–164

navigator object, 164, 179–184

<noscript> tag, 178

overview, 159–160, 160–161, 489–490

screen object, 164

similarities between the Document Object Model
(DOM) and the, 506–507

summary, 185

window object, 161–163

browser properties, checking for supported,
176–178

browser support, remote scripting and, 621–622

browser type

code for checking, 668–671

exercise questions, 186

browser window, opening a new, 257–261

browsers

back button, 653

checking for different, 179–184

compatibility of, 12–13

differences between, 13

differences with plug-ins and ActiveX controls, 614–615

displaying errors in, 30–34

example for changing background color in the, 7–8

JavaScript tool, 5

script language defaults, 7

summary, 186

testing order, 625

version checking, 175–184

version compatibility, 12

W3C guidelines and, 160

bubbles **property**, event **object**, 522–526

building

- a DHTML (Dynamic HTML) toolbar, 528–540
- an XML (Extensible Markup Language) application, 582–592

butAddWedexampleonclick(), **function**, 219–220

butCalculate_onclick(), **function**, 402–403

butCheckForm_onclick(), **function**, 201–203

butCheckfunctiononclick(), **function**, 212–213

butCheckQ_onclick(), **function**, 283–284

butCheckValid_onclick(), **function**, 303–304

butGetText_onclick(), **function**, 269

butOpenWin_onclick(), **function**, 267–269

butPlay_onclick(), **function**, 613–614

butShowVisitedfunctiononclick(), **function**, 255–257

butToCent_onclick(), **function**, 673–674

button, adding to a pop-up window, 681–686

button **property**, mouseEvent **object**, 522–526

button1_onclick(), **function**, 320–321, 384–388

button_mouseHandler(), **function**, 535–537

buttononCheckQfunctiononclick(), **function**, 229–233

Buttons

- HTML Forms, 194–197
- summary, 234

C

cache, cleaning the, 408–410

calcFactorial(), **function**, 400–401

calculating

- the date, 663–664
- temperature conversion, 657–658
- times tables, 660–663

Call Stack Window, Microsoft Script Debugger, 383–386

calling

- functions, 93
- methods, 156

cancelable **property**, event **object**, 522–526

^ (caret), position character, 305–308

[^...] (caret and periods in brackets), character class, 300–304

carriage return, escape sequence, 23

Cascading Style Sheets (CSS)

- # (pound sign) and, 471
- hexadecimal values and, 463
- overview, 458–459, 462
- property name guidelines, 471
- rgb() notation, 463–464
- <style> tag, 459–461
- style attribute, 461
- styling properties, 462–470
- url() notation, 464

case

- default, 80
- sensitivity errors, 363–364

case **statements**, switch **statement**, 80

CDATA, XML (Extensible Markup Language), 556

CDATASection **object**, high-level DOM **object**, 502

ceil() **method**, overview, 123

changeImg(), **function**, 174–175

character class, defined, 300

character data, XML (Extensible Markup Language), 555–556

characters

- escape, 23
- reusing groups of, 310–311

charAt() **method**, String **object**, 291

charCodeAt() method, String object

`charCodeAt()` **method**, **String object**, **291**

check box

creating a, 207–214

summary, 234

Checkbox object, **HTML Forms**, **208–214**

CHECKED, **attribute**, **208**

`checkEmail()`, **function**, **643–644**

`checkEmail_callback()`, **function**, **643–644**,
724–726

`checkUsername()`, **function**, **642–644**,
650–651

`checkUsername_callback()`, **function**,
643–644, **650–651**, **724–726**

child node, **defined**, **499**

`chkDaylightSaving_onclick()`, **function**,
345–347

Choose File dialog box, **613**

`choosePageframesonchange()`, **function**,
254–257

class

character, 300

defined, 460

defining a, 147–149

`HttpRequest`, 655, 716

`JavaScript`, 146–156

object instances, 149–151

overview, 146–147

class attribute, **using the**, **475–478**

classid attribute, **using the**, **602–605**

clause, **finally**, **404**

cleaning, **the cache**, **408–410**

clearing, **the temporary Internet file folder in**
Internet Explorer, **408–410**

`clearInterval()` **method**

summary, 360

using the, 352–353

`clearTimeout()` **method**, **using the**, **349–352**

click, **mouse event**, **523–526**

client

defined, 3

server versus, 651

clientX property, **MouseEvent object**,
522–526

clientY property, **MouseEvent object**,
523–526

`cloneNode(cloneChildren)` **method**, **Node**
object, **516–518**

closing

braces errors, 364

parentheses errors, 365–366

tags versus opening tags, 558

tags in XML (Extensible Markup Language),
552–553

closure, **defined**, **630**

`cmdStartQuiz_onclick()`, **function**, **354–359**

code

access between frames, 247–257

branching defined, 438

for checking browser type, 670–671

defined, 1

examples opened in Internet Explorer 6 and 7, 6

executing, 1

repeating blocks of, 84–88

stepping through, 375–379

codebase attribute, **using the**, **602–605**

coding. *See also* **coding explanation**

A Simple Form with Validation example, 198–200

A Two-Dimensional Array example, 55–56

Adding and Removing List Options example,
217–218

An Array example, 51–52

Animating Content example, 484–485

Assigning Variables the Values of Other Variables
example, 28–29

Calculations example, 36

Check Boxes and Radio Buttons example,
208–211

Checking a Character's Case example, 113–114

Checking for and Dealing with Different Browsers
example, 179–181

Checking a Passphrase for Alphanumeric Charac-
ters example, 301–302

Checking for Supported Browser Properties
example, 176–177

Concatenating Strings example, 42–43

- Converting a Series of Fahrenheit Values example, 85–86
- Converting Strings to Numbers example, 46–47
- Counting Button Clicks example, 194–195
- Counting Occurrences of Substrings example, 119
- Create HTML Elements and Text with DOM (Document Object Model) JavaScript example, 517
- Creating Attributes example, 519–520
- Creating an IFrame Smart Form example, 646–648
- Creating an XMLHttpRequest Smart Form example, 636–638
- Cross-Browser Image Rollover example, 451–453
- Declaring Variables example, 26–27
- Displaying a Random Image when the Page Loads example, 173–174
- Event Watching example, 205–206
- Fahrenheit to Centigrade example, 40–41
- Fahrenheit to Centigrade Function example, 95
- between frames, 241–247
- Image Rollover with Property Event Handlers example, 455–456
- Image Selection example, 168
- Inter-Window Scripting example, 265–267
- JavaScript's Rounding Functions Results Calculator example, 124–125
- Making the Temperature Converter User-Friendly example, 44–45
- Moving an Element Around example, 478–481
- Multiple Conditions example, 73–74
- Multiple Frames example, 238–240
- Navigating Your HTML Document Using the DOM (Document Object Model) example, 512–513
- onmouseup and onmousedown example, 196
- Opening Up New Windows example, 258–260
- Painting the Document Red example, 7
- Replacing Single Quotes with Double Quotes example, 315–317
- Reversing the Order of Text example, 292–294
- Scripting Frames example, 251–254
- Setting Colors According to the User's Screen Color Depth example, 165–166
- Sorting an Array example, 136
- Splitting the Fruit String example, 312
- Splitting HTML example, 319–320
- Storing Previous Quiz Results example, 430–432
- for tabulating a file, 585–588
- The Cross-Browser Toolbar example, 544–547
- The forms Array example, 188–189
- The if Statement example, 67–68
- The srcElement Property example, 443–445
- The target Property example, 448–449
- The Toolbar in Firefox and Opera example, 541–543
- The type Property for the Other Browsers example, 447
- The Way Things Flow example, 8–10
- The World Time Method of the Date Object example, 334–336
- try...catch and Throwing Errors example, 397–399
- Updating a Banner Advertisement example, 349–351
- Using the className Property example, 476–477
- Using Cookies to Check for Web Site Changes example, 424–427
- Using the Date Object to Retrieve the Current Date example, 138–139
- Using the DOM (Document Object Model) Event Model example, 525
- Using the Frameset Page as a Module example, 242–244
- Using pow() example, 127–128
- Using the select Element for Date Difference Calculations example, 221–224
- Using setCookie() example, 420–421
- Using the style Object example, 472–474
- Using the switch Statement example, 80–82
- Using the type Property example, 439–441
- World Time Converter (Part I) example, 341–344
- World Time Converter (Part II) example, 353
- Writing the Current Time into a Web Page example, 143–144

coding explanation. See also coding

- A Simple Form with Validation example, 200–203
- A Two-Dimensional Array example, 56–58
- Adding and Removing List Options example, 218–220
- An Array example, 52–54
- Animating Content example, 485–488
- Assigning Variables the Values of Other Variables example, 29
- Calculations example, 36–37
- Check Boxes and Radio Buttons example, 211–214
- Checking a Character's Case example, 114–116
- Checking for and Dealing with Different Browsers example, 181–184
- Checking a Passphrase for Alphanumeric Characters example, 302–304
- Checking for Supported Browser Properties example, 177–178
- Concatenating Strings example, 43
- Converting a Series of Fahrenheit Values example, 86–87
- Converting Strings to Numbers example, 47–48
- Counting Button Clicks example, 195
- Counting Occurrences of Substrings example, 119–120
- Create HTML Elements and Text with DOM (Document Object Model) JavaScript example, 517–518
- Creating Attributes example, 520–521
- creating an IFrame Smart Form example, 649–651
- creating an XMLHttpRequest Smart Form example, 641–644
- Cross-Browser Image Rollover example, 453–455
- Declaring Variables example, 27
- Displaying a Random Image when the Page Loads example, 174–175
- Event Watching example, 206–207
- Fahrenheit to Centigrade example, 41–42
- Fahrenheit to Centigrade Function example, 96
- Image Rollover with Property Event Handlers example, 456–458
- Image Selection example, 168–169
- Inter-Window Scripting example, 267–269
- JavaScript's Rounding Functions Results Calculator example, 125–126
- Making the Temperature Converter User-Friendly example, 45
- Moving an Element Around example, 481–483
- Multiple Conditions example, 74–76
- Multiple Frames example, 240–241
- Navigating Your HTML Document Using the DOM (Document Object Model) example, 514–515
- onmouseup and onmousedown example, 196–197
- Opening Up New Windows example, 260–261
- Painting the Document Red example, 7–8
- Replacing Single Quotes with Double Quotes example, 317–319
- Reversing the Order of Text example, 294–295
- Scripting Frames example, 254–257
- Setting Colors According to the User's Screen Color Depth example, 166–167
- Sorting an Array example, 136–137
- Splitting the Fruit String example, 313–314
- Splitting HTML example, 320–321
- Storing Previous Quiz Results example, 432–434
- for tabulating a file, 589–592
- The Cross-Browser Toolbar, 547–548
- The forms Array example, 189–190
- The if Statement example, 69–70
- The srcElement Property example, 445–446
- The target Property example, 449–450
- The Toolbar in Firefox and Opera example, 543–544
- The type Property for the Other Browsers example, 447–448
- The Way Things Flow example, 11–12
- The World Time Method of the Date Object example, 336–339
- try...catch and Throwing Errors, 400–401
- Updating a Banner Advertisement example, 352
- Using the className Property example, 477–478
- Using Cookies to Check for Web Site Changes example, 427–428

- Using the Date Object to Retrieve the Current Date example, 139–141
- Using the DOM (Document Object Model) Event Model example, 526
- Using the Frameset Page as a Module example, 245–247
- Using `pow()` example, 128–129
- Using the `select` Element for Date Difference Calculations example, 224–227
- Using `setCookie()` example, 421–422
- Using the `style` Object example, 474–475
- Using the `switch` Statement example, 82–83
- Using the `type` Property example, 441–442
- World Time Converter (Part I) example, 344–347
- World Time Converter (Part II) example, 353
- Writing the Current Time into a Web Page example, 144–145
- color**
 - changing background, 463–464
 - changing foreground, 462–463
- color, property, 460–461**
- colorDepth property, screen object, 164**
- COLS attribute**
 - overview, 204–207
 - summary, 234
- Command Window**
 - BOM (Browser Object Model) and, 383
 - Microsoft Script Debugger, 381–383
- comment**
 - defined, 11
 - text defined, 556
- Comment object, high-level DOM object, 502**
- common mistakes. See also errors**
 - Case Sensitivity, 363–364
 - Equals Rather than Is Equal To, 365
 - Incorrect Number of Closing Braces, 364
 - Incorrect Number of Closing Parentheses, 365–366
 - Missing Plus Signs During Concatenation, 364–365
 - Undefined Variables, 362–363
 - Using a Method as a Property and Vice Versa, 366
- comparing, strings, 78–79**
- comparison operators**
 - assigning the results of comparisons, 66
 - assignment operator versus, 66
 - assignment versus comparison, 66
 - common, 65
 - decision making, 65
 - defined, 64
 - precedence, 65
 - summary, 101
- compatibility**
 - browser, 12–13
 - version, 12
- compiled language, interpreted language versus, 2**
- computer language, defined, 1**
- computer requirements, for creating a form with XMLHttpRequest, 635–644**
- concat() method, using the, 131–132**
- concatenation**
 - defined, 42
 - errors, 364–365
- conditions**
 - defined, 64
 - exercise questions/solutions, 659–663
- constructor**
 - class, 147
 - defined, 108
- content, animating, 484–488**
- continue statement**
 - looping and the, 91–92
 - summary, 102
- conversion, data type, 45–49**
- convertToCentigrade(), function, 96**
- cookie, property, 422–428**
- cookie pair, defined, 429**
- cookie security, Internet Explorer 6 and Internet Explorer 7, 429–434**
- cookie string**
 - defined, 407
 - domain, 418
 - expires, 416
 - name and value, 415–416
 - path, 416–417
 - secure, 418

cookies

- checking for enabled, 429
 - components of, 415–418
 - creating, 408, 418–422
 - defined, 404
 - deleting, 422
 - exercise questions, 435, 697–701
 - exercise solutions, 697–701
 - getting the value of, 422–428
 - limitations of, 428–429
 - security and, 429–434
 - session, 428
 - summary, 434
 - using to load advertisements, 699–701
 - viewing in Firefox, 413–415
 - viewing in Internet Explorer, 408–413
- Cookies dialog box, 414**
- Coordinated Universal Time (UTC)**
- defined, 334
 - summary, 360
- copyHistory, window feature, 262–264**
- copying, part of a string, 120–121**
- © (copyright symbol), inserting in a string, 23**
- createAttribute(attributeName) method,**
- Document **object, 516–518**
- createDocument(), function, 578–582**
- createElement(elementName) method,**
- Document **object, 516–518**
- createTextNode(text) method, Document object, 516–518**
- createToolBar(), function, 533, 537**
- createXmlHttpRequest(), function, 623–625**
- createXmlHttpRequest() method, using the, 629, 631–632, 718–726**
- creating**
- an array, 50–54
 - the AskQuestion.htm file for the trivia quiz, 283–284
 - attributes, 519–521
 - check box, 207–214
 - class object instances, 149–151
 - cookies, 408, 418–422
 - cross-browser DHTML (Dynamic HTML), 544–548
 - a Date object, 137–138
 - a DTD file, 559–561
 - forms, 188
 - forms with an iframe, 644–651
 - forms with XMLHttpRequest, 634–644
 - fraGlobalFunctions for the trivia quiz, 280–283
 - fraMenubar for the trivia quiz, 279
 - frames-based Web pages, 678–680
 - the fraQuizPage file for the trivia quiz, 274–278
 - the fraTopFrame file for the trivia quiz, 278–279
 - functions, 92–96
 - the GlobalFunctions.htm file for the trivia quiz, 284–287
 - hidden text boxes, 204
 - HTML files, 239–241
 - links on a Web page, 701–703
 - multi-dimensional arrays, 55–58
 - multiple frames, 238–241
 - objects, 107–109
 - pages, 667–671
 - pages with the RealPlayer plug-in/ActiveX control, 713–715
 - password text boxes, 203–204
 - the QuizPage.htm file for the trivia quiz, 283
 - radio buttons, 207–214
 - remote scripting methods, 629–634
 - Reset buttons, 197
 - Submit buttons, 197
 - text boxes, 197
 - a timer-based trivia quiz, 354–359
 - toolbars, 538–540
 - the top window frameset page for the trivia quiz, 273–274
 - tree structures, 708–710
 - trivia quiz answer radio buttons, 229–233
 - trivia quiz forms, 228–229
 - user interfaces, 672–678
 - visitor count pages, 697–699
 - XML (Extensible Markup Language) documents, 557–566, 711
 - XMLHttpRequest objects in Internet Explorer 7, Firefox, Opera, and Safari, 622–624

cross-browser. See also DHTML (Dynamic HTML)

compatible defined, 12
 DHTML creation, 544–548
 event handlers as attributes, 438–439
 retrieval of an XML (Extensible Markup Language) file, 581–582
 event handlers as properties, 455–458
 events in Internet Explorer, 439–446
 events in other browsers, 446–450
 image rollover, 450–455
 overview, 437–438
 XMLHttpRequest object, 622–625

Cross-Browser Image Rollover, example, 451–455**CSS (Cascading Style Sheets)**

(pound sign) and, 471
 hexadecimal values and, 463
 hiding an iframe through, 650
 overview, 458–459
 property name guidelines, 471
 rgb() notation, 463–464
 rule defined, 459
 <style> tag, 459–461
 style attribute, 461
 styling properties, 462–470
 url() notation, 464

ctrlKey **property**, **mouseEvent** **object**,
523–526

{ } (curly braces), marking out a block of code
 with, 67

currentTarget **property**, **event** **object**,
522–526

D

\D, **character class**, **300–304**

data

adding to an XML (Extensible Markup Language) document, 561–566
 character, 555–556
 converting to numbers, 658–659
 numerical calculations, 35–39
 storing in memory, 24–25
 types of, 21–24

data storage

exercise questions, 657–659
 exercise solutions, 657–659

data type

conversion, 45–49
 exercise questions, 62

database, defined, 24**date**

calculating the, 663–664
 exercise questions, 360
 overview, 333

Date **object**

calculations and dates, 141–142
 creating a, 137–138
 example, 138–141
 exercise questions, 157, 663–667, 689–693
 exercise solutions, 663–667, 689–693
 get methods, 138–141
 getting date values, 138–141
 getting time values, 142–145
 overview, 107, 156
 setting date values, 141
 setting time values, 145–146

debugging

exercise questions, 405
 Firefox, 388–393

decision making, comparison operators, 65**decisions, exercise questions, 102–103****declaring, variables, 25****decrement (—), operators, 37–39**

default **case**, **switch** **statement**, 80

defining, a class, 147–149**Delete Browsing History dialog box, 409****deleting, cookies, 422****DHTML (Dynamic HTML). See also cross-browser**

accessing elements, 470
 browser compatibility and, 450–455
 changing appearances, 470–478
 creating cross-browser, 544–548
 defined, 437
 example, 483–488
 exercise questions, 488, 701–707
 exercise solutions, 701–707
 positioning and moving content, 478–483
 summary, 488

DHTML (Dynamic HTML) toolbar

- adding style to the, 529–532
- building a, 528–540
- Firefox and Opera, 540–544
- HTML structure, 528–529
- Internet Explorer 5+, 526–540
- storing button information, 532–533
- user interaction, 534–537

dialog box

- Choose File, 613
- Cookies, 414
- Delete Browser History, 409
- Delete Browsing History, 409
- Internet Explorer 6 error message, 33–34
- Internet Options, 409
- Options, 413–414
- Temporary Internet Files and History Settings, 410

directories, **window feature, 262–264**

disabling

- JavaScript in Firefox, 5–6
- JavaScript in Internet Explorer, 6

`displayDogs()`, **function, 589–592**

`DisplayEvent()`, **function, 207**

displaying

- browser errors, 30–34
- a daily message, 582–592
- XML (Extensible Markup Language), 566–576

`displayLinks()`, **function, 668**

<div/> elements

- animating content, 488
- `className` property, 477–478
- content positioning, 467–470
- exercise questions, 703–707
- exercise solutions, 703–707
- moving elements, 481–483

`divAdvert_onMouseOut()`, **function, 474–478**

`divAdvert_onMouseOver()`, **function, 474–478**

`doAnimation()`, **function, 486–488**

document

- defined, 8
- element defined, 554

- flow defined, 458

- navigating the, 509–515

- structure in XML, 554

- valid, 559

- validating an XML (Extensible Markup Language), 564

Document object

- `documentElement`, 512–513

- high-level DOM object, 502

- methods, 516–518

- properties of the, 507

- summary, 185

- using the, 165–169, 668

Document Object Model (DOM)

- accessing elements with the, 470

- Browser Object Model (BOM) versus, 497

- changing appearances with the, 471–478

- defined, 159

- event model, 521–526

- exercise questions, 708–710

- exercise solutions, 708–710

- HTML document tree structure, 497–501

- objects, 501–502

- overview, 489–490

- properties and methods, 502–521

- similarities between the Browser Object model and the, 506–507

- summary, 548

- Web standard, 496–497

Document Type Definition (DTD)

- creating a file, 559–561

- defined, 494

- overview, 559

`documentElement`, **property, 507, 512–513**

`DocumentFragment` **object, high-level DOM object, 502**

`document.getElementById()`, **method, 643–644, 706**

`document.images` **array, using the, 672**

`document.lastmodified`, **property, 427–428**

`DocumentType` **object, high-level DOM object, 502**

`document.write()`
 exercise questions, 102–103
 function, 378–383
 method, 660–661
\$ (dollar sign), position character, 305–308
DOM (Document Object Model)
 accessing elements with the, 470
 BOM (Browser Object Model) versus, 497
 changing appearances with the, 471–478
 defined, 159
 event model, 521–526
 exercise questions, 548–549, 708–710
 exercise solutions, 708–710
 objects, 501–502
 overview, 489–490
 properties and methods, 502–521
 similarities between the BOM (Browser Object Model) and the, 506–507
 summary, 548
 Web standard, 496–497
domain
 cookie strings and, 418
 name defined, 412
 name servers defined, 3
`DoPlay()` **method, using the, 715**
== (double equal to sign), operator symbol, 65
`do...while` **loop**
 overview, 90–91
 summary, 101
downloading
 Microsoft Script Debugger, 367–368
 PHP, 636
 Venkman, 388
Dreamweaver, regular expressions, 297
DTD (Document Type Definition)
 creating a file, 559–561
 defined, 494
 overview, 559
dynamic, defined, 12
Dynamic HTML (DHTML). See also cross-browser
 accessing elements, 470
 adding a style to the, 529–532
 browser compatibility and, 450–455
 changing appearances, 470–478
 creating cross-browser, 544–548
 defined, 437

example, 483–488
 exercise questions, 488, 701–707
 exercise solutions, 701–707
 positioning and moving content, 478–483
 summary, 488

E

ECMA (European Computer Manufacturers Association), Web standards and the, 493
ECMAScript Web standards
 overview, 492–493
 summary, 548
Element object
 high-level DOM (Document Object Model) object, 502
 methods, 518–521
 properties of the, 508–509
elements, <form>, 193–194
 adding to the document, 516–518
`apply_templates`, 572–574
 defined, 49
`<div/>`, 467–470, 477–478, 481–483, 485–488, 703–707
 document, 554
`<embed>/<embed/>`, 596–601, 607–614
 empty, 553
 finding involved, 528
`<input/>`, 481–483
 nesting in XML, 553
`<noembed/>`, 608–614
 object, 602–605
 opening versus closing tags, 558
`<p/>`, 468–470
`<param/>`, 605–614
 removing from the document, 516–518
 returning, 502–506
 returning a reference to the topmost, 507
 root, 554
`<script/>`, 703
 select, 215–227
 showing/hiding, 465–466
`<tbody/>`, 591–592
 textarea, 204–207
`<thead>`, 590–592
`<tr/>`, 590–592
 XML (Extensible Markup Language), 554–555

`elements[]` **array property**, <form> **elements**,
193

`elements[]` **property**, `Form` **object**, **190**

`else if` **statement**, **example**, **78**

`else` **statement**

example, 77–78

summary, 101

`<embed>/<embed/>`, **elements**, **596–601**,
608–614

embedding, **defined**, **596**

empty element, **defined**, **553**

enabling, **the script debugger in Internet Explorer**,
5, **370**

`Entity` **object**, **high-level DOM (Document**
Object Model) object, **502**

entity references, **overview**, **555–556**

`Entityreference` **object**, **high-level DOM (Doc-**
ument Object Model) object, **502**

equals/is equal to, **errors**, **365**

error handling

exercise questions, 693–696

exercise solutions, 693–696

summary, 404

errors. *See also* **common mistakes**

case sensitivity, 363–364

closing braces, 364

closing parentheses, 365–366

concatenation, 364–365

dealing with, 32–34

defined, 394

displaying in browsers, 30–34

displaying in Internet Explorer, 31–32

displaying in Firefox, 30–31

equals/is equal to, 365

example, 397–401

exercise questions, 405

finally clauses, 404

logic versus syntax, 693–696

methods versus properties, 366

nested try...catch statements, 401–403

preventing, 393–394

summary, 404

throwing, 397–401

try...catch statements, 394–397

undefined variables, 362–363

`escape()`, **function**, **419–420**

escape character, **defined**, **23**

escape sequences

common, 23

Unicode, 23

European Computer Manufacturers Association
(ECMA), **Web standards and the**, **493**

event handlers

as attributes, 170–172, 438–439

defined, 170

Image Rollover with Property, 455–458

onblur, 198, 233–234

onchange, 198

onclick, 673–674

onfocus, 193–194, 198, 233–234

onkeydown, 198

onkeypress, 198

onkeyup, 198

onload, 667–671

onmouseover/onmouseout, 477–478, 715

onreadystatechange, 627–629

onselect, 198

onsubmit, 723–726

as properties, 172–175, 455–458

summary, 186

event model, **Internet Explorer 5+**, **526–528**

event watching, **example**, **204–207**

`eventPhase` **property**, `event` **object**, **522–526**

events. *See also* **Web Page events**

defined, 169

firing, 527

in non-Internet Explorer browsers, 446–450

onload, 186

onmousedown/onmouseup, 196–197

onmouseover/onmouseout, 672

select element, 221–227

submit, 482

example

A Simple Form with Validation, 198–203

Adding and Removing List Options, 217–220

Animated Advertising, 483–488

Animating Content, 484–488

- Assigning Variables the Values of Other Variables, 28–29
- Changing the Background Color of the Browser, 7–8
- Check Boxes and Radio Buttons, 208–214
- Checking a Character's Case, 113–116
- Checking for and Dealing with Different Browsers, 179–184
- Checking for the Existence of Cookies, 424–428
- Checking a Passphrase for Alphanumeric Characters, 301–304
- Checking for Supported Browser Properties, 176–178
- Concatenating Strings, 42–43
- Converting a Series of Fahrenheit Values, 85–87
- Converting Strings to Numbers, 46–49
- Counting Button Clicks, 194–195
- Counting Occurrences of Substrings, 119–120
- Create HTML Elements and Text with DOM JavaScript, 517–518
- Creating an Array, 51–54
- Creating Attributes, 519–521
- Creating a Multi-Dimensional Array, 55–58
- Creating Multiple Frames, 238–241
- Cross-Browser Image Rollover, 451–455
- Declaring Variables, 26–27
- Displaying a Random Image when the Page Loads, 173–175
- Event Watching, 205–207
- Fahrenheit to Centigrade converter, 40–42
- Fahrenheit to Centigrade Function, 95–96
- forms Array, 188–190
- functions, 95–96
- if Statement, 67–70
- Iframe Smart Form, 646–651
- Image Rollover with Property Event Handlers, 455–458
- Image Selection, 168–169
- Inter-Window Scripting, 265–269
- JavaScript's Rounding Functions Results Calculator, 124–126
- Moving an Element Around, 478–483
- Multi-Condition if Statements, 73–76
- Navigating Your HTML Document Using the DOM, 512–515
- Numerical Calculations, 36–37
- onmousedown/onmouseup Events, 196–197
- Opening Up New Windows, 258–261
- Operator Precedence, 40–42
- Parsing a Web Page, 8–12
- pow Method, 127–129
- Replacing Single Quotes with Double Quotes, 315–318
- Reversing the Order of Text, 292–295
- Scripting Frames, 251–257
- Setting Colors According to the User's Screen Color Depth, 165–167
- Sorting an Array, 136–137
- Splitting the Fruit String, 312–314
- Splitting HTML, 319–321
- srcElement Property, 443–446
- for Statement, 85–87
- Storing Previous Quiz Results, 430–434
- switch Statement, 80–84
- Tabulating the Dogs, 585–592
- target Property for Non-Internet Explorer Browsers, 448–450
- temperature converter, 44–45
- The Cross-Browser Toolbar, 544–548
- The Toolbar in Firefox and Opera, 541–544
- The World Time Method of the Date Object, 334–339
- try...catch/throw Statements, 397–401
- Updating a Banner Advertisement, 349–352
- Using the className Property, 476–478
- using the Date Object to Retrieve the Current Date, 138–141
- Using the DOM Event Model, 525–526
- Using the Frameset Page as a Module, 242–247
- Using the Select Element for Date Difference Calculations, 221–227
- Using setCookie(), 420–422
- Using the style Object, 472–475
- Using the type Property, 439–442
- World Time Converter, 341–347, 353
- Writing the Current Time into a Web Page, 143–145
- XMLHttpRequest Smart Form, 636–644
- XSLT Template, 571–574

!= (exclamation point equal to), operator symbol

!= (exclamation point equal to), operator symbol, 65

exception, defined, 394

Exception object, error handling, 395–397

executing code, defined, 1

exercise questions

ActiveX controls, 618, 713–715

BOM (Browser Object Model), 667–672

browser type, 186

conditions, 659–663

cookies, 435, 697–701

data storage, 657–659

data type, 62

date, 360

Date object, 157, 663–667, 689–693

decisions, loops, and functions, 102–103, 659–663

DHTML (Dynamic HTML), 488, 701–707

Document Object Model (DOM), 548–549, 708–710

DOM (Document Object Model), 548–549, 708–710

error handling, 693–696

errors, 405

Extensible Markup Language (XML), 593, 710–713

fix() function, 157, 665

frames and windows, 289, 678

href property, 186, 667

HTML code, 548–549

HTML Forms, 235

HttpRequest class, 655, 716–726

 tag, 186, 671

Math object, 663–667

Microsoft Script Debugger, 405, 693

plug-ins, 618, 713–715

pop-up window, 289, 681

pow() method, 157, 665

prompt() function, 62, 658–659

RegExp() object, 330, 686–689

regular expressions, 330–331, 687–689

remote scripting, 716–726

storing, 157

string manipulation, 686–689

time, 360

timers, 689–693

user interface, 235, 672–678

XML (Extensible Markup Language), 593, 710–713

exercise solutions

ActiveX controls, 713–715

Browser Object Model (BOM), 667–672

conditions, 659–663

cookies, 697–701

data storage, 657–659

Date object, 663–667, 689–693

decisions, loops, and functions, 659–663

Document Object Model (DOM), 708–710

error handling, 693–696

Extensible Markup Language (XML), 710–713

fix() function, 665

frames and windows, 678

href property, 667

HttpRequest class, 716–726

 tag, 671

Math object, 663–667

plug-ins, 713–715

pop-up window, 681

pow() method, 665

prompt() function, 658–659

RegExp() object, 686–689

regular expressions, 687–689

remote scripting, 716–726

string manipulation, 686–689

timers, 689–693

user interface, 672–678

XML (Extensible Markup Language), 710–713

expires, cookie strings and, 416

Extensible Markup Language (XML)

CDATA, 556

character data, 555–556

closing tags, 552–553

comments, 556–557

creating a document, 557–566

displaying, 566–576

document structure, 554

- exercise questions, 593, 710–713
- exercise solutions, 710–713
- formatting, 566–576
- manipulating with JavaScript, 576–592
- nesting elements, 553
- opening tags, 552–553
- overview, 551–552
- style sheets and, 566–568
- summary, 592–593
- template rule, 570
- Web standard, 493–494
- as a well-formed language, 552
- XML (Extensible Markup Language) elements, 554–555
- XSLT transformations, 568–576

Extensible Markup Language (XML) documents

- adding data to, 561–566
- creating, 711
- creating a DTD file, 559–561
- Document Type Definition (DTD), 559
- linking to XSL style sheets, 574–576

Extensible Markup Language (XML) file

- cross-browser retrieval of an, 581–582
- exercise questions, 593
- loading an, 579–581, 711–713
- retrieving in Firefox and Opera, 580
- retrieving in Internet Explorer, 577–579

external, defined, 237

F

\f, escape sequence, 23

factorial, defined, 399

files. See also Trivia Quiz Files

- creating HTML, 239–241
- Zip, 552

finally clause, error handling, 404

Firefox

- adding a plug-in to the page in, 596–598
- blocking pop-ups in, 257
- the blur event and, 203

- checking for/installing plug-ins in, 598–601
- creating an XMLHttpRequest object in, 624
- Debugging, 388–393
- DHTML (Dynamic HTML) toolbar, 540–544
- disabling JavaScript in, 5–6
- displaying errors in, 30–31
- DOM (Document Object Model) Standards and, 497
- Exception object, 395–397
- retrieving an XML (Extensible Markup Language) file in, 580
- status bar text and, 359
- troubleshooting the blur event, 203
- validating XML (Extensible Markup Language) documents in, 564
- viewing cookies in, 413–415
- viewing the JavaScript Console from, 30
- web site, 5

Firefox 2.0, Show Cookies button, 415

Firefox Venkman, summary, 404

firstCall, function, 385

firstChild property, Node object, 510–512

fix() function

- defining, 128–129
- exercise questions, 157, 665
- exercise solutions, 665–667

floating-point numbers, defined, 22

floor() method, overview, 123

focus() method

- <form> elements, 193–194
- summary, 233

font-family, property, 460–461, 464–465

font-size, property, 460–461, 464–465

font-style

- attribute, 461
- property, 465, 474–475

font-weight, property, 465

fonts/text, styling, 464–465

<form>/</form> tag

- summary, 233
- using the, 188–189, 723–726

<form> elements

<form> elements

- `blur()` method, 193–194
- `elements[]` array property, 193
- `focus()` method, 193–194
- `form` property, 193
- HTML elements within, 191–192
- `name` property, 193
- `onfocus/onblur` event handlers, 193–194
- `value` property, 193

for loop

- summary, 101
- using the, 578, 661–662, 668
- work flow of the, 85

for statement

- example, 85–87, 88
- overview, 84–85, 87–88

foreground color, changing, 462–463

for...in loop

- arrays and the, 87–88
- summary, 101

form elements. See HTML forms

form feed, escape sequence, 23

Form object

- `elements[]` property, 190
- `reset()` method, 190
- `submit()` method, 190
- summary, 233
- using the, 188–189

form property

- <form> elements, 193
- summary, 233

formatting

- XML (Extensible Markup Language), 566–576
- XML (Extensible Markup Language) documents
 - with XSLT transformations, 568–576

forms

- creating, 188, 691–693
- creating with an `iframe`, 644–651
- creating with `XMLHttpRequest`, 634–644
- exercise questions, 360
- HTML, 187–188
- received data, 635
- requesting information for, 634–635

forms[] Array

- overview, 188–190
- summary, 185

form_submit(), function, 723–726

forward() method, history object, 163

fpCallback, function, 630–631

<frame>/<frameset> tags, window object, 238–257

fraGlobalFunctions, creating for the trivia quiz, 280–283

fraMenubar, creating for the trivia quiz, 279

frames

- code access between, 247–257
- coding between, 241–247
- exercise questions, 289, 678
- exercise solutions, 678–680
- ID attribute, 240–241
- multiple, 238–241
- overview, 237
- scripting, 251–257
- security, 270–271
- summary, 288
- trivia quiz, 272–273
- window object and, 238–257

frames[] array

- summary, 288
- window object, 249

frames-based, Web page creation, 678–680

frameset, diagrammed, 248

fraQuizPage, creating for the trivia quiz, 274–278

fraTopFrame, creating for the trivia quiz, 278–279

fromCharCode() method, overview, 117

fromElement property, event object, 528

function

- calling a, 93
- closure, 630
- creating a, 92–96
- creating for the trivia quiz, 97–100
- defined, 11, 92
- exercise questions, 102–103, 659–663
- exercise solutions, 659–663

- invoking a, 93
- overview, 92
- parameter passing, 93–94
- summary, 102
- function (ActiveX and Plug-Ins)**
 - play(), 715
 - window_onload(), 715
- function (ActiveX and Remote Scripting), but-**
 - Play_onclick(), 613–614
- function (Ajax and Remote Scripting)**
 - checkEmail(), 643–644
 - checkEmail_callBack(), 643–644, 724–726
 - checkUsername(), 642–644, 650–651, 724–726
 - checkUsername_callBack(), 643–644, 650–651
 - createXmlHttpRequest(), 623–625
 - form_submit(), 723–726
 - fpCallback, 630–631
 - handleData(), 633–634
 - oHttp_readyStateChange(), 628–629
 - request_readystatechange(), 630–631, 720–726
- function (Basic JavaScript), alert(), 11–12**
- function (Browser Programming)**
 - changeImg(), 174–175
 - displayLinks(), 668
 - getBrowserName(), 179–184, 670–671
 - getBrowserVersion(), 179–184
 - linkSomePage_onclick(), 172–173
- function (Common Mistakes, Debugging, and Error Handling)**
 - butCalculate_onclick(), 402–403
 - button1_onclick(), 384–388
 - calcFactorial(), 400–401
 - document.write(), 378–383
 - firstCall(), 385
 - secondCall(), 385
 - thirdCall(), 385
 - writeTimesTable(), 373–374, 390–393
- function (Cookies)**
 - escape(), 419–420
 - getCookieValue(), 431–434
 - getQuestion(), 431–434
 - setCookie(), 420–422, 431–434
 - unescape(), 419–420
- function (Data Types and Variables)**
 - isNaN(), 48–49
 - parseFloat(), 46–48, 658–659
 - parseInt(), 46–48
 - prompt(), 41, 62, 658–659
- function (Date, Time, and Timers)**
 - chkDaylightSaving_onclick(), 345–347
 - cmdStartQuiz_onclick(), 354–359
 - getQuestion(), 358–359
 - getTimeString(), 346–347
 - resetQuiz(), 355–359
 - switchImage(), 352
 - updateTime(), 345–347, 353
 - updateTimeLeft(), 356–359
 - updateTimeZone(), 344–347
 - window_onload(), 352, 353
- function (Decisions, Loops, and Functions)**
 - answerCorrect(), 99–100
 - convertToCentigrade(), 96
 - document.write(), 102–103
 - isNaN(), 663
 - Number(), 69
 - prompt(), 69
 - writeTimesTable(), 661–662
- function (Dynamic HTML)**
 - divAdvert_onMouseOut(), 474–478
 - divAdvert_onMouseOver(), 474–478
 - doAnimation(), 486–488
 - image_EventHandler(), 445–446, 449–458
 - moveBox(), 482–483
 - moveDiv(), 705–707
 - paragraph_eventHandler(), 441–442
 - showBoxOne(), 703
 - showBoxTwo(), 703
 - startTimer(), 705–707
- function (Dynamic HTML in Modern Browsers)**
 - button_mouseHandler(), 535–537
 - createToolBar(), 533, 537
 - handleClick(), 526

function (HTML Forms)

function (HTML Forms)

answerCorrect(), 229–233
butAddWedexampleonclick(), 219–220
butCheckForm_onclick(), 201–203
butCheckfunctiononclick(), 212–213
butToCent_onclick(), 673–674
buttononCheckQfunctiononclick(), 229–233
DisplayEvent(), 207
getMonth(), 226
getQuestion(), 229–233
myButton_onclick(), 195
myButton_onmousedown(), 196–197
myButton_onmouseup(), 196–197
radCPUSpeed_onclick(), 212
recalcDateDiff(), 226
txtAge_onblur(), 201–203
txtName_onchange(), 201–203
updateOrderDetails(), 677–678
windowfunctiononload(), 189–190
windowonload(), 225–226
writeMonthOptions(), 225
writeOptions(), 224

function (Object-Based Language), fix, 128–129, 157

function (String Manipulation)

answerCorrect(), 323–329
butCheckValid_onclick(), 303–304
button1_onclick(), 320–321
getAnswer(), 328–329
getQuestion(), 323–329
regExpIs_valid(), 302–304
replaceQuote(), 317–318
splitAndReverseText(), 294–295

function (Windows and Frames)

addBookToBasket(), 683
addPage(), 245–247
addToBasket(), 685–686
answerCorrect(), 285–287
butCheckQ_onclick(), 283–284
butGetText_onclick(), 269
butOpenWin_onclick(), 267–269

butShowVisitedfunctiononclick(), 255–257
choosePageframesonchange(), 254–257
getQuestion(), 285–287
removeItem(), 683
returnPagesVisited(), 245–247
setFrameAndPageControls(), 255–257
showBasket(), 684
updateOrderDetails(), 679–680
windowonload(), 241
window_onunload(), 269

function (XML)

createDocument(), 578–582
displayDogs(), 589–592
getDailyMessage(), 583–585
xmlDoc_readyStateChange(), 580

G

g, attribute, 299, 330

garbage collection, defined, 27

Garrett, Jesse James, *Ajax: A New Approach to Web Applications*, 621

getAnswer(), function, 328–329

getAttribute(attributeName) method, Element object, 519–521

getBrowserName(), function, 179–184, 670–671

getBrowserVersion(), function, 179–184

getCookieValue(), function, 431–434

getDailyMessage(), function, 583–585

getDate() method, overview, 138

getDay() method
overview, 138
using the, 692–694

getElementsById() method
DOM method, 486–488, 503–504
using the, 470

getElementsByTagName() method, DOM object, 505–506, 590–592

getFullYear() method, overview, 138

getHours() method, overview, 142–145

`getMilliseconds()` **method, using the, 142–145**
`getMinutes()` **method, overview, 142–145**
`getMonth()` **function, 226**
`getMonth()` **method**
 overview, 138
 using the, 664
`getQuestion()` **function, 229–233, 285–287, 323–329, 358–359, 431–434**
`getSeconds()` **method, using the, 142–145**
`getTimeString()` **function, 346–347**
`getTimezoneOffset()` **method**
 Date object, 334–339
 summary, 360
getting, attributes, 518–521
`getUTCDate()` **method, using the, 340–348**
`getUTCDay()` **method, using the, 340–348**
`getUTCFullYear()` **method, using the, 340–348**
`getUTCHours()` **method, using the, 340–348**
`getUTCMilliseconds()` **method, using the, 340–348**
`getUTCMinutes()` **method, using the, 340–348**
`getUTCMonth()` **method, using the, 340–348**
`getUTCSeconds()` **method, using the, 340–348**
global
 match attribute, 299
 object defined, 161
 scope defined, 96
global variable, `switchDirection`, 486–488
`GlobalFunctions.htm`, **creating for the trivia quiz, 284–287**
GMT (Greenwich Mean Time), defined, 334
`go()` **method, history object, 163**
Google Maps, remote scripting and, 620
Google Suggest, remote scripting and, 620–621
> (greater than)
 operator symbol, 65
 XML (Extensible Markup Language) character data, 555–556
>= (greater than or equal to), operator symbol, 65
Greenwich Mean Time (GMT), defined, 334
grouping, regular expressions, 308–311

`>`, **entity reference, 555–556**

H

`handleClick()` **function, 526**
`handleData()` **function, 633–634**
`hasChildNodes()` **method, Node object, 516–518**
height, window feature, 262–264
height property, screen object, 164
hexadecimal values, assigning colors with, 463
hidden-frame technique, process of the, 644–651
hidden text box, creating a, 204
hiding
 elements, 465–466
 an `iframe`, 650
high-level DOM objects, overview, 501–502
history object
 overview, 163
 summary, 185
href property
 exercise questions, 186, 667
 exercise solutions, 667–668
 location object, 164
HTML
 adding to a new window, 261–262
 code exercise questions, 548–549
 comments versus XML (Extensible Markup Language) comments, 557
 document tree structure, 497–501
 editor as a JavaScript tool, 4–5
 elements summary, 233
 elements within `<form>` elements, 191–192
 Files, 239–241
 origination of, 490–492
 page elements in Netscape, 490
 page examples, 499–501
 page styling, 459–470
 splitting, 319–321
 structure of a DHTML (Dynamic HTML) toolbar, 528–529
 Web standard, 492
 Web standards summary, 548
 XHTML versus, 495

HTML Forms

- `alert()` method, 214
- Array object, 213
- Button object, 194–195
- check boxes and radio buttons, 207–214
- Checkbox object, 208–214
- CHECKED attribute, 208
- creating a Submit/Reset button, 197
- creating a text box, 197
- example, 198–203
- exercise questions, 235
- hidden text box, 204
- onmousedown/onmouseup event, 196–197
- Option object, 215–221
- options[] array, 216
- overview, 187–188
- password text box, 203–204
- Radio object, 208–214
- `<select>/</select>` tags, 215
- select elements, 215–227
- summary, 233
- Text Elements, 197–207
- textarea element, 204–207

HTTP (HyperText Transfer Protocol)

- defined, 3
- origination of, 490–492
- HttpRequest **class**
 - exercise questions, 655, 716–726
 - exercise solutions, 716–726
- HttpRequest **constructor, using the, 629–631**
- Hungarian notation, defined, 25
- HyperText Transfer Protocol (HTTP)
 - defined, 3
 - origination of, 490–492

I

- i, **attribute, 299, 330**
- icon
 - Step Into, 274
 - Step Out, 375
 - Step Over, 376
 - styling, 532
- ID **attribute, frames, 240–241**

IE. See Internet Explorer

- if **statement**
 - example, 67–70
 - multiple conditions inside an, 73–76
 - overview, 66–67
 - summary, 100, 101
- iframes
 - creating a form with, 644–651
 - hiding, 650
 - Safari and, 645
 - server response, 645
- image, **inserting in a page, 671–672**
- Image Rollover, with Property Event Handlers, **455–458**
- image_EventHandler(), **function, 445–446, 449–458**
- images[]
 - Array, 167–169
 - summary, 185
- `` **tag**
 - exercise questions, 186, 671
 - exercise solutions, 671–672
- img **object, using the, 672**
- increment (**++**)
 - operators, 37–39
 - for statement, 85
- index **property, Option object, 215–216**
- index value, **defined, 49**
- indexOf() **method, overview, 117–118, 291, 423**
- infinite loop, **defined, 90**
- initialization, **for statement, 85**
- `<input>`, **tag, 197–203**
- `<input/>`, **elements, 481–483**
- insertBefore(newNode, referenceNode)
 - method, Node object, 516–518**
- inserting
 - advertisements into a web page, 690–691
 - an image in a page, 671–672
 - JavaScript into a web page, 6–7
- installing
 - an ActiveX control, 606–607
 - Microsoft Script Debugger, 368–370

plug-ins in Firefox, 598–601

Venkmán, 388–389

a Web server, 635–636

integers, defined, 22

Internet Explorer

adding an ActiveX control to the page in,
602–605

adding new options with, 220–221

blocking pop-ups in, 257

clearing the temporary Internet file folder in,
408–410

cookie security and, 429–434

disabling JavaScript in, 6

displaying errors in, 31–32

Exception object, 395–397

JScript and, 3

navigator.plugins array, 598

retrieving an XML (Extensible Markup Language)
file in, 577–579

validating XML (Extensible Markup Language)
documents in, 564

viewing cookies in, 408–413

Web standards and, 491–492

World Wide Web Consortium (W3C) and, 491

Internet Explorer 5

creating an XMLHttpRequest object in, 622–624

enabling the script debugger in, 370

event model, 526–528

using ActiveX in, 622–624

Internet Explorer 6

cookie security in, 429–434

creating an XMLHttpRequest object in, 622–624

DOM (Document Object Model) properties and,
524

DOM (Document Object Model) Standard and, 497

error message dialog box, 33–34

opening code examples in, 6

Service Pack 1b and ActiveX controls changes,
617

using ActiveX in, 622–624

Internet Explorer 7

cookie security in, 429–434

creating an XMLHttpRequest object in, 624

DOM (Document Object Model) properties and,
524

opening code examples in, 6

status bar text and, 359

Internet Explorer events

srcElement property, 442–446

type property, 439–442

Internet file folder, clearing the temporary, 408–410

Internet Options dialog box, cookies and the, 409

Internet protocol (IP) addresses, defined, 3

interpreted language, compiled language versus, 2

invoking, functions, 93

IP (Internet protocol) addresses, defined, 3

is equal to, error, 365

isNaN(), function, 48–49, 663

J

Java language, JavaScript versus, 2

JavaScript. See also LiveScript

advantages of using, 3–4

classes, 146–156

disabling in Firefox, 5–6

disabling in Internet Explorer, 6

inserting into a web page, 6–7

Java language versus, 2

matching VBScript and, 309–310

objects, 106–107

overview, 1–2

uses of, 4

the Web and, 2–3

JavaScript Console

viewing from Firefox, 30

viewing the, 362

JavaScript objects, using, 107–110

JavaScript Web Applications, tools for creating, 4–6

join() method, using the, 133–134

JScript, Internet Explorer and, 3

L

language, computer, 1

`lastChild` **property**, `Node` **object**, 510–512

`lastIndexOf()` **method**, **overview**, 117–118, 291

leaf node, **defined**, 499

left, **window feature**, 262–264

left operand, **defined**, 65

length **property**

Array objects and the, 107, 233

`elements[]` **property**, 190

overview, 112

String **object**, 291

summary, 233

using the, 130–131

< (less than)

operator symbol, 65

XML (Extensible Markup Language) character data, 555–556

<= (less than or equal to), **operator symbol**, 65

Level 0, DOM (Document Object Model) Standard, 496

Level 1, DOM (Document Object Model) Standard, 496

Level 2, DOM (Document Object Model) Standard, 496

Level 3, DOM (Document Object Model) Standard, 497

`<`, **entity reference**, 555–556

lifespan, **defined**, 416

lifetime

defined, 97

summary, 102

links

adding to a pop-up window, 681–686

creating on a Web page, 701–703

`links[]` **Array**

summary, 185

using the, 169, 668

`linkSomePage_onclick()`, **function**, 172–173

literal value, **defined**, 27

LiveScript. *See also* **JavaScript**

Netscape Navigator 2 and, 2–3

loading

advertisements using cookies, 699–701

an XML (Extensible Markup Language) file, 579–581, 711–713

location **object**

overview, 163–164

summary, 185

window feature, 262–264

logic, **errors versus syntax errors**, 693–696

logical operators

AND (&&), 71

NOT (!), 72

OR (||), 71–72

overview, 70

summary, 101

loop

`do...while`, 90–91

exercise questions, 102–103, 659–663

exercise solutions, 659–663

`for`, 84–87, 578, 661–662, 668

`for...in`, 87–88

infinite, 90

`while`, 88–90, 662–663

looping

`break` statement and, 91

`continue` statement and, 91–92

defined, 84

`for/while` statements, 84–88

summary, 101–102

M

`m`, **attribute**, 299

Mac browsers, `url()` **notation**, 464

machine code, **defined**, 2

`match()` **method**

String **object**, 296, 318–319

summary, 329

matching, **JavaScript and VBScript**, 309–310

Math **object**

`abs()` **method**, 122

`ceil()` **method**, 123

example, 124–126

exercise questions, 663–667

exercise solutions, 663–667

- floor() method, 123
- overview, 122, 156
- pow() method, 127–129
- random() method, 126
- round() method, 123
- menubar, **window feature**, 262–264
- metaKey **property**, mouseEvent **object**, 523–526
- method**
 - troubleshooting a, 366
 - versus properties errors, 366
- method (ActiveX and Plug-Ins)**
 - DoPlay(), 715
 - setSource(), 613–614, 715
- method (Ajax and Remote Scripting)**
 - createXmlHttpRequest(), 629, 631–632, 718–726
 - document.getElementsById(), 643–644
 - open(), 626, 719–726
 - remote scripting, 631–632
 - send(), 626–627, 629, 631–632, 718–726
- method (Browser Programming)**
 - back(), 163
 - forward(), 163
 - go(), 163
 - replace(), 164
- method (Common Mistakes, Debugging, and Error Handling)**, **getDay()**, 694
- method (Cookies)**
 - indexOf(), 423
 - toGMTString(), 416
- method (Date, Time, and Timers)**
 - clearInterval(), 352–353, 360
 - clearTimeout(), 349–352
 - getDay(), 692–693
 - getTimeZoneOffset(), 334–339, 360
 - for getting UTC date and time, 339–348, 340–348
 - getUTCDate(), 340–348
 - getUTCDay(), 340–348
 - getUTCFullYear(), 340–348
 - getUTCHours(), 340–348
 - getUTCMilliseconds(), 340–348
 - getUTCMinutes(), 340–348
 - getUTCMonth(), 340–348
 - getUTCSeconds(), 340–348
 - setHours(), 334–339
 - setInterval(), 352–353, 360, 691
 - setTimeout(), 352
 - setUTCDate(), 339–348
 - setUTCFullYear(), 339–348
 - setUTCHours(), 334–339
 - setUTCMilliseconds(), 339–348
 - setUTCMinutes(), 339–348
 - setUTCMonth(), 339–348
 - setUTCSeconds(), 339–348
 - toLocaleDateString(), 334–339
 - toLocaleString(), 334–339
 - toLocaleTimeString(), 334–339
 - toTimeString(), 334–339
 - toUTCString(), 334–339
 - window_onload(), 353
- method (Decisions, Loops, and Functions)**, **document.write()**, 660–661
- method (Dynamic HTML)**
 - document.getElementsById(), 706
 - getElementsById(), 470, 486–488
- method (Dynamic HTML in Modern Browsers)**
 - appendChild(newNode), 516–518
 - cloneNode(cloneChildren), 516–518
 - createAttribute(attributeName), 516–518
 - createElement(elementName), 516–518
 - createTextNode(text), 516–518
 - Document object, 516–518
 - Element object, 518–521
 - getAttribute(attributeName), 519–521
 - getDay, 138
 - getElementsById(), 503–504
 - getElementsByTagName(), 505–506
 - hasChildNode(), 516–518
 - insertBefore(newNode, referenceNode), 516–518
 - Node object, 516–518
 - removeAttribute(attributeName), 519–521
 - removeChild(childNode), 516–518
 - replaceChild(newChild, oldChild), 516–518
 - setAttribute(attributeName, value), 519–521

method (HTML Forms)

- `add()`, 220–221
- `alert()`, 214
- `blur()`, 193–194
- `focus()`, 193–194, 233
- `onblur()`, 193–194
- `remove()`, 220–221
- `reset()`, 190, 234
- `submit()`, 190, 234

method (Object-Based Language)

- `abs()`, 122
- `ceil()`, 123
- `charAt()`, 112–116
- `charCodeAt()`, 112–116
- `concat()`, 131–132
- `document.write()`, 665
- `floor()`, 123
- `fromCharCode()`, 117
- `getDate`, 138
- `getFullYear()`, 138
- `getHours()`, 142–145
- `getMilliseconds()`, 142–145
- `getMinutes()`, 142–145
- `getMonth()`, 138, 664
- `getSeconds()`, 142–145
- `indexOf()`, 117–118
- `join()`, 133–134
- `lastIndexOf()`, 117–118
- `pow`, 127–129, 157
- `random()`, 126
- `reverse()`, 135–137
- `round()`, 123
- `setHours()`, 145–146
- `setMilliseconds()`, 145–146
- `setMinutes()`, 145–146
- `setSeconds()`, 145–146
- `slice()`, 132–133
- `sort()`, 107, 134–135, 665
- `substr()`, 120–121
- `substring()`, 120–121
- `toDatestring()`, 138
- `toFixed()`, 129–130
- `toLowerCase()`, 121–122

- `toString()`, 142–145

- `toUpperCase()`, 121–122

method (String Manipulation)

- `blur()`, 233–234
- `charCodeAt()`, 291
- `lastIndexOf()`, 291
- `match`, 296, 318–319, 329
- `replace()`, 295–296, 314–318, 329
- `search()`, 296, 318, 329
- `split()`, 292–295, 312–314, 329
- `substr()`, 291
- `substring()`, 291
- `test()`, 302–304
- `toLowerCase()`, 291
- `toUpperCase()`, 291

method (Windows and Frames)

- `moveBy()`, 270, 288
- `moveTo()`, 269–270, 288
- `open()`, 257–261, 288
- `resizeBy()`, 270, 288
- `resizeTo()`, 269–270, 288
- `substr()`, 245–247
- `window.close()`, 288
- `window.open()`, 288

method (XML), `getElementsByTagName()`, 590–592

Microsoft

- MSXML library, 577–579
- web site, 5

Microsoft Script Debugger

- breakpoints, 380–381
- Call Stack Window, 383–386
- Command Window, 381–383
- downloading the, 367–368
- exercise questions, 405, 693
- exercise solutions, 693
- installing the, 368–370
- opening a page in, 370–375
- overview, 367
- Running Documents Window, 386–388
- stepping through code, 375–379
- summary, 404
- using the, 370–388

- versions, 370
- viewing pages, 387
- Microsoft Visual Studio, script debuggers and, 367**
- Microsoft Word, regular expressions, 297**
- MIME (Multipurpose Internet Mail Extensions), type, 597–601**
- `mimeType` array, **navigator** object, 600–601
- mistakes, common, 361–366**
- modal, defined, 11**
- modular, defined, 237**
- `mousedown`, **mouse event**, 523–526
- `mousemove`, **mouse event**, 523–526
- `mouseout`, **mouse event**, 523–526
- `mouseover`, **mouse event**, 523–526
- `mouseup`, **mouse event**, 523–526
- `moveBox()`, **function**, 482–483
- `moveBy()` **method**
 - summary, 288
 - window object, 270
- `moveDiv()`, **function**, 705–707
- `moveTo()` **method**
 - summary, 288
 - window object, 269–270
- moving**
 - content, 478–483
 - windows, 269–270
- multi-condition if statements, example, 73–76**
- multi-dimensional array, defined, 54**
- multi-frame application, defined, 18**
- multi-line flag, attribute for, 299**
- multiple frames, example, 238–241**
- Multipurpose Internet Mail Extensions (MIME), type, 597–601**
- `myButton_onclick()`, **function**, 195
- `myButton_onmousedown()`, **function**, 196–197
- `myButton_onmouseup()`, **function**, 196–197

N

- `\n`, **escape sequence**, 23
- `{n}`, **repetition character**, 304–305

- `{n, }`, **repetition character**, 304–305
- name, of cookie strings, 415–416**
- name attribute, summary, 233**
- name property**
 - `<form>` elements, 193
 - summary, 233
- NamedNodeMap object, base DOM object, 501**
- names, storing and sorting, 664–665**
- namespace, defined, 495**
- Nan, value, 48**
- navigating**
 - the document, 509–515
 - a tree structure, 708–710
- navigator object**
 - browser checking with the, 179–184
 - summary, 185
 - using the, 164
- `navigator.plugins` **array, in Internet Explorer, 598**
- `navigator.userAgent`, **property, 181–184**
- space symbol, 23**
- negation, defined, 72**
- nested**
 - elements in XML, 553
 - `try...catch` statements, 401–403
- Netscape**
 - the blur event and, 203
 - HTML page elements and, 490
 - troubleshooting the blur event, 203
 - Web standards and, 491–492
 - World Wide Web Consortium (W3C) and, 491
- new line, escape sequence, 23**
- `nextSibling` **property, Node object, 510–512, 515**
- `{n,m}`, **repetition character**, 304–305
- NN, escape sequence, 23**
- “No Plug-in or ActiveX Control” Redirection Script, testing your, 614**
- node**
 - child, 499
 - defined, 498
 - leaf, 499
 - parent, 499

Node object

Node **object**

- `appendChild(newNode)` method, 516–518
- base DOM object, 501
- `firstChild` property, 510–512
- `hasChildNodes()` method, 516–518
- `insertBefore(newNode, referenceNode)` method, 516–518
- `lastChild` property, 510–512
- methods of the, 516–518
- `nextSibling` property, 510–512
- `nodeName` property, 510–512
- `nodeType`, 510–512
- `nodeValue`, 510–512
- `ownerDocument` property, 510–512
- `parentNode` property, 510–512
- `previousSibling` property, 510–512
- properties of the, 509–515
- `removeChild(childNode)` method, 516–518
- `replaceChild(newChild, oldChild)` method, 516–518

NodeList **object, base DOM object, 501**

`nodeName` **property, Node **object**, 510–512**

`nodeType` **property, Node **object**, 510–512**

`nodeValue` **property, Node **object**, 510–512**

`<noembed/>`, **element, 608–614**

non-Internet Explorer browsers

- `target` property for, 448–450
- `type` property for, 447–448

non-Internet Explorer events, 446–450

`<noscript>` **tag, using the, 178**

NOT (!), logical operator, 72

notation

- `rgb()`, 463–464
- `url()`, 464

Notation **object, high-level DOM object, 502**

`Number()`, **function, 69**

Number Characters, regular expressions, 300–304

Number **object**

- overview, 129
- `toFixed()` method, 129–130

numbers

- binary, 2
- floating-point, 22
- mixing strings and, 44–45

numerical

- calculations and data, 35–39
- calculations example, 36–37
- data type, 22

O

`<object>` **tag, using the, 608–614**

object

- `ActiveXObject()`, 577–579
- `Array`, 107–109, 130–137, 156, 213, 233
- `Attr`, 502
- base DOM, 501
- `Button`, 194–195
- `CDATASection`, 502
- `Checkbox`, 208–214
- `Comment`, 502
- creating an, 107–109, 156
- `Date`, 107, 137, 138, 156, 157, 334–339
- defined, 105–106
- document, 165–169, 507, 512–513, 516–518, 668
- `DocumentType`, 502
- `Element`, 502, 508–509, 518–521
- `Entity`, 502
- `Entityreference`, 502
- event, 522–526, 528
- `Exception`, 395–397
- `Form`, 188–190, 233
- global, 161
- high-level DOM, 501–502
- history, 163, 185
- `img`, 672
- instance defined, 147
- JavaScript, 106–107
- location, 163–164, 185
- `Math`, 122, 123, 124–126, 127–129, 156, 663–667

- mouseEvent, 522–526
- NamedNodeMap, 501
- navigator, 164, 179–184, 185
- Node, 509–515, 516–518
- NodeList, 501
- Notation, 502
- Number, 129–130
- Option, 215–221, 677–678
- Plugin, 598–601
- primitives and, 110–111
- ProcessingInstruction, 502
- prototype, 632
- Radio, 208–214
- RegExp, 330
- screen, 164, 185
- String, 111–122, 156, 291, 295–296, 314–318
- style, 506–507
- summary, 156
- Text, 198, 502
- using JavaScript, 107–110
- window, 161–163, 249, 269–270
- window.event, 457–458
- XMLHttpRequest, 622–629
- object-based programming**
 - objects, 105–106
 - objects in JavaScript, 106–107
 - overview, 105
 - primitives and objects, 110–111
 - summary, 156
 - using JavaScript objects, 107–110
- object methods, calling and, 110**
- object properties, using, 109–110**
- offsetLeft, **property, 484**
- offsetTop, **property, 484**
- oHttp_readyStateChange(), **function, 628–629**
- onblur **event handler**
 - summary, 233–234
 - Text object, 198
- onblur **method, <form> elements, 193–194**
- onchange **event handler, Text object, 198**
- onclick
 - attribute, 170–172
 - event handler, 673–674
- one-shot timers, defined, 347**
- onfocus **event handler**
 - <form> elements, 193–194
 - summary, 233–234
 - Text object, 198
- onkeydown **event handler, Text object, 198**
- onkeypress **event handler, Text object, 198**
- onkeyup **event handler, Text object, 198**
- onload **event, exercise questions, 186**
- onload **event handler, page creation and the, 667–671**
- onmousedown **event, HTML Forms, 196–197**
- onmouseout, **event handler, 715**
- onmouseout **event, using the, 672**
- onmouseover, **event handler, 715**
- onmouseover **event, using the, 672**
- onmouseup **event, HTML Forms, 196–197**
- onreadystatechange, **event handler, 627–629**
- onselect **event handler, Text object, 198**
- onsubmit, **event handler, 723–726**
- open() **method**
 - summary, 288
 - using the, 626, 719–726
 - window object, 257–261
- opening**
 - code examples in Internet Explorer 6 and 7, 6
 - a new browser window, 257–261
 - new windows, 257–264
 - a page in Microsoft Script Debugger, 370–375
 - tags versus closing tags, 558
 - tags in XML (Extensible Markup Language), 552–553
- Opera**
 - creating an XMLHttpRequest object in, 624
 - DHTML (Dynamic HTML) toolbar, 540–544
 - DOM (Document Object Model) Standards and, 497
 - retrieving an XML (Extensible Markup Language) file in, 580

operating systems, plug-in and ActiveX control

differences between, 615

operations, basic string, 42–43

operators

AND (&&), 71

bitwise, 70

comparison, 64

comparison versus assignment, 66

decrement (—), 37–39

defined, 35

increment (++), 37–39

logical, 70–72, 101

NOT (!), 72

OR (||), 71–72

precedence, 39–42

precedence example, 40–42

(ophone) ?, **regular expression, 324–329**

Option object

HTML Forms, 215–221

properties of the, 215–216

using the, 677–678

options

adding with Internet Explorer, 220–221

adding new, 216–220

options[], **array, 220–221**

options[] **array, HTML Forms, 216**

Options dialog box, cookies and the, 413–414

OR (||), operator, 71–72

ownerDocument **property, Node object, 510–512**

P

<p/>, **elements, 468–470**

P3PEdit, policy creation software, 430

page

creating a, 667–671

creating a visitor count, 697–699

inserting an image in a, 671–672

viewing a, 387

page-level scope, defined, 96

<param/>, **elements, 605–614**

paragraph_eventHandler(), **function, 441–442**

parameter

defined, 11, 92

indexOf() **method, 117–118**

lastIndexOf() **method, 117–118**

passing, 93–94

parent node, defined, 499

parent **property, summary, 288**

parentNode **property, Node object, 510–512**

parseFloat(), **function, 46–48, 658–659**

parseInt(), **function, 46–48**

parsing

defined, 8

a web page example, 8–12

password text box, creating a, 203–204

path, cookie strings and, 416–417

pattern is case-sensitive, attribute for, 299

. (period), character class, 300–304

[...] (periods in brackets), character class, 300–304

Perl

regular expressions, 297

usability of, 4

PHP, downloading, 636

pixels, positioning and, 467–470

play(), **function, 715**

+ (plus sign), repetition character, 304–305

plug-ins

ActiveX control versus, 601–602

adding to the page in Firefox, 596–598

browser differences with, 614–615

checking for/installing in Firefox, 598–601

defined, 595

exercise questions, 618, 713–715

exercise solutions, 713–715

overview, 595–596

problems with, 614–617

RealPlayer, 595–596, 713–715

scripting differences, 615

summary, 617–618

using, 607–617

version differences, 616

Plugin **object, using the, 598–601**

pluginspage, **attribute, 598–601**

policy creation software, P3PEdit, 430

pop-up window

- adding a button/link to a, 681–686
- blocking, 257
- exercise questions, 289, 681
- exercise solutions, 681–686

positioning

- characters in regular expressions, 305–308
- content, 478–483
- of content, 467–470
- overview, 467–470

(pound sign), CSS (Cascading Style Sheets) and the, 471**pow() method**

- example, 127–129
- exercise questions, 157, 665
- exercise solutions, 665–666
- overview, 127

precedence

- defined, 39
- operator, 65

preventing, errors, 393–394

previousSibling property, Node object, 510–512, 515

primitives, objects and, 110–111**processed, defined, 1**

ProcessingInstruction object, high-level

DOM object, 502**prompt() function**

- exercise questions, 62, 658–659
- exercise solutions, 658–659
- using the, 41, 69

properties

- browser, 176–178
- class, 147
- of the Document object, 507
- of the Element object, 508–509
- event handlers as, 172–175, 455–458
- of the Node object, 509–515
- setting, 12
- styling, 462–470
- troubleshooting, 366
- versus methods errors, 366

property

- altKey, 522–526
- appName, 179–184
- async, 579–580, 717–726
- background-color, 463–464
- bubbles, 522–526
- buttons, 522–526
- cancelable, 522–526
- clientX, 522–526
- clientY, 523–526
- color, 460–461
- colorDepth, 164
- cookie, 422–428
- ctrlKey, 523–526
- currentTarget, 522–526
- documentElement, 507, 512–513
- document.lastmodified, 427–428
- elements[], 190
- elements [] array, 191–194
- eventPhase, 522–526
- firstzchild, 510–512
- font-family, 460–461, 464–465
- font-size, 460–461, 464–465
- font-style, 465, 474–475
- font-weight, 465
- form, 193, 233
- fromElements, 528
- height, 164
- href, 164, 186, 667–668
- index, 215–216
- lastChild, 510–512
- length, 107, 112, 130–131, 190, 233, 291
- name, 193, 233
- navigator.userAgent, 181–184
- nextSibling, 510–512, 515
- nodeName, 510–512
- nodeType, 510–512
- nodeValue, 510–512
- offsetLeft, 484
- offsetTop, 484
- ownerDocument, 510–512
- parent, 288

property (continued)

- parentNode, 510–512
- previousSibling, 510–512, 515
- readyState, 579–580
- relatedTarget, 523–526
- request, 629, 630
- responseXML, 629
- screenX, 523–526
- screenY, 523–526
- selectedIndex, 677–678
- shiftKey, 523–526
- srcElement, 442–446, 528
- status, 626–627
- style, 471–472
- tagName, 508–509
- target, 448–450, 522–526
- text, 215–216
- text-decoration, 465, 475
- toElement, 528
- top, 250–257, 288
- type, 233, 447–448
- userAgent, 179–184
- value, 193, 215–216, 234
- visibility, 466
- window.event.type, 442

prototype, **object**, 632

Punctuation Characters, regular expressions, 300–304

Q

? (question mark), repetition character, 304–305
questions, storing using arrays, 58–61

QuizPage.htm

- creating for the trivia quiz, 283
- trivia quiz, 274–275

“ (quote marks)

- escape sequence, 23
- strings and, 22

R

\r, **escape sequence, 23**
radCPUSpeed_onclick(), **function, 212**

radio buttons

- creating, 207–214, 229–233
- summary, 234

Radio **object, HTML Forms, 208–214**

random() **method, overview, 126**

readyState **property, using the, 579–580**

RealPlayer plug-in, using to create a page, 595–596, 713–715

recalcDateDiff(), **function, 226**

reference, defined, 108

RegExp() **constructor, RegExp object, 321–322**

RegExp() **object**

- exercise questions, 330, 686–689
- exercise solutions, 686–689
- overview, 291, 297
- RegExp() constructor, 321–322
- summary, 330
- test() method, 302–304
- using the, 298–311

regExpIs_valid(), **function, 302–304**

regular expressions

- defined, 291
- exercise questions, 330–331, 687–689
- exercise solutions, 687–689
- grouping, 308–311
- Number Characters, 300–304
 - (ophone)?, 324–329
- overview, 297
- position characters, 305–308
- repetition characters, 304–305
- Sax, 324–329
- simple, 297–300
- summary, 329–330
- Text, Numbers, and Punctuation Characters, 300–304
- trivia quiz and, 322–329

relatedTarget **property, mouseEvent object, 523–526**

relative positioning, defined, 467

remote scripting

- class creation, 629–634
- defined, 619
- exercise questions, 716–726

exercise solutions, 716–726
 XMLHttpRequest constructor, 629–631
 methods, 631–632
 uses of, 619–622
 remove() **method**, options[] **array**, 220–221
 removeAttribute(attributeName) **method**,
 Element **object**, 519–521
 removeChild(childNode) **method**, Node
 object, 516–518
 removeItem(), **function**, 683
 removing, elements from a document, 516–518
 repetition characters, regular expressions,
 304–305
 replace() **method**
 location object, 164
 String object, 295–296, 314–318
 summary, 329
 using the, 670–671
 replaceChild(newChild, oldChild)
 method, Node **object**, 516–518
 replaceQuote(), **function**, 317–318
 request **property**, XMLHttpRequest **object**,
 629
 request_readystatechange(), **function**,
 630–631, 720–726
 reserved words, defined, 25
 Reset button, creating a, 197
 reset() **method**
 Form object, 190
 summary, 234
 resetQuiz(), **function**, 355–359
 resizable, window feature, 262–264
 resizeBy() **method**
 summary, 288
 window object, 270
 resizeTo() **method**
 summary, 288
 window object, 269–270
 resizing, windows, 269–270
 responseXML, **property**, 629
 returnPagesVisited(), **function**, 245–247
 reverse() **method**, using the, 135–137

rgb() **notation**, Cascading Style Sheets (CSS),
 463–464
 right operand, defined, 65
 root element, defined, 554
 root node, defined, 498
 round() **method**, 123
 rounding
 functions example, 124–126
 methods summary, 123–124
 ROWS
 attribute, 204–207
 summary, 234
 rows **attribute**, frames, 240–241
 rules, style, 567
 running code, defined, 1
 Running Documents Window, Microsoft Script
 Debugger, 386–388

S
 \S, character class, 300–304
 Safari
 creating an XMLHttpRequest object in, 624
 DOM (Document Object Model) Standard and,
 497
 iframes and, 645
 Safari 2, XML (Extensible Markup Language) sup-
 port and, 577
 Same-Origin Policy, Ajax, 652
 Sax, regular expression, 324–329
 scope
 defined, 96
 summary, 102
 <script/>, elements, 703
 screen color depth, defined, 165
 screen **object**
 summary, 185
 using the, 164
 screenX **property**, MouseEvent **object**,
 523–526
 screenY **property**, MouseEvent **object**,
 523–526
 </script>, close tag, 8

script block, defined, 6

Script Debugger. See **Microsoft Script Debugger**

script language, browser defaults, 7

<script> tag

- attributes of, 7
- overview, 6–7
- using the, 6–12

scripting

- frames, 251–257
- between windows, 264–269

scrollbars, **window feature, 263–264**

search() method

- String object, 296, 318
- summary, 329

secondCall(), **function, 385**

secure, cookie strings and, 418

Secure Sockets Layer (SSL), cookies and, 418

security, frames and windows, 270–271

<select>/</select> tags

- HTML Forms, 215
- summary, 235

select elements

- adding new options, 216–220
- events, 221–227
- HTML Forms, 215–227
- <select>/</select> tags, 215

selectedIndex **property, using the, 677–678**

selector, defined, 459

;(semicolon)

- requirements for the, 660
- using the, 8

send() method

- exercise questions, 718–726
- exercise solutions, 719–726
- remote scripting, 629
- XMLHttpRequest object, 626–627, 631–632

Sequence, Unicode/Common escape, 23

server, client versus, 651

server response, iframe, 645

server-side processing, defined, 187

session

- cookies defined, 428
- defined, 416

setAttribute(attributeName,value)

method, Element object, 519–521

setCookie(), **function, 420–422, 431–434**

setFrameAndPageControls(), **function, 255–257**

setHours() **method**

- Date object, 334–339
- using the, 145–146

setInterval() **method**

- summary, 360
- using the, 352–353, 691

setMilliseconds() **method, using the, 145–146**

setMinutes() **method, using the, 145–146**

setSeconds() **method, using the, 145–146**

SetSource(), **method, 613–614**

setSource() **method, using the, 715**

setTimeout(), **method, 352**

setTimeout() **method**

- summary, 360
- using the, 348–352

setting

- attributes, 518–521
- changing a time, 340
- the UTC date/time, 339–348

setUTCDate() **method, using the, 339–348**

setUTCFullYear(), **using the, 339–348**

setUTCHours()

- Date object, 334–339
- method, 339–348

setUTCMilliseconds() **method, using the, 339–348**

setUTCMinutes() **method, using the, 339–348**

setUTCMonth() **method, using the, 339–348**

setUTCSeconds() **method, using the, 339–348**

SGML (Standard Generalized Markup Language), origination of, 490–492

shiftKey **property, mouseEvent object, 523–526**

Show Cookies button, Firefox 2.0, 415

showBasket(), **function, 684**

showBoxOne(), **function, 703**

showBoxTwo(), **function, 703**

showing, elements, 465–466**slice() method**

- parameters of the, 132
- using the, 132–133

software, P3Edit policy creation, 430**sort() method**

- Array objects and the, 107
- sorting an array with the, 664
- using the, 134–135

sorting

- an array, 665
- names, 664–665

split() method

- String object, 292–295, 312–314
- summary, 329

splitAndReverseText(), function, 294–295**splitting**

- HTML, 319–321
- the string, 292–295

src, attribute, 151**srcElement property**

- event object, 528
- events and the, 442–446

SSL (Secure Sockets Layer), cookies and, 418**Standard Generalized Markup Language (SGML),
origination of, 490–492****startTimer(), function, 705–707****statement**

- break, 80, 91, 102
- case, 80
- continue, 91–92, 102
- defined, 8
- else, 77–78
- else if, 78
- for, 84–88
- if, 66–70, 73–76, 100, 101
- nested try...catch, 401–403
- switch, 79–84, 100, 101
- try...catch, 394–397, 401–403
- while, 84–88

static, defined, 12**status**

- property, 626–627
- window feature, 263–264

Step Into, icon, 374, 375**Step Out, icon, 375****Step Over, icon, 376****storing**

- data in memory, 24–25
- DHTML (Dynamic HTML) toolbar button information, 532–533
- exercise questions, 157
- names, 664–665
- the results of a comparison, 66

string manipulation

- exercise questions, 686–689
- exercise solutions, 686–689

string methods, additional, 292–296**String objects**

- charAt()/charCodeAt() methods, 112–116, 291
- fromCharCode() method, 117
- indexOf()/lastIndexOf() methods, 117–120, 291
- length property, 112, 291
- match() method, 296, 318–319
- overview, 111–112
- replace() method, 295–296, 314–318
- search() method, 296, 318
- split() method, 292–295, 312–314
- substr()/substring() methods, 120–121, 291
- toLowerCase()/toUpperCase() methods, 121–122, 291

strings

- basic operations, 42–43
- changing the case of, 121–122
- comparing, 78–79
- concatenating example, 42–43
- converting arrays into single, 133–134
- converting character codes to, 117
- copying, 120–121
- copying part of, 120–121
- defined, 22
- finding strings inside other, 117–118
- mixing numbers and, 44–45
- non-convertible, 48–49
- “ (quote marks) and, 22

strings (continued)

- regular expressions, 297–311
- searching and replacing in, 295–296
- selecting a single character from, 112–113
- splitting, 292–295
- sUrl, 630–631
- <style> **tag, adding style with the, 459–461**
- style, **attribute, 461**
- style**
 - declarations defined, 459
 - rules, 567
 - sheets and XML (Extensible Markup Language), 566–568
- style **object, using the, 506–507**
- style **property, using the, 471–472**
- styling**
 - backgrounds and color, 463–464
 - buttons, 529–532
 - colors and background, 462
 - the DHTML toolbar, 529–532
 - fonts and text, 464–465
 - foreground color, 462–463
 - icons, 532
 - position, 466–470
 - showing and hiding elements, 465–466
- submit, **event, 482**
- Submit button, creating a, 197**
- submit() **method**
 - Form object, 190
 - summary, 234
- substr() **method, using the, 120–121, 245–247, 291**
- substring, defined, 117**
- substring() **method, overview, 120–121, 291**
- sUrl, **string, 630–631**
- switch **statement**
 - elements of the, 79–80
 - example, 80–84
 - summary, 100, 101
- switchDirection, **global variable, 486–488**
- switchImage(), **function, 352**
- syntax, errors versus logic errors, 693–696**

T

- \t, **escape sequence, 23**
- tab, escape sequence, 23**
- tag**
 - </script>, 8
 - <A>, 170–175
 -
, 665
 - closing in XML (Extensible Markup Language), 552–553
 - <form>/</form>, 188–189, 233, 723–726
 - <frame>/<frameset>, 238–257
 - , 186, 671–672
 - <input>, 197–203
 - <noscript>, 178
 - <object>/<object/>, 608–614
 - <script>, 6–12
 - <select>/</select>, 215, 235
 - <style>, 459–461
 - <textarea>, 204–207
 - <textarea>/</textarea>, 204–207, 234
- tagName, **property, 508–509**
- target, **attribute, 188–189**
- target **property, event object, 522–526**
- <tbody/>, **elements, 591–592**
- temperature conversion, calculating, 657–658**
- temperature converter, example, 44–45**
- template**
 - rule defined, 570
 - rule in XML (Extensible Markup Language), 570
- Temporary Internet Files and History Settings dialog box, 410**
- test(), **method, 302–304**
- test condition, for statement, 85**
- test expression, switch statement, 80**
- testing**
 - browsers order, 625
 - your “No Plug-in or ActiveX Control” Redirection Script, 614
- <textarea>/</textarea> **tag**
 - summary, 234
 - textarea element, 204–207

text

- comment, 556
- data type, 22–23
- editor JavaScript tool, 4

text box

- creating a, 197
- creating a hidden, 204
- creating a password, 203–204

Text Characters, regular expressions, 300–304

text-decoration, **property**, 465, 475

Text Elements

- creating a text box, 197–198
- select() method, 197–198
- summary, 234
- Text object, 197–198

text/fonts, styling, 464–465

Text object

- event handlers, 198
- high-level DOM object, 502

text **property**, Option **object**, 215–216

textarea **element**, HTML Forms, 204–207

<thead/>, **elements**, 590–592

thirdCall(), **function**, 385

throw **statement**, error handling, 397–401

time

- exercise questions, 360
- overview, 333
- settings, 340
- world, 334

timers

- clearing, 349–352
- exercise questions, 689–693
- exercise solutions, 689–693
- one-shot, 347–352
- overview, 333
- regular interval, 352–353
- summary, 360
- Web Page, 347–352

times tables, calculating, 660–663

timestamp **property**, event **object**, 522–526

toDateStr() **method**

- Date object, 334–339
- overview, 138

toElement **property**, event **object**, 528

toFixed() **method**, overview, 129–130

toGMTString(), **method**, 416

toLocaleDateString() **method**, Date **object**, 334–339

toLocaleString() **method**, Date **object**, 334–339

toLocaleTimeString() **method**, Date **object**, 334–339

toLowerCase() **method**, overview, 121–122, 291

toolbar

- creating a, 538–540
- DHTML, 528–540
- toolbar, **window feature**, 263–264

top

- property summary, 288
- property window object, 250–257
- window feature, 263–264

toString() **method**

Date object, 334–339

using the, 142–145

toUpperCase() **method**, overview, 121–122, 291

toUTCString() **method**, Date **object**, 334–339

<tr/>, **elements**, 590–592

tree structure

- creating/navigating a, 708–710
- overview, 498–499

Trivia Quiz

- building the functionality of the, 18
- creating the answer radio buttons, 229–233
- creating AskQuestion.htm for the, 283–284
- creating the form, 228–229
- creating fraGlobalFunctions for the, 280–283
- creating fraMenubar, 279
- creating the fraQuizPage, 274–278
- creating fraTopFrame, 278–279
- creating a function for the, 97–100
- creating GlobalFunctions.htm for the, 284–287
- creating QuizPage.htm for the, 283
- creating a timer-based, 354–359

Trivia Quiz (continued)

Trivia Quiz (continued)

- creating the top window frameset page, 273–274
- design and programming overview of the, 16–18
- frames, 272–273
- overview, 13–16
- regular expressions and the, 322–329
- storing questions using arrays, 58–61

Trivia Quiz Files

- `AskQuestion.htm`, 283–284
- `GlobalFunctions.htm`, 284–287
- `QuizPage.htm`, 283

troubleshooting

- ActiveX, 652–653
- ActiveX controls, 614–617
- case sensitivity, 363–364
- closing braces, 364
- closing parentheses, 365–366
- concatenation, 364–365
- `Equals/Is Equal To`, 365
- exercise questions, 102
- Firefox, Netscape, and the `blur` event, 203
- methods, 366
- properties, 366
- variables, 362–363

truth table

- AND operator and `a`, 70
- NOT operator and `a`, 72
- OR operator and `a`, 72

`try...catch` statements

- error handling and, 394–397
- nested, 401–403

`txtAge_onblur()`, function, 201–203

`txtName_onchange()`, function, 201–203

type, attribute, 7

type property

- event object, 522–526
- events and the, 439–442
- for non-Internet Explorer browsers, 447–448
- summary, 233

U

undefined, variables, 362–363

`unescape()`, function, 419–420

Unicode, escape sequence, 23

`updateOrderDetails()`, function, 677–678, 679–680

`updateTime()`, function, 345–347, 353

`updateTimeLeft()`, function, 356–359

`updateTimeZone()`, function, 344–347

`url()` notation, CSS (Cascading Style Sheets), 464

usability, concerns, 653–654

user base, defined, 13

user interaction, DHTML toolbar, 534–537

user interface

- creating a, 672–678

- exercise questions, 235, 672–678

- exercise solutions, 672–678

`userAgent` property, navigator object and the, 179–184

UTC (Coordinated Universal Time)

- defined, 334

- setting the date, 339–348

- setting the time, 339–348

- summary, 360

- time summary, 360

V

valid

- defined, 559

- documents defined, 559

validating, XML (Extensible Markup Language) documents, 564

value, attribute storage versus arrays, 677

value

- of cookie strings, 415–416

- of cookies, 422–428

- hexadecimal, 463

- Nan, 48

value property

- `<form>` elements, 193

- `Option` object, 215–216

- summary, 234

variables

- advantages of, 24–25

- assigning values to, 26

- declaring, 25
- defined, 24
- example on assigning variables the values of
 - other, 28–29
- example on declaring, 26–27
- exercise questions, 62
- scope summary, 102
- storing data in memory, 24–25
- troubleshooting, 362–363
- undefined, 362–363
- valid/invalid names of, 25

VBScript

- matching JavaScript and, 309–310
- regular expressions, 297
- usability of, 4

Venkman

- downloading, 388
- Firefox Debugging with, 388–393
- installing, 388–389
- summary, 404

| (vertical bar), alternation character, 309–310

viewing

- cookies in Firefox, 413–415
- cookies in Internet Explorer, 408–413
- the JavaScript console, 362
- pages, 387

visibility, **property**, 466

W

\W, character class, 300–304

W3C (World Wide Web Consortium)

- goals of the, 491
- guidelines for browsers, 160
- origination of, 491–492
- Web site, 492

Web, JavaScript and the, 2–3

web browsers

- back button, 653
- checking for different, 179–184
- compatibility of, 12–13
- differences between, 13
- differences with plug-ins and ActiveX controls, 614–615

- displaying errors in, 30–34
- example for changing background color in the, 7–8
- JavaScript tool, 5
- script language defaults, 7
- summary, 186
- testing order, 625
- version checking, 175–184
- version compatibility, 12
- W3C guidelines and, 160

web page

- adding a border to a, 710
- creating a frames-based, 678–680
- creating links on a, 701–703
- creating with the RealPlayer plug-in/ActiveX control, 713–715
- example of parsing a, 8–12
- inserting advertisements into a, 690–691
- inserting JavaScript into a, 6–7
- timers, 347–352

web page events. See also events

- event handlers as attributes, 170–172
- event handlers as properties, 172–175
- overview, 169–170

web server

- defined, 3
- installing a, 635–636

web space, free, 417

Web standards

- DOM (Document Object Model), 496–497
- ECMAScript, 492–493
- HTML, 492
- overview, 490–492
- summary, 548
- XHTML, 494–495
- XML, 493–494

well-formed documents, defined, 552

while loop

- overview, 88–90
- summary, 101
- using the, 662–663

while statement, radio buttons, 232

width, window feature, 263–264

width property, screen object, 164

window

- adding features to a, 262–264
- adding HTML to a new, 261–262
- Call Stack, 383–386
- Command, 381–383
- exercise questions, 289
- moving and resizing a, 269–270
- opening a new, 257–264
- overview, 237, 257
- Running Documents, 386–388
- scripting between, 264–269
- security, 270–271

window **object**

- frames and the, 238–257
- overview, 161–162
- parent/child, 239
- summary, 185
- top property, 250–257
- using the, 162–163

windowarrayonload(), **function, 189–190**

window.close() **method, summary, 288**

window.event, **object, 457–458**

window.event.type, **property, 442**

windowonload(), **function, 225–226, 241**

window_onload(), **function, 352, 353, 715**

window_onunload(), **function, 269**

window.open() **method, summary, 288**

Windows 2000, script debuggers and, 367

World Time

- Converter example, 341–347, 353
- overview, 334

World Wide Web Consortium (W3C)

- goals of the, 491
- origination of, 491–492
- Web site, 492

wrap, **attribute, 204–207**

writeMonthOptions(), **function, 225**

writeOptions(), **function, 224**

writeTimesTable(), **function, 373–374, 390–393, 661–662**

X

XHTML

- HTML versus, 495
- Web standard, 494–495, 548

XML (Extensible Markup Language)

- building an application, 582–592
- CDATA, 556
- character data, 555–556
- closing tags, 552–553
- comments, 556–557
- comments versus HTML comments, 557
- creating a document, 557–566
- displaying, 566–576
- document structure, 554
- exercise questions, 593, 710–713
- exercise solutions, 710–713
- formatting, 566–576
- manipulating with JavaScript, 576–592
- nesting elements, 553
- opening tags, 552–553
- overview, 551–552
- schema defined, 494
- style sheets and, 566–568
- summary, 592–593
- template rule, 570
- Web standard, 493–494
- Web standards summary, 548
- as a well-formed language, 552
- XML (Extensible Markup Language) elements, 554–555
- XSLT transformations, 568–576

XML (Extensible Markup Language) documents

- adding data to, 561–566
- creating, 711
- creating a DTD file, 559–561
- Document Type Definition (DTD), 559
- exercise questions, 593
- linking to XSL style sheets, 574–576

XML (Extensible Markup Language) file

- cross-browser retrieval of an, 581–582
- exercise questions, 593
- loading an, 579–581, 711–713
- retrieving in Firefox and Opera, 580
- retrieving in Internet Explorer, 577–579

xmlDoc_readyStateChange(), **function, 580**

XMLHttpRequest **object**

- creating in Internet Explorer 5/6, 622–624
- creating in Internet Explorer 7, Firefox, Opera, and Safari, 624

XMLHttpRequest object

- asynchronous requests, 627–629
- creating a form with, 634–644
- cross-browser issues, 622–625
- Internet Explorer 5 and 6 and the, 622–625
- other browsers, 624–625
- overview, 622
- summary, 654–655
- using the, 626–627

\xNN, escape sequence, 23

XSL style sheets, linking XML (Extensible Markup Language) documents to, 574–576

XSLT

- defined, 568–570
- overview, 568–570
- template example, 571–574
- template rules, 570–574
- transformations and formatting XML (Extensible Markup Language) documents, 568–576

Z

Zip, files, 552